Source Code Representation Approach Based on Multi-Head Attention

Lei Xiao[1,*], Hao Zhong[1], Yiliang Lai[2], Xinyang Lin[3], and Hanghai Shi[1]
[1]Xiamen University of Technology, Xiamen, Fujian, China
[2]Yilang (Xiamen) Technology Co. Ltd, Xiamen, Fujian, China
[3]Xiamen Zhonglian Century Co. Ltd, Xiamen, Fujian, China
lxiao@xmut.edu.cn, 849826920@qq.com, 1030184381@qq.com, linxy@zhonglian.com, shh@xmut.edu.cn
*corresponding author

Abstract—Code classification and code clone are one of the current research hotspots. How to represent source code characteristics is a key link in code classification and code clone. At present, with the rapid development of deep learning, compared with traditional source code feature representation methods, deep learning methods can better represent source code features. However, there are still some shortcomings in the existing deep learning methods. Some models have high complexity and lack of parallel ability. To solve this problem, this paper proposes a source code representation method based on multi head attention mechanism (SCRMHA). SCRMHA converts the source code snippet into statement vector and uses the multi-head attention mechanism to capture the representations of the entire code snippet. We apply the obtained source code snippet characteristics to code classification and code clone. Experimental results show that the performance of SCRMHA is better than that of traditional source code feature representation methods. SCRMHA takes less time than ASTNN and its complexity is about 1/3 of ASTNN, which can effectively represent the characteristics of source code.

Keywords–multi-head attention; code clone; code classification; source code representation; abstract syntax tree

1. Introduction

Code classification can help us to better manage and maintain code. Through profound study of code classification, we can better understand the structure and function of code. Code clone can help software companies improve the efficiency of software development to a certain extent, but if the cloned code itself has vulnerabilities, then the cloned code with vulnerabilities will lay hidden dangers for the system. At present, code classification and code clone have aroused extensive research. One of the key links is how to represent the source code features.

The traditional methods of extracting source code features are usually text-based, token sequence based, abstract syntax tree based, program dependence graph (PDG) based and control flow graph (CFG) based. For example, Wang et al. [14] use a token-based approach to obtain the feature representation of the code. The traditional methods of obtaining code features can only extract and compare the underlying features, but can not dig out the rich information behind its structure. In recent years, the use of neural networks for obtaining code representations has become increasingly popular. For example, CDLH [15] uses Tree-LSTM to obtain the features of code snippets. Wang et al. [13] proposed a fast and accurate semantic tagging tool called CCStokener, which extracts the types of relevant nodes in the AST path of each token, converts these types into fixed dimensional vectors, and then models their semantic information by applying n-grams on their related tokens. Ehsan et al. [4] developed a clone ranking model using machine learning to help developers identify the clones with the highest risk as early as possible. Guo et al. [5] proposed a method that can use the similarity of tokens and the architecture of abstract syntax trees to detect code clones. The structure of the syntax tree maintains the accuracy of detecting clone pairs while also maintaining the speed of matching code similarity. AST can well represent the deep characteristics of source code, and it can be used for code snippets that can not be compiled completely to comparative analysis, so it has caused extensive research and achieved many good results. Existing networks for feature extraction on AST trees still face various problems, such as the high complexity and the lack of parallel processing ability. To solve this problem, this paper proposes a source code representation method based on multi-head attention mechanism(SCRMHA).

The main contribution of this paper is the modification of the ASTNN model to use a multi-head attention mechanism to capture the features of the entire code snippet. Using multi-head attention mechanisms allows model to extract source code features in parallel to a higher degree, which can significantly improve training and reasoning speed. Specifically, the process of extracting features from source code snippets using the SCRMHA model is as follows: SCRMHA first converts the source code snippet into an abstract syntax tree (AST), then cuts the AST into small sequence of statement trees and encodes the sequence of statement trees into statement vectors using a statement encoder, then uses a multi-head attention mechanism to capture the features of the entire code snippet, and finally uses two public datasets. The obtained code snippet features are applied to code classification and code clone. Experimental results show that SCRMHA performs better than traditional source code representation. SCRMHA takes less time than ASTNN and is 1/3 as complex as ASTNN, and can capture source code features effectively.

2. Background

In this section, we will introduce the techniques used by SCRMHA such as AST, multi-head attention mechanism, and related work.

