Hindawi
Scientific Programming
Volume 2022, Article ID 6350686, 9 pages
https://doi.org/10.1155/2022/6350686

Research Article

Code2tree: A Method for Automatically Generating Code Comments

Wanzhi Wen, Jiawei Chu, Tian Zhao, Ruinian Zhang, Bao Zhi, and Chenqiang Shen

School of Information Science and Technique, Nantong University, Nantong 226019, China

Correspondence should be addressed to Wanzhi Wen; wenwanzhi@126.com

Received 4 May 2022; Accepted 10 August 2022; Published 29 September 2022

Academic Editor: Francisco Ortin

Source code comments can improve the efficiency of software development and maintenance. However, due to the heterogeneity of natural language and program language, the quality of code comments is not so high. So, this paper proposes a novel method Code2tree, which is based on the encoder-decoder model to automatically generate Java code comments. Code2tree firstly converts Java source code into abstract syntax tree (AST) sequences, and then the AST sequences are encoded by GRU encoder to solve the long sequence learning dependency problem. Finally, the attention mechanism is introduced in the decoding stage, and the quality of the code comment is improved by increasing the weight of the key information. We use the open dataset java-small to train the model and verify the effectiveness of Code2tree based on common-used indicators BLEU and $F1$-Score.

1. Introduction

For large-scale software systems, software development and maintenance personnel face the problems of poor code annotation quality, annotation missing, and mismatching of annotations and code. So, software developers and maintainers spend a lot of time on understanding the program [1]. Good code comments can help developers and maintainers understand the program more accurately and faster, so as to save a lot of reading time [2]. The automatic code comment technique aims to reduce the workload of developers to write annotations, assist developers to understand the code better, and improve the efficiency of software development and maintenance.

With the continuous expansion of the scale of software code, how to help developers understand and maintain the code during software developing and maintenance has become an important topic in the field of software engineering. Information retrieval is the earliest technique used in the research of automatic code comment generation. Based on information retrieval techniques, review algorithms generally use related techniques such as vector space model (VSM), latent semantic indexing (LSI), latent Dirichlet allocation (LDA), or code clone detection. Bai et al. [3] use a code cloning detection technique to find similar code fragments and copy the comments in the similar code fragments to the target code. However, information retrieval implements similar code comments through annotation migration. The quality of the generated annotations depends on the accuracy of the code similarity measurement and the completeness of the annotations in the dataset to be retrieved. It will not be generated good comments due to the inaccurate code similarity measurement or the incomplete annotations of the retrieved data [4].

In recent years, with the development of deep neural networks in natural language processing, machine translation [5], question answering [6], and speech processing [7], deep neural networks have also been introduced in the field of software engineering to solve the problem of code comment. People introduce the neural machine translation model NMT (neural machine translation) into the task of code understanding. Iyer et al. [8] use the recurrent neural network (RNN) as the encoder to directly encode the code sequence to obtain its intermediate vector, and then use the decoder to translate the intermediate vector into code comments and introduce an attention mechanism. However, the code is

different from the general text and it has more structural information and complex semantic understanding. The process of machine translation from the code sequence to the comment sequence loses too much code internal information, which leads to the inability to accurately understand the semantic information and generates bad comments. In order to extract code semantic information, researchers proposed a code semantic extraction method based on an abstract syntax tree. Hu et al. [9] use the abstract syntax tree AST (Abstract Syntax Tree) as code presentation. The tree structure is transformed into a linear sequence of nodes, and the Long-Short Term Memory (LSTM) neural network is used as an annotation for the code generation method of the encoder and decoder. Huang et al. [10] use the abstract syntax tree traversal of code snippets. Then, token sequences are generated. Finally, the single-layer gated recurrent unit neural network GRU (Gated Recurrent Unit) is constructed and the annotation of the code block is generated by the encoder and decoder. Alon et al. [11] first analyze each Java method in the corpus and build an abstract syntax tree (AST). Then, they traverse the AST and extract the syntactic path between the AST leaves. Finally, they calculate the learning weighted average of the path vector by attention mechanism to generate the code vector.

In this paper, based on the machine translation framework NMT, we propose a neural network model code2tree that combines source code structure and semantics, an alternative method of encoding source code using the grammatical structure of a programming language. Our model represents code fragments as a set of combined paths of an abstract syntax tree (AST), and each path is compressed into a fixed-length vector [12]. We use GRU as the decoder, and the different weighted averages of the path vectors are processed by the attention mechanism during the decoding process to generate each output token. Java is one of the most popular and major programming languages, so this paper selects Java language as our research subject which aims to improve Java code comments.