2.1 Abstract Syntax Tree

AST is a key data structure in programming language processing, used to represent the abstract syntax structure of source code [1]. AST can represent the hierarchical structure and syntax relationships of code, providing a strong foundation for compilers, interpreters, and various code analysis tools. AST plays an important role in source code analysis, optimization, and transformation. Figure 1 shows a source code snippet (a) from the BigCloneBench [9] (BCB) dataset and an abstract syntax tree (b) for that source code snippet. As can be seen from the Figure 1, abstract syntax tree does not include specific implementation details of source code snippets, but only contains syntax structure information and semantic information of source code.

2.2 Multi-head Attention Mechanism

The multi-head attention mechanism [11] comes from the Transformer architecture, which processes sequential data by computing attention in parallel across different subspaces. This mechanism allows the model to focus on multiple locations in the sequence at the same time, capturing complex dependencies. Multi-head attention mechanism is a variant of self-attention mechanism that can be described as a mapping between the Query matrix (Q), Key matrix (K), Value matrix (V), and output matrix, which represents the input data X as Q, K, V, as shown in formula (1). Where $Q \in R^{n\times d_k}, K \in R^{m\times d_k}, V \in R^{m\times d_v}, d_k$ represents the number of hidden layers of the neural network, $\sqrt{d_k}$ is used to avoid the excessive dimension of Q and K inner product, and plays a regulating role. In this way, correlations within the data are captured. The multi-head attention mechanism can capture information about different subspaces of the same sequence and compute from different locations of the data, because it can generate multiple heads of different subspaces for computation depending on the degree of mapping between Q, K, V and the output vector. This feature allows the model to focus on and capture characteristics of different aspects of the data at the same time. Each head fuses the extracted features, as shown in formula (2), thereby enhancing the weight of the features and making the features extracted by the model more comprehensive and effective.

$$head_i = \mathrm{Attention}(Q_i, K_i, V_i) = \mathrm{softmax}(\frac{Q_i K_i^{\mathrm{T}}}{\sqrt{d_k}})V_i \quad (1)$$

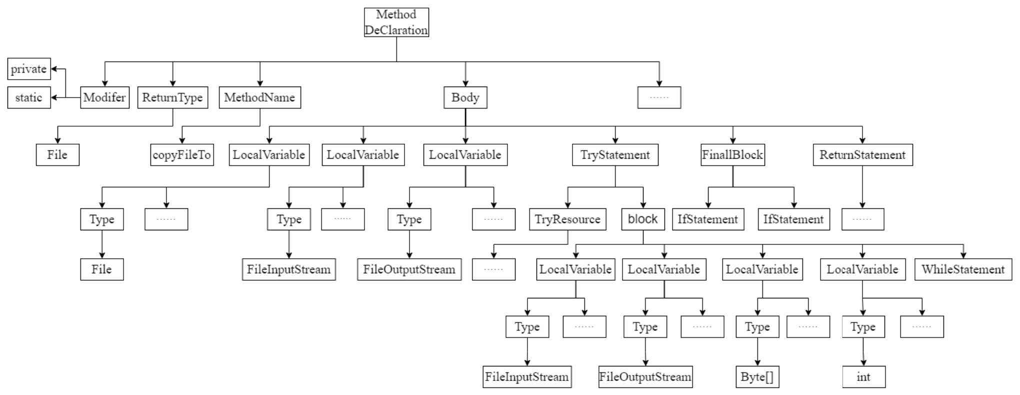
$$\mathrm{S} = \mathrm{Concat}(\mathrm{head}_1, \mathrm{head}_2, \ldots, \mathrm{head}_n)\mathrm{W}_{hr} \quad (2)$$

2.3 Related work

The rise of deep learning has revolutionized many fields. Deep learning methods can automatically learn lexical, syntactic, and semantic information in code, thereby obtaining a feature representation of the source code. Among the current source code feature extraction methods combined with deep learning, AST has caused extensive research and achieved many good results due to its ability to represent the deep features of the code well and its ability to compare and analyze the code using code snippets that cannot be compiled completely. For example, Zhang et al. [18] proposed a model named ASTNN for extracting code features. ASTNN decomposes each large AST into a series of small statement trees, and encodes the statement trees into vectors by capturing the lexical and syntactic knowledge of the statements. A bidirectional GRU model is used to generate a vector representation of the code snippet. The method has a high accuracy of 98.2% for code classification and is sufficient to detect all types of code clones. Zhang et al. [18] combined linear support vector machine with traditional infrared methods to compare the effectiveness of astnn. Extract text features using TF-IDF, N-gram, and LDA. For TF-IDF, they use tokens extracted from the source code file as the corpus. For N-grams, they set the value of grams to 2 and the maximum number of features to 20000. For LDA, they set the number of topics to 300. In addition, they also used deep learning methods to compare with astnn, using TextCNN and LSTM. They processed code fragments as pure text to

```
private static File copyFileTo(File file, File
directory)
throws IOException {
File newFile = new File(directory,
file.getName());
FileInputStream fis = null;
FileOutputStream fos = null;
try {
fis = new FileInputStream(file);
fos = new FileOutputStream(newFile);
byte buff[] = new byte[1024];
int val;
while ((val = fis.read(buff)) > 0)
fos.write(buff, 0, val);
} finally {
if (fis != null) fis.close();
if (fos != null) fos.close();
}
return newFile;
}
```

(a) Code Snippet



(b) Abstract syntax tree

Figure 1. Code snippet and abstract syntax tree

adapt them to the task of token sequences. For TextCNN, they set the kernel size to 3, the number of filters to 100, and for LSTM, they set the dimension of hidden states to 100. Chochlov et al [3] used the CodeBERT deep neural network to embed each code snippet into a fixed-length feature vector to detect clones of type III and type IV. Their results show a shorter execution time compared to other methods. Mou et al. [8] proposed a tree based convolutional neural network (TBCNN), where the ast based convolutional layer is the core of TBCNN. It applies a fixed depth feature detector by sliding the entire ast. TBCNN adopts a bottom-up encoding layer to integrate some global information, improving its locality. Xue et al. [16] designed a new semantic graph based deep detection method called SEED. For a pair of code snippets, SEED constructs a semantic graph of each code snippet based on intermediate representations to more accurately represent code semantics compared to representations based on lexical and syntactic analysis. Hu et al. [6] proposed a tree based scalable code cloning detector, TreeCen, which satisfies scalability while effectively detecting semantic clones. Wang et al. [12] proposed a UAST neural network. Using self attention combined with bidirectional LSTM to extract code AST sequence features to capture the global logical structure features of the code, and using Graph Convolutional Network (GCN) to extract the graph structure features of the AST. After fusing these two features, the structural and semantic features of the code are obtained. In order to reduce the differences between different programming languages, a unified vocabulary is used to eliminate the differences between different language ASTs when generating ASTs. Finally, connecting the two feature vectors together is the code vector. A neural network trained using this vector for classification can be used to detect cross language code clones. Zhang et al. [17] proposed a cross language code plagiarism detection method based on program flowchart and graph attention network. Firstly, convert the code into a program flowchart and use a graph attention network to extract the features of the program flowchart as a representation of the code. Secondly, the cross matching method is used to compare the representation of the code line by line, in order to obtain similar feature vectors of the code. Finally, the similar feature vectors of the reception detection code are combined, and the probability of plagiarism is calculated using a fully connected neural network.

Traditional methods can only detect lexical similarity and relatively low syntactic similarity, while for stronger syntactic similarity or semantic similarity, traditional code feature representation methods are more difficult to solve. Moreover, there are problems of high model complexity and lack of parallel processing ability in deep learning based methods, therefore we propose a source code representation method based on multi-head attention mechanism.

3. Our approach

This section describes the approach taken in this paper. The structure of SCRMHA model is shown in Figure 2. First, with the idea of ASTNN splitting AST, SCRMHA takes the function snippet of source code as the granularity, converts the function snippet of source code into an abstract syntax tree through an abstract syntax tree parser, then divides AST into small statement tree sequences through a traverser and a constructor, and then uses a sentence encoder to capture the vocabulary and syntax information at the statement level and express the statement tree sequence as statement vectors. Finally, deep-level features are extracted using the multi-head attention mechanism and vector features of the entire code snippet are obtained through the pooling layer.