Code2tree generates reviews verbatim from AST sequences. Based on an open dataset on GitHub, we trained and evaluated the performance of Code2tree. During the experiment, we generally trained across multiple items and make predictions for different items. Our experimental results show that code2tree can generate higher quality code comments, and the effectiveness is better than the existing methods.

The main contributions of this paper are as follows:

Firstly, we convert the automatic generation of code comments into machine translation problems and construct the neural network machine translation model (NMT) for program understanding.

Secondly, we propose a Code2tree method, which extracts structural information from source code based on a sequence-to-sequence model to generate annotations for the java program. In particular, we directly use the path in the abstract syntax tree (AST) for end-to-end sequence generation.

The rest of this paper is organized as follows. Section 2 introduces the language model and background knowledge of NMT. Section 3 elaborates on the details of Code2tree. Section 4 presents the experimental setup, results, and threats to its effectiveness. Section 5 discusses our works. Section 6 gives the related work. Section 7 summarizes the paper and points out future directions.

2. Preliminary

2.1. Language Models. The seq2seq model [13] used in this paper is a variant of a recurrent neural network, which includes two parts: encoder and decoder. The model has achieved great success in automatic summarization, dialogue systems, and machine translation. We apply the model to the task of automatically generating code comments. The model can generate code comments by capturing long-term dependencies between languages, such as grammatical structures. As shown in Figure 1, the frame structure of the seq2seq model is shown in detail.

As a core component of a natural language processing (NLP) system, language models can provide word representations and the maximum likelihood of word sequences. The language model describes the probability of words appearing in the sequence. For a sequence of n natural language$x = (x_1, \cdots, x_n)$, the language model is based on the known sequence$(x_1, \cdots x_{k-1})$, and we predict the probability of the next wordx_k as a formula:

$$p(x_1, \cdots, x_n) = \prod_{k=1}^{n} p(x_k \mid x_1, \cdots, x_{k-1}). \tag{1}$$

When modeling language models, in order to reduce the dimensionality disaster during modeling, Brown et al. [14]. proposed an approximate method of N-gram language models, that is, the appearance of the predicted k word only depends on the previous k-1 words as a formula:

$$p(x_k \mid x_1, \cdots, x_{k-1}) \approx p(x_k \mid x_{k-n+1}, \cdots, x_{k-1}). \tag{2}$$

However, this N-gram method has obvious limitations. For example, it does not consider the discreteness, combination, and sparsity of natural language. To solve this problem, deep neural networks are introduced into the training of language models, such as recurrent neural networks (RNNs), long short-term memory (LSTM), and gated recurrent unit (GRU). The language model used in this paper is based on the deep neural network gated recurrent unit (GRU).

(1) Recurrent neural networks. The RNN consists of three layers. Each input is mapped to the input layer in the vector. On the cyclic hidden layer, the model cyclically calculates and updates the hidden state after reading each input vector. On the output layer, the model uses the hidden state to calculate the predicted token probability. During the training process of RNN, the gradient size of each layer will increase or decay exponentially on the long sequence. The problem of gradient explosion or disappearance makes it difficult for RNN models to learn long-distance correlations in sequences.

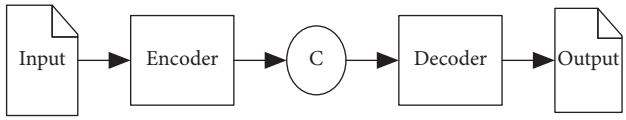

Figure 1: The sequence generating process.

To solve these problems, some researchers have proposed several variants to maintain long-term dependence. These variants include LSTM and gated recurrent unit (GRU). In this paper, we adopt the GRU that has been successful on many NLP tasks.

(2) Long short-term memory. LSTM is an effective chain loop neural network by introducing a memory unit that can save the state for a long time, so the problem of long-term dependence on learning can be effectively solved. LSTM can be divided into single-layer and multilayer. The two-layer LSTM in this paper can input the input sequence from two directions. The double feed information input can improve the long memory ability of LSTM so that the possibility of forgetting the information is reduced. Because the information is provided from both sides of the input, the model reduces the propagation distance of the information.

(3) Gated recurrent unit. GRU is a variant of LSTM. GRU combines the forget gate and input gate into a single update gate. It also mixes the cell state and the hidden state and adds some other changes. The GRU model is simpler than the standard LSTM model. The GRU model is one less gate than LSTM, so there are fewer matrix multiplications. In the case of large training data, GRU can save a lot of time.

2.2. Neural Machine Translation. Neural machine translation (NMT) is used to translate words from one language to another. The model uses neural networks to predict the possibility of word sequences and it is usually in the form of entire sentences. Unlike statistical machine translation, which consumes more memory and time, neural machine translation trains its parts end-to-end to maximize performance.