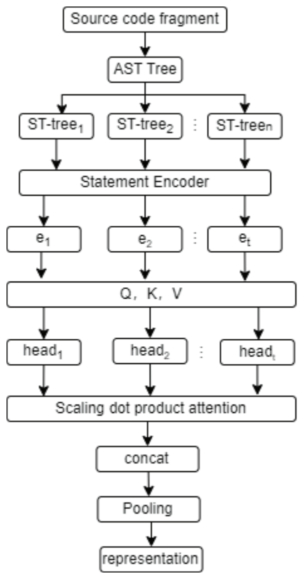


Figure 2. SCRMHA model diagram

3.1 Generating an AST tree and cutting and coding the AST tree into a sequence of statement trees

Taking the function snippets of the source code as the granularity, the function snippets are converted into AST by using an AST parser. However, the converted AST suffer from the problem of partially oversizing, which tends to lead to the disappearance of the gradient. Therefore, it is necessary to cut and encode the converted AST into a small sequence of statement trees. Consider Method Declaration as a special statement node. First, the statement node S of AST is extracted by prior order traversal, so we have $S' = S \cup \{Method\ Declaration\}$, and for nested statements such as $While$, $DoWhile$, Try, For, If, $Switch$, $FuncDef$, we define the independent node $P = \{block,body\}$. Block is used to slice the header and body of nested statements and body is used for method declarations. For all statement nodes s ($s \in S'$), the child nodes are defined as $D(s)$. For any $d \in D(s)$, if there exists a path between nodes s and d through node p ($p \in P$), indicating that a statement in node s contains node d, then node d is called a substatement node of node s. Thus, s and its descendant node $D(s)$ form a statement tree with s ($s \in S$) as root. Since a statement tree node may have three or more child nodes, for such a statement tree node is called a multiplexed statement tree. In this way, an AST can be decomposed into a series of non-overlapping multiplexed statement trees. The statement tree is constructed recursively using traversers and constructors and added to the statement tree sequence. The statement tree sequence order implies hierarchical information about the code structure. This yields the statement tree sequence as the raw input to SCRMHA.

For the resulting statement tree sequence, the encoder is used to encode the statement tree sequence into statement tree vectors. The syntax symbols obtained by the first order traversal are used as the training corpus, and Word2Vec is used to train the symbol embedding vector, thus the initial parameters of the sentence encoder are obtained. Then traversing the statement tree recursively computes the symbol of the current node as the new input, and uses a dynamic batch algorithm [18] to calculate the hidden state of its leaf nodes. Specifically, for the given statement tree t, let C be the number of child nodes, n the number of non-leaf nodes, and use the pre-trained embedding parameter W_e ($W_e \in \mathbb{R}^{|V| \times d}$), V is the vocabulary and d is the embedding dimension, then the lexical vector of node n is shown in formula (3), where W_e is the pre-trained word embedding matrix. x_n is the one-hot vector representation of symbol n and v_n is the embedding. The vector representation of node n is shown in formula (4), where $W_n \in \mathbb{R}^{d \times k}$ is a weight matrix with encoding dimension k, b_n is a bias term, h_i is the hidden state of the child node i, h is the updated hidden state, and, σ is the activation function. Finally, the vector representing et for each statement tree is obtained by using maximum pooling, as shown in formula (5), where N is the number of nodes in the statement tree.

$$v_n = {W_e}^{\top} x_n \tag{3}$$

$$h = \sigma({W_n}^{\top} v_n + \sum_{i \in [1,C]} h_i + b_n) \tag{4}$$

$$e_t = [\max h_{i1}, maxh_{i2}, \cdots, maxh_{ik}], i = 1, 2, \cdots, N \tag{5}$$

3.2 Obtaining a vector representation of a code snippet

The obtained statement tree vector is used as the input of the multi-head attention mechanism, and the heads of different subspaces are calculated using formula (1). Then, the extracted features of each head pair are fused using formula (2) to enhance the weight of the features and make the features extracted by the model more comprehensive and effective. Finally, the most important semantics are captured by sampling by maximum pooling. Finally, a vector $r \in \mathbb{R}^{2m}$ is generated as a vector representation of the code snippet.

4. Application of SCRMHA

The vector representation of code snippets obtained by model can be applied to multiple tasks. In order to verify the validity of the model, the model is applied to code classification and code clone detection, to evaluate the validity of the model and calculate the complexity of the model.

Table 1. Code clone types

Original Code	Code snippet 1	Code snippet 2	Code snippet 3	Code snippet 4
			if(a==b)	
	if(a==b)	if(g==f)	{	switch(true)
if(a=b)	{	{	//comment1	{
{	//comment1	//comment1	c=a*b;	//comment1
c=a*b;	c=a*b;	h=g*f;	//new	case a ==b:
}	}	}	b=a-c;	c=a*b;
else{	else{	else{	}	//comment2
c=a/b;	//comment2	//comment2	else{	case a!=b;
}	c=a/b;	c=g/f;	//comment2	c=a/b;
	}	}	c=a/b;	}
			}	
	Type-1	Type-2	Type-3	Type-4

4.1 Code classification

Code classification is a task to classify code snippets according to their functions in order to facilitate the understanding and maintenance of programs [8]. In the task of code classification, first, given the number of categories M, the source code snippet is sent into the SCRMHA model to obtain the vector representation t of the source code snippet, and then logits are used as shown in formula (6), where W_o is the weight matrix ($W_o \in \mathbb{R}^{2m \times M}$) and b_o is the offset term. This paper uses the cross entropy loss function as the loss function of the classification task. In the formula, Θ is the ownership heavy matrix parameter in the model, and y is the true class of the function snippet. Code classification is a multi classification task and the predicted value can be obtained by formula (8).

$$\hat{x} = W_o r + b_o \tag{6}$$

$$J(\Theta, \hat{x}, y) = \sum \left(-log \frac{exp(\hat{x}_y)}{\sum_j exp(\hat{x}_j)} \right) \tag{7}$$

$$prediction = \arg max(p_i), i = 1, \cdots, M \tag{8}$$

4.2 Code clone detection

The code clone detection task can help us identify the copied code snippets in the program, when the copied code snippets have security defects, it is conducive to detect the copied code snippets, and achieve autonomous control. Bellon et al. [2] classified code clone into four types:

- **Type-1:** Code pairs in which two code snippets are identical except for spaces and comments.
- **Type-2:** Code fragments with identical syntactic structure, differing only in variable names, numbers, types, characters, spaces, layout and comments.
- **Type-3:** Cases where there are several statements or expressions with additions, deletions, etc., or pairs of code that use different identifiers, text, types, spaces, layouts, and comments, but still have similar syntax.
- **Type-4:** Heterogeneous code for the same function is not textually or syntactically similar, but semantically similar.

As shown in Table 1, given a simple source code, **Type-1** clone example has two more comment statements than the original code, **Type-2** example changes the variable name on this basis, and **Type-3** Type adds the code statement "b= A-C;" without changing the semantics. Finally, **Type-4** clones are syntactically modified, changing the original if statement structure to the $switch$ structure.

In code clone detection, source code snippets are converted into two vector fragments r_1, r_2 by SCRMHA. In this paper, the distance between r_1, r_2 is calculated by formula (9) and r is represented as semantic relevance [10].In this paper, the sigmoid function is used to obtain the output $\hat{y}, \hat{y}_i \in [0,1]$, as a measure of similarity, as shown in formula (10) and formula (11) , where $W_{\mathbf{o}} \in \mathbb{R}^{2m \times M}$ is the weight matrix and b_0 is the bias term. In this paper, the loss function is defined as a binary cross entropy loss function, as shown in formula (12). In order to improve the computational efficiency of the model and minimize the loss of the model, the optimizer AdaMax [7] is used in this paper to complete the calculation.

$$r = |r_1 - r_2| \tag{9}$$

$$\widehat{x} = W_0 r + b_0 \tag{10}$$

$$\widehat{y} = \text{sigmoid}(\widehat{x}) \tag{11}$$

$$J(\Theta, \hat{y}, y) = \sum(-(y \cdot log(\hat{y}) + (1-y) \cdot log(1-\hat{y}))) \tag{12}$$

When all the parameters are optimized, the model is stored. We load the trained model to make predictions about the statement tree sequence of the new code snippet. Code clone detection is a binary task. P is a number in the range [0,1]. We can get the predicted value by formula (13), where δ is the threshold, we set δ to 0.5. When p is greater than 0.5, the code pair is judged to be cloned, otherwise the code pair is not cloned.

$$\text{prediction} = \begin{cases} \text{True}, & p > \delta \\ \text{False}, & p \leq \delta \end{cases} \tag{13}$$

4.3 Complexity calculation

The floating point operations (FLOPs) and the number of parameters (Params) of a deep learning model are key metrics for evaluating the performance and complexity of the model. These two metrics not only affect the training and inference speed of the model, but also reflect the size and learning capability of the model.

- **FLOPs:** FLOPs refer to the number of floating point operations required when the model performs reasoning or training tasks. This includes basic mathematical operations such as addition and multiplication. The amount of calculation directly affects the efficiency and speed of model calculation.
- **Params:** Params refers to the number of weights and biases in the model to be trained. These parameters learn to adapt to the input data during the training process, so that the model can make accurate predictions. The size of the parameter number is directly related to the capacity of the model, and a larger parameter number usually indicates that the model has a strong learning ability, but it may also lead to overfitting.