NMT has been widely adopted in multinational organizations to help them communicate internally and externally. The NMT system is rapidly developing to the forefront of machine translation and has recently surpassed the traditional form of the translation system. The main advantage of this method is that a single system can be trained directly on the source text and the target text, and the pipeline of a dedicated system for statistical machine learning is no longer needed.

Previously, machine translation was trapped by the multilayer perceptual neural network model, which was limited to fixed-length sequences and the output must have the same length. Nowadays, the model has updated the position and added the attention mechanism to learn how to focus on the input sequence when decoding each word in the output sequence. So, the model can improve the translation performance of long word sequences.

Unlike traditional machine translation methods that involve individually designed components, NMT can work closely together to maximize its performance. In addition, NMT also uses vector representations to describe words and internal states. During the vector representations stage, words are transcribed into vectors defined by unique sizes and directions. Generally, when a number sequence is given, NMT uses an artificial neural network to predict the number sequence. NMT encodes each word into a sequence of numbers, which represents the target sentence for translation. NMT also uses a two-way recurrent neural network (encoder) to process the source sentence into a vector of a second recurrent neural network (decoder). This process is better in terms of speed and accuracy.

3. Proposed Approach

One of the key factors in generating high-quality code annotations is constructing accurate mapping relationships between source code and natural language. The direct method is to convert the problem into a machine translation problem, in which the source sentence is the token sequence of the code, and the target sentence is the corresponding comment sequence. Iyer et al. [8] proposed an LSTM-based annotation generation model CodeNN, which uses LSTM with an attention mechanism to generate annotations for C# and SQL. CodeNN directly aligns the words in the comments with the relevant code tokens by adding an attention mechanism. Allamanis et al. [15] used convolutional neural networks (CNNs) and attention mechanisms to generate a summary of the source code. This method uses the convolution attention module to extract features from the input source code and determine the important tokens that should be paid attention to in the sequence. In addition, some papers model the source code as a series of tokens [16] and characters [17]. These tasks have achieved very good performance in generating code comments and documentation. There are also some methods that take into account the structural information of the source code. Liang and Zhu [18] proposed an AST encoder Code-RNN based on RNN. Chen and Wan [19] proposed a tree sequence model Tree2Seq for code comment generation. Tree2Seq uses an AST-based encoder instead of the RNN encoder.

There are some existing tools generating code comments with transformer models. The DeepCom plugin generates code comments based on AST techniques, but generated comments can cause OOV problems. Tools based on the transformer model are great for generating comments for long code sequences.

Compared with the NMT model that only translates natural language, the NMT model in this paper is a language model constructed based on Java source code and its corresponding source code comments. The words in the comments are consistent with the hidden state of the RNN involving the semantics of code markup. Code2tree extracts code semantic information (such as key sentences, symbols, and keywords) from a large Java corpus, separates the

identifiers and symbols in the code and automatically generates code comments after multiple model training. However, due to the difference between programming language and natural language, the migration of neural network models from natural language processing to annotation generation still faces a series of difficulties as follows:

(1) A large number of low-frequency code vocabulary easily generate low-quality of code comments. In the neural machine translation model NMT, the vocabulary is usually limited to 30,000 to 50,000 words, and words beyond the vocabulary will be treated as unknown words-usually marked as UNK. This is very effective in the general natural language translation model. In the code corpus, the vocabulary is usually composed of keywords, operators, and identifiers. Developers often define various new identifiers. A large number of low-frequency vocabulary will be generated, resulting in low-quality of automatically generated code comments.

(2) The semantics of program language is different from that of natural language. The behavior represented by the code has nothing to do with the words of the code but is determined by the keywords and structure. The structuring of programming language is also reflected in language rules, semantic expression, and contextual information. In addition, many NMT models are based on sequence-to-sequence (seq2seq) models, which have certain difficulties in analyzing the semantic information of programming languages.

For vocabulary issues, Code2tree transforms Java methods into abstract syntax trees (ASTs). For each abstract syntax tree (AST), we use depth-first search to traverse it. Specifically, during AST extraction, we did not describe each variable using common symbols but directly abstracted the actual name of the variable. Based on this, we reduced the impact of a large number of unknown words to a certain extent.

To solve the problem of programming language structure, Code2tree uses a sequence-based model to parse the abstract syntax tree (AST) to obtain the structure information of the Java method. For each abstract syntax tree (AST), we use the depth-first search method to abstract the code information. Finally. Java source code is converted into serialized data for neural network learning using Code2tree.

The overall framework of Code2tree is shown in Figure 2. The Code2tree method mainly includes three stages, namely: data preprocessing, model training, and online testing. We parse and process the Java-small data set downloaded from GitHub into a parallel Java method corpus and its corresponding annotations. Before the Java method is input to the model, in order to learn the structural information of the Java method, we convert the Java method into an AST sequence through an abstract syntax tree. Based on the generated AST sequence and the parallel corpus of comments, we build and train generative neural models based on the idea of NMT. There are two challenges in the training process as follows:

(1) How to use AST to store the structure information of Java methods?