5. Experiment and result analysis

In this section the experimental results of the application of the model to code categorization and code cloning will be evaluated, comparing the experimental results with traditional methods. We will also compare the complexity and time consumed by the model with ASTNN.

5.1 Experiment Data

We use two common datasets for code classification and code clone detection. One of the two public datasets is 104 programming questions collected by Mou et al.[1] from an online judge system. The other is a dataset provided by Svajlenko et al. [9] for code clone detection. These two datasets are widely used by researchers, and these two common datasets can be applied to model evaluation.

5.2 Evaluation indicators

Code classification is a multi classification task. In multi classification problems, there may be significant differences in the number of samples from different categories. In this case, accuracy and recall may be affected by imbalanced sample sizes, leading to inaccurate evaluation results. In contrast, accuracy is a better choice as it considers the correct classification of all categories and is more suitable for measuring the overall performance of multi classification models. Therefore, for code classification, we only use test accuracy to calculate the percentage of correctly classified test sets. The calculation formula of Accuracy is shown in formula (14), where True Positive (TP) indicates that the prediction result is positive samples, and the actual samples are also positive samples, that is, the number of positive samples correctly identified. False Positive (FP) indicates that the predicted result is positive sample, but the actual sample is negative, that is, the number of negative samples is false positive. True Negative (TN) indicates that the predicted result is negative samples, but the actual samples are negative samples, that is, the number of negative samples is correctly identified. False Negative (FN) indicates that the predicted result is negative samples, but the actual samples are positive samples, that is, the number of positive samples missed.

$$Acc = \frac{TP + TN}{TP + FP + TN + FN} \tag{14}$$

Code clone can be seen as a binary task (clone or not). Precision, Recall, and F1 values are widely used in binary classification tasks because they provide the ability to evaluate different aspects of model performance on positive and negative examples, while balancing the accuracy and coverage of the model. We choose the commonly used Precision, Recall and F1-measure as evaluation indicators. The calculation formula of P is shown in formula (15). The calculation formula of R is shown in formula (16). The calculation formula of F1 is shown in formula (17).

$$P = \frac{\text{TP}}{\text{TP} + FP} \tag{15}$$

Table 2. Comparison of code classification test accuracy

	Method	Test Accuracy(%)
Traditional method	SVM+TF-IDF	79.4
	SVM+N-gram	84.7
	SVM+LDA	47.9
Deep learning method	TextCNN	88.7
	LSTM	88.0
	TBCNN	94.0
	SCRMHA	97.7

$$R = \frac{TP}{TP + FN} \tag{16}$$

$$F = \frac{2TP}{2TP + FP + FN} \tag{17}$$

5.3 Experimental environment

The software environment used in this experiment is Ubuntu 18.04.6 LTS and Python3.8. Based on the existing laboratory hardware equipment, the hardware environment used in this experiment is Intel(R) Xeon(R) Gold 6242R CPU (3.10GHz), 256GB running memory, 36TB hard disk, and 4 NVIDIA GeForce RTX 2080 Ti (11GB video random access memory).

5.4 Experimental results and analysis

In order to better evaluate and analyze the model, we put forward four questions:

RQ1: How well does our model work on code classification? Aiming at this problem, we use the OJ dataset to evaluate the effect of the model on code classification. We compare with traditional methods (SVM+TF-IDF, SVM+N-gram and SVM+LDA) and neural network methods (TextCNN, LSTM, TBCNN). As shown in Table 2. We can find that the effect of SCRMHA is 18%, 13% and 50% higher than that of SVM+TF-IDF, SVM+N-gram and SVM+LDA, and 9%, 10% and 4% higher than that of TextCNN, LSTM and TBCNN, respectively. This indicates that our method outperforms traditional methods and neural network methods. This is because traditional methods can only extract shallow semantic features of code to classify its functionality, while variable symbols used in OJ datasets are usually meaningless names such as i, j, k, a, etc. Therefore, these traditional methods cannot accurately extract code features and classify them. SCRMHA uses a multi head attention mechanism to capture the semantic relationships of code sequences, which is more flexible. SCRMHA can better understand and represent the syntax and semantic results of the code. At the same time, SCRMHA can focus on multiple positions in the input sequence, making it easier to capture global information of the code and better understand the entire code fragment. This makes SCRMHA more accurate in extracting code features and improves the accuracy of code classification. In neural network methods, although TextCNN can process text in parallel and improve computational efficiency, its receptive field is fixed and may not be able to capture long-distance dependencies. SCRMHA, which uses a multi head attention mechanism, can better handle long-distance dependencies because it takes into account the relationships of all positions in the sequence when calculating attention weights. Although LSTM can capture long-term dependencies in sequences, its information capture is unidirectional and can only look forward from the current position, without utilizing future information. SCRMHA, which uses a multi head attention mechanism, can simultaneously consider the pre - and post information in the sequence, and has a more comprehensive information capture ability. TBCNN captures sentence structure information by constructing a syntax tree, but its feature representation may be limited by the quality of the tree structure and the construction method. SCRMHA, which uses a multi head attention mechanism, does not rely on specific syntactic tree structures and can learn complex relationships within sequences more flexibly.