(2) How to deal with extraword tags in the source code?

In the following paragraphs, we will introduce the details of the model and the methods we propose to solve the above challenges.

3.1. Sequence-to-Sequence Model. In this paper, we use a sequence-to-sequence (Seq2Seq) model to learn source code information and generate annotations. The sequence-to-sequence (Seq2Seq) model has achieved great success in machine translation, speech recognition, and text summarization. The model consists of three components, namely, an encoder, a decoder, and an attention component. Among them, the encoder and decoder are mainly based on GRU. The GRU model is simpler than the standard LSTM model. The GRU model is one less gate than LSTM, so there are fewer matrix multiplications. In the case of large training data, one less gate can save a lot of time and solve the problem of gradient disappearance. GRU encoder maps input sequence tokens $x = (x_1, \cdots, x_n)$ to a sequence of continuous representations $s = (s_1, \cdots, s_n)$.Given s,the GRU decoder then generates a series of output tokens $y = (y_1, \cdots, y_t)$, generating one token at a time and modeling the conditional probability $p(y_1, \cdots, y_t | x_1, \cdots, x_n)$.

At each decoding step, the probability of the next target token depends on the previously generated token, so it can be decomposed as formula

$$p(y_1, \ldots, y_t \mid x_1, \ldots, x_n) = \prod_{j=1}^{t} p\left(y_j \mid y_{<j}, s_1, \ldots, s_n\right). \tag{3}$$

3.2. AST Encoder. Abstract syntax tree (AST) is used to represent Java source code fragments. The leaves of the tree are called terminals, which usually represent user-defined values. Nonleaf nodes are called nonterminal symbols, which represent a set of restricted structures in the program, such as loop statements and expressions. Given a set of AST paths $\{x_1, \cdots, x_k\}$, for each set of AST paths x_i, create a vector z_i to represent each AST path consisting of nodes and their subindexes, which come from a limited vocabulary of up to 364 symbols. On the embedding layer, the shape (batch_size, vocab_size, embd_dim) is output, where batch_size is the batch size, vocab_size is the vocabulary size, and embd_dim is the embedding dimension, that is, the vector of the word embedding dimension corresponding to each word. By embedding each feature of each word, a vectorized representation of each feature can be obtained. Then the GRU unit is used for AST coding. The GRU maps the input $(z_i, \cdots, z_n)$ of this layer to the hidden state $(h_1, \cdots, h_n)$. At time step t, the hidden state h_t is recursively updated according to the current input z_t and the state h_{t-1} of the previously hidden layer as formula:

$$h_t = f_{\mathrm{GRU}}(h_{t-1}, z_t). \tag{4}$$

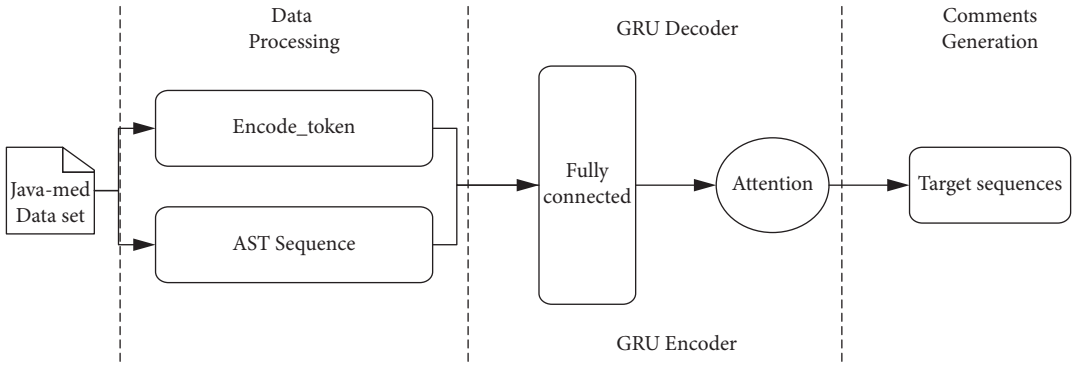

Figure 2: The framework of Code2tree.

Figure 3 is a simple example of a Java method. Figure 4 is the abstract syntax tree of the Java method in Figure 3. Each node is a statement, and the edge is the relationship between the nodes. For the name of the variable, we directly extract the name in the java method, and do not use common symbols. After the AST is generated, the AST is encoded.

```
boolean f(Set<String> set, String value) {
    for (String entry : set) {
        if (entry.equalsIgnoreCase(value)) {
            return true;
        }
    }
    return false;
}
```

Figure 3: A simple example of a java method.

3.3. Attention. The attention mechanism has an important effect on sequence learning tasks. In our framework of the Code2tree, we introduced the attention mechanism to the encoder and decoder. The source data sequence is weighted and transformed.

When the input sequence is very long, it is difficult for the model to learn a reasonable vector to represent the sequence input. As the sequences continue to grow, the original performance based on the time step is getting worse and worse. All contextual input information is limited to a fixed length, and the capabilities of the entire model are also limited.

The attention mechanism retains the intermediate output results of the GRU encoder on the input sequence, and then trains a model to selectively learn these inputs and associate the output sequence with it when the model outputs. The weighted change of the target data can effectively improve the system performance in the natural way of sequence to sequence.

Our goal is to help the decoder introduce different word weights during word generation. In the training stage, the information in the decoder is defined as a query. The encoder contains all possible words, which we regard as a dictionary, and the key of the dictionary is the sequence information of all encoders.

In this paper, the attention mechanism selects important parts for each target word from the input sequence, and dynamically selects the distribution represented by these k combinations during decoding. The weight of the attention mechanism is calculated as formula

$$a_{ij} = \frac{\exp\left(\text{score}\left(h_j, h_{i-1}^*\right)\right)}{\sum_{j=1}^{n}\exp\left(\text{score}\left(h_j, h_{i-1}^*\right)\right)}, \tag{5}$$

where n is the number of tokens, h_j is the $j$thhidden state in the encoder, h_{i-1}^* is the $i-1$thhidden state in the decoder, and the score function is to score the matching degree between the hidden state h_{i-1}^* of the decoder and the hidden state h_j of the encoder.

3.4. Decoder. The decoder does not use a fixed context vector, but combines the attention information collected from the encoder. Context vector c_i is defined for predicting each target word y_i. Context vector c_i is the weighted sum of all hidden states A in the encoder and calculated as formula:

$$c_i = \sum_{j=1}^{m} \alpha_{ij} h_j. \tag{6}$$

Then, the decoder generates the target sequence y by sequentially predicting the probability of the word y_i conditioned on the context vector c_i and the previously generated word $y_1, \cdots, y_{i-1}$.

$$p\left(y_i | y_1, \cdots, y_{i-1}, x\right) = g\left(y_{i-1}, h_i, c_i\right), \tag{7}$$

where g is used to estimate the probability of the word y_i

4. Experiments

4.1. Datasets. The experiment uses the Java-small dataset from GitHub, which contains 11 relatively large Java projects. In the past, the dataset was not divided according to the project, which would cause almost the same code of the same project to appear in the training set and the test set at the same time, which made the model severely overfit the training data, making the BLEU score falsely high. By dividing the data set by project, we perform cross-project

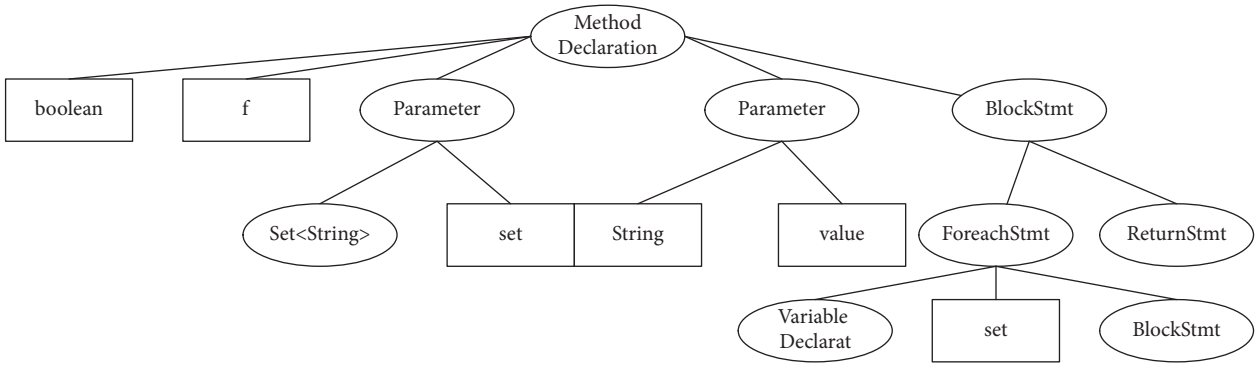

Figure 4: The abstract syntax tree of the java method is in Figure 3.

training and prediction tasks on different projects to obtain more realistic prediction results. The dataset is divided into three groups. It contains eleven top-level java projects. Nine projects are used as training projects, one project is used as a verification project, and one project is used as a test project. In total, the dataset contains approximately 700 K examples (Table 1).