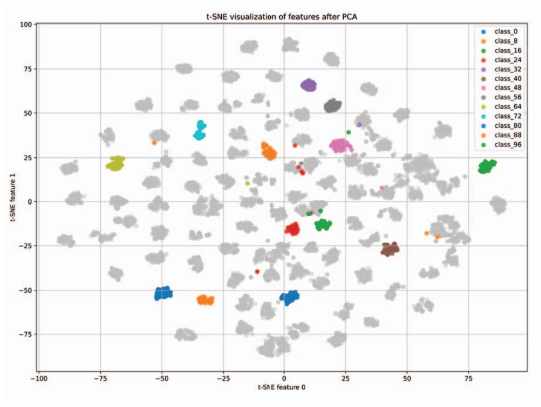


Figure 3. T-SNE diagram

The T-SNE diagram can intuitively see the quality of the feature extraction effect of the model. We use T-SNE to visualize the feature extraction and show the distribution of feature extraction. Since there are 104 categories in the code classification, we select 12 individual categories for visualization. As shown in Figure 3, the color of the points represents which category the sample belongs to. In the Figure 3, some points that do not gather together are called outliers, and the outliers represent relatively poor feature extraction of samples, often those points that are classified incorrectly. Fewer outliers means better feature extraction. The Figure 3 shows that this model can distinguish the samples of this category well and distinguish them from other categories, which indicates that the feature extraction effect of the model is good.

RQ2:How well does our model work on code clone? For this problem we use OJ and BCB dataset to assess model on code clone detection effect. We compare token-based methods (RAE+,Token-A [14]) with tree-based methods (CDLH). We divide code clones into four types, namely Type-1, Type-2,Type-3, and Type-4. In Tpye-3, we can divide it into Type-S3 and Type-M3 according to the strength of clone. A Type-All is obtained using a weighted sum based on the percentage of various clone types [15]. Aiming at this problem, we use the OJ and BCB dataset to assess model effect on code clone. Since the dataset we use is the same as the dataset in ASTNN,

Table 3. Comparison results of code clone

	Dataset	Type	P	R	F1
RAE+	OJ	/	52.5	68.3	59.4
	BCB	Type-1	100	100	100
		Type-2	86.5	97.2	91.5
		Type-S3	79.9	72.2	75.9
		Type-M3	66.4	74.8	70.3
		Type-4	76.3	58.7	66.3
		Type-All	76.4	59.1	66.6
CDLH	OJ	/	47	73	57
	BCB	Type-1	-	-	100
		Type-2	-	-	100
		Type-S3	-	-	94
		Type-M3	-	-	88
		Type-4	-	-	84
		Type-All	92	74	82
Token-A	BCB	Type-1	-	99	-
		Type-2	-	98	-
		Type-S3	-	88	-
		Type-M3	-	43	-
		Type-4	-	2.3	-
		Type-All	91	-	-
SCRMHA	OJ	/	97.6	91.3	94.4
	BCB	Type-1	100	100	100
		Type-2	100	100	100
		Type-S3	99.6	94.3	96.6
		Type-M3	99.6	91.4	95.4
		Type-4	98.8	88.6	93.4
		Type-All	98.9	88.7	93.5