4.2. Evaluation. In order to evaluate the quality of source code comment generation, this paper uses automatic evaluation indicators BLEU and *F*1-Score to evaluate the annotation generation quality. Among them, BLEU (bilingual evaluation understudy) is a machine translation automatic evaluation method based on N-gram proposed by IBM in 2002 [13]. The score is calculated as follows:

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log P_n\right), \tag{8}$$

where p_n is the ratio of candidate length n subsequences. In this paper, we use the default BLEU-4. BP is the concise penalty.

$$BP = \begin{cases} 1, & \text{if} c > r, \\ e^{1-r/c}, & \text{if} c \le r, \end{cases} \tag{9}$$

where c is the candidate translation length, and r is the effective reference sequence length.

The calculation formula of *F*1-Score is as follows:

$$F_1 = \frac{2 * P * R}{P + R}. \tag{10}$$

Precision represents the proportion of examples classified as positive examples that are actually positive examples. The calculation formula of precision is as follows:

$$P = \frac{TP}{TP + FP}, \tag{11}$$

where TP is to predict the positive class as the number of positive classes, FP is to predict the negative class as positive class number false positive.

The recall is a measure of coverage. The measure has multiple positive cases and is classified as positive cases. The formula for calculating the Recall rate is as follows:

Table 1: Java-small detailed information.

	Java-small
Training—projects	10
Validation—projects	1
Test—projects	1
Training—examples	665115
Validation—examples	23505
Test—examples	56165

$$R = \frac{TP}{TP + FN}, \tag{12}$$

where TP is to predict the positive class as the number of positive classes, FN is to predict the positive class as a negative class number.

In this paper, these indicators are introduced to evaluate the similarity between automatically generated annotations and manual annotations and to measure the quality of the annotation generation of the model.

4.3. Experimental Design. Code2tree is based on the TensorFlow framework. For each GRU that encodes the AST path, there are 256 units, and the decoder GRU has 640 units. During the training process, we used Adam to optimize the model. Regarding the hyperparameters of the model, we optimized the cross-entropy loss with the Nesterov momentum of 0.95. The learning rate is set to 0.01, each epoch attenuation is 0.05, and the embedding size is 128. In the choice of k value, we tried different k values, that is, the number of sampling paths for each Java method example. After continuous experiments, we set the k value to 200. We used the above hyperparameters to train the model for 1000 epochs. If we did not find any improvement after multiple epochs, we will manually stop the model training.

4.4. Results. In this section, we evaluate different methods by measuring the accuracy of the generated Java comments. Specifically, we mainly focus on the following research questions.

RQ1: compared with state-of-the-art baseline methods, how does Code2tree perform?

We compare Code2tree with Code2vec [11], which is an advanced code summarization method based on deep learning. Code2vec presents code fragments as fixed continuously distributed vectors to generate code comments.

We also compare the Code2tree method with the Seq2Seq model and the attention-based Seq2Seq model. The Seq2Seq model and the attention-based Seq2Seq model take source code as input. The purpose is to evaluate the effectiveness of neural machine translation (NMT) methods in comment generation.

In this paper, nondeep learning-based methods are not selected. Compared with the deep learning-based methods, the evaluation indicators that automatically generate annotations cannot be unified.

We evaluate the model through the popular evaluation indicators of machine translation, BLEU, and *F*1-Score. As shown in Table 2, compared with Code2vec, the improvement of Code2tree we proposed is great, and the scores of machine translation indicators are both ahead of the Code2vec model.

Compared with Code2vec, the *F*1-Score score of Code2tree improves by about 17%, and the BLEU-4 score of Code2tree improves by about 15%. The experimental results show that structured information is of great significance for translating text in structured language into unstructured language.

RQ2: how about the quality of comments by Code2tree for Java methods with different lengths?

We further analyzed the accuracy of Java methods and annotation predictions of different lengths. The performance of each model changes with the growth of the scale of the test method. All models have better annotation effects on short code fragments. As the input code size increases, the performance of all models shows different declines.

As shown in Figure 5, this paper chooses a Java method with a code length between 1–30 lines to verify the *F*1 score of the model in order to obtain the intuitive performance of the code length on the model performance. We compared Code2tree and Code2vec [11] and computed *F*1 scores based on different code lengths. From the figure, the *F*1 scores of the two methods of Code2tree and Code2vec will decrease as the code length increases. However, in the case of the same code length, the performance of Code2tree is always better than Code2vec.

The generated code comments can be described through the scores of code sequences of different lengths, which can well indicate that the code comments score tends to be stable with the increase of code sequence length. The scores will remain almost unchanged when the code length is more than 15.

RQ3: compared with the state-of-the-art baseline method, how about the quality of comments generated by Code2tree?

This paper considers the structure information and sequence information of the source code to better generate code comments. Table 3 and Table 4 compare the quality of comments generated by Code2tree and Code2vec [11] based on the same Java method example. It can be concluded from Table 3 that when the code length of the Java method is relatively short, the comments generated by the two methods are not much different. It can be concluded from Table 4 that when the code length of the Java method is relatively long, the comments generated by Code2vec will lose some key information in the Java method, and the quality is relatively poor. The comments generated by Code2tree can describe the Java method well.

Table 2: Comparison of scores of different models.

Model	*F*1	BLEU-4
Code2vec	26.19	19.45
Seq2Seq	30.90	20.45
Seq2Seq + attention	35.25	23.78
Code2tree	43.02	34.82

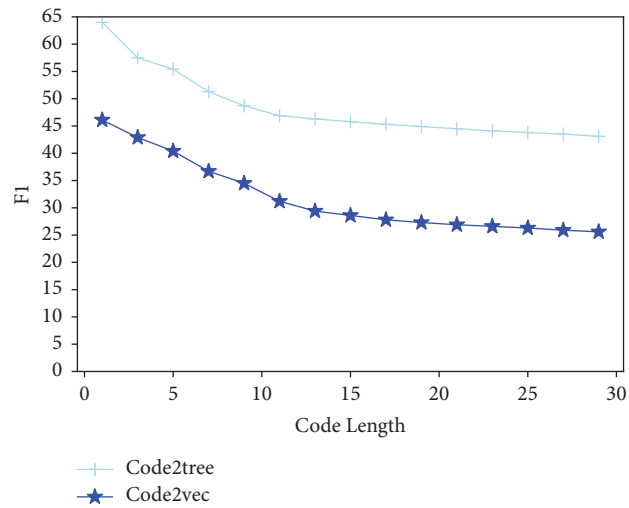

Figure 5: Comparison between Code2tree and Code2vec.

Table 3: An example of comments generated by Code2vec and Code2tree.

Java method	Public boolean is empty (){return name = = null; }
Code2vec comment	Returns true if the tree is.
Code2tree comment	Returns true if the symbol is empty.

4.5. Threats to Validity. In our research and experiment, there are some threats to validity as follows:

Firstly, there may be some deviation based on other dataset. We used the Java-small dataset for model training. Although the dataset is an open source data set and contains approximately 700 K examples, we cannot conclude that similar experimental results will be obtained on other data sets. In the future, we will do more experiments on other datasets.

Secondly, our experiments are only based on Java programs. We did not experiment with other programming languages. Although our method Code2tree has proved our technique's effectiveness on Java, we still need to verify its

Table 4: An example of comments generated by Code2vec and Code2tree.

Java method	Public static boolean instanceofAny (object o, collection<class> classes {for(class c: classes) {if(c.isInstance(o)) return true; } return false; }
Code2vec comment	Return ture if the symbol instanceofAny.
Code2tree comment	Return true if the object is registered otherwise return false.

validity based on other programming languages because of different program features.

Thirdly, we only used machine translation indicators BLEU and *F*1-Score to evaluate the gap between automatically generated comments and manually written comments. These indicators are widely used in machine translation problems, which can reduce the subjective influence of manual evaluation [13]. In the future, we will introduce other indicators to comprehensively evaluate the effectiveness of our technique.

5. Discussion

In the first experiment, we compared our technique with the Code2vec [11] model. During the data preprocessing, we extracted the variable names from AST. We computed scores of BLEU and *F*1-Score and our technique performs better than Code2vec. In our model, we use more specific words to replace common words in Java methods to make comments more accurate.

By analyzing the generated comments and source code, the performance of Code2tree is better than that of manually written comments. However, Code2tree is not good at learning methods or identifying names in comments. Developers have defined various names during programming, and these names usually appear rarely in the comments. During the training process, we replace all unknown identifiers in the AST sequence with their types, and these identifiers are represented by tags<UNK>.

In addition, for the Java-small data set we use, we can deal with some Java methods that have too short comments or too long lines of code, which is beneficial to the comments generated by Code2tree. Because the comment is too short or the number of lines of code is too long, it will cause the absence of Java comments. We compressed the AST sequences and converted each Java method into a fixed-length vector.

In future work, we will extend the Code2tree to other programming languages (such as python). In the process of dataset construction and preprocessing, we will adopt more advanced ways to build the common model.

6. Related Work

The research work on automatic code comment generation can be traced back to the research work of Haiduc et al. [20]. They first used an information retrieval technique to try to automatically generate text summaries for the code. In the early research phase of the problem, researchers focused more on template-based generation. The generation method based on information retrieval use heuristic rules to extract key information from the code and synthesize comments based on natural language description. With the rapid development of deep learning techniques, deep learning-based methods have effectively improved the quality of automatically generating code comments, and become the main research method for this problem. Among them, based on the sequence-to-sequence model in deep learning, Iyer et al. [8] proposed the CODE-NN method. Ahamd et al. [21] used the transformer model to generate code comments. The transformer model is a sequence-to-sequence model based on multihead self-attention, which can effectively capture long-range dependencies.

In this paper, we combined the deep learning method and the structural characteristics of the source code to generate code comments. Code2tree presents the process of code comments from the perspective of machine translation.

7. Conclusion

The automatic generation of source code comments can assist developers in better understanding the code and improving the efficiency of software development and maintenance. This paper proposes a Seq2Seq-based method Code2tree, which can generate high-quality annotations for Java methods. Code2tree converts the AST sequence into a vector as input, which can better maintain the structural information of the source code. This technique is superior to the state-of-the-art methods of code comments. It can achieve better performance based on common-used indicators BLEU and *F*1-Score.

In the future, we will apply our proposed method to other software engineering tasks and improve our method in practice.

Data Availability

The data used in this paper are all from public datasets, which can be found on GitHub.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported in part by the College Students' Innovation and Entrepreneurship Training Program under Grant 202110304086Y, in part by the Nantong Science and Technique Research Project under Grant JC2021125.

References

[1] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: a large-scale field study

with professionals," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 951–976, 2018.

[2] G. Sridhara, E. Hill, and D. Muppaneni, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM International Conference on Automated software engineering*, pp. 43–52, Antwerp, Belgium, September 2010.

[3] Y. Bai, L. Zhang, and S. Yan, "Automatic generation of code comments based on comment reuse and program parsing," in *Proceedings of the 2019 2nd International Conference on Safety Produce Informatization (IICSPI)*, pp. 380–388, Chongqing, China, November 2019.

[4] X. Song, H. Sun, X. Wang, and J. Yan, "A survey of automatic generation of source code comments: algorithms and techniques," *IEEE Access*, vol. 7, pp. 111411–111428, 2019.

[5] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2014, https://arxiv.org/abs/1409.0473.

[6] J. Yin, X. Jiang, and Z. Lu, "Neural generative question answering," 2015, https://arxiv.org/abs/1512.01337.

[7] C. Chelba, D. Bikel, and M. Shugrina, "Large scale language modeling in automatic speech recognition," 2012, https://arxiv.org/abs/1210.8440.

[8] S. Iyer, I. Konstas, and A. Cheung, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pp. 2073–2083, Berlin, Germany, August 2016.

[9] X. Hu, G. Li, and X. Xia, "Deep code comment generation," in *Proceedings of the 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pp. 200–210, IEEE, Gothenburg, Sweden, May 2018.

[10] Y. Huang, S. Huang, H. Chen et al., "Towards automatically generating block comments for code snippets," *Information and Software Technology*, vol. 127, no. 3, Article ID 106373, 2020.

[11] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," in *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, Beijing China, August 2019.

[12] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[13] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proceedings of the Advances in neural information processing systems*, pp. 3104–3112, Montreal, Quebec, December 2014.

[14] P. F. Brown, V. J. Della Pietra, and P. V. Desouza, "Class-based n-gram models of natural language," *Computational Linguistics*, vol. 18, no. 4, pp. 467–480, 1992.

[15] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proceedings of the International Conference On Machine Learning*, pp. 2091–2100, New York, NY, USA, June 2016.

[16] P. Loyola, E. Marrese-Taylor, and Y. Matsuo, "A neural architecture for generating natural language descriptions from source code changes," 2017, https://arxiv.org/abs/1704.04856.

[17] P. Bielik, V. Raychev, and M. Vechev, "Program synthesis for character level language modeling," in *Proceedings of the International Conference on Learning Representations*, pp. 1–17, Toulon, France, April 2017.

[18] Y. Liang and K. Zhu, "Automatic generation of text descriptive comments for code blocks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 5229–5236, Orleans, LA, USA, February 2018.

[19] M. Chen and X. Wan, "Neural comment generation for source code with auxiliary code classification task," in *Proceedings of the 2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 522–529, IEEE, Putrajaya, Malaysia, December 2019.

[20] S. Haiduc, J. Aponte, and L. Moreno, "On the use of automated text summarization techniques for summarizing source code," in *Proceedings of the 2010 17th Working Conference on Reverse Engineering*, pp. 35–44, IEEE, Washington, DC, USA, October 2010.

[21] W. U. Ahmad, S. Chakraborty, and B. Ray, "A transformer-based approach for source code summarization," 2020, https://arxiv.org/abs/2005.00653.