and ASTNN has done comparisons with other models, we refer directly to the results of code clones of pairs in ASTNN. As can be seen from Table 3, our SCRMHA model is superior to RAE+, Token-A and CDLH. In OJ dataset, SCRMHA is 45%, 23% and 35% higher than RAE+ in accuracy, recall rate and F1 value, and 50%, 18% and 37% higher than CDLH, respectively. In BCB, SCRMHA is 22%, 30% and 27% higher than RAE+, and 7%, 15% and 11% higher than CDLH, respectively. In terms of clone Types 1-4, it can be seen from the Table 3 that SCRMHA is superior to Token-A in terms of accuracy and recall rate. This is because RAE+ and Token-A use a token-based approach, which is the smallest unit of code that typically contains only lexical information and ignores syntactic and semantic information. This causes token-based approaches to have limitations in understanding the structure and meaning of the code. Although CDLH uses Tree-LSTM, it is difficult to capture a lot of lexical information due to the use of a large number of i, j, k and other variables with no practical meaning in the OJ dataset, resulting in the loss of a lot of lexical information. In BCB datasets, SCRMHA uses multi-head attention to allow a higher degree of parallel processing to focus on multiple locations in the sequence at the same time, thus capturing complex dependencies and thus performing well with CDLH.

RQ3: How does the effect of our model compare to the effect of the ASTNN model?

Table 4. Code classification comparison results

	Test Accuracy(%)
ASTNN	98.2
SCRMHA	98.0

Table 5. Code Clone Comparison Results

	Dataset	Type	P	R	F1
ASTNN	OJ	/	98.9	92.7	95.5
	BCB	Type-1	100	100	100
		Type-2	100	100	100
		Type-S3	99.9	94.2	97.0
		Type-M3	99.9	91.5	95.3
		Type-4	99.6	88.4	93.7
		Type-All	99.6	88.6	93.8
SCRMHA	OJ	/	97.6	91.3	94.4
	BCB	Type-1	100	100	100
		Type-2	100	100	100
		Type-S3	99.6	94.3	96.6
		Type-M3	99.6	91.4	95.4
		Type-4	98.8	88.6	93.4
		Type-All	98.9	88.7	93.5

From Table 4, we can see that in terms of code classification, the acc of the modified model is slightly lower than that of the original model, and the prediction labels and actual labels of ASTNN and SCRMHA are shown in Figure 4. From Table 5, we can see that in terms of code clone detection, the P and F1 value of the modified model are slightly lower than that of the original model, which may be due to the following reasons:

- SCRMHA uses a multi-head attention mechanism, which, due to its complexity, may be more prone to overfitting, especially if relatively little training data may require a larger dataset to take full advantage of it, and performance may suffer if the data volume is small.
- Multi-head attention mechanisms may be more sensitive to the characteristics of the data when processing it. If the characteristics of the dataset do not match the expectations of the multi-head attention mechanism, such as focusing more on local information than global information, performance can be degraded. In the field of code, different pieces of code may have different dependency structures, which can affect the performance of multi-head attention mechanisms.
- Multi-head attention mechanisms and GRU (neural networks used in ASTNN) may have different advantages when dealing with different types of tasks. The recall rate is slightly higher than that of the original model, indicating that the model is slightly stronger than ASTNN for the true clone pairs of Type-S3 and Type-4.

RQ4: How does the complexity of our model compare to that of ASTNN?

We used Python's thop library to calculate FLOPs and Params the model.The complexity and time consumption of the two models, as shown in Table 6. The results show that the complexity of SCRMHA is about 1/3 that of ASTNN, and

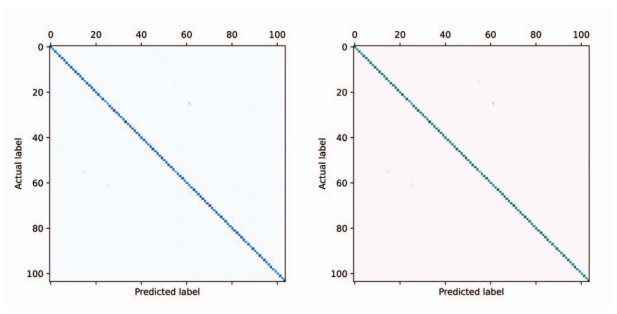


Figure 4. ASTNN (left) SCRMHA (right)

Table 6. Complexity comparison

	FLOPs(M)	Params(M)	Time Consuming (S)		
			Code Classification	Code Clone	
				OJ	BCB
ASTNN	234.15	0.18	5862	1087	5664
SCRMHA	77.64	0.05	5249	1059	5120

the time spent in code classification and cloning is also better than ASTNN under the premise of no obvious reduction in accuracy, presion, recall and F1. This shows that the proposed method can effectively reduce the complexity of the model on the premise of ensuring the accuracy. This may be because, relative to traditional RNN or GRU structures, multi-head attention mechanisms can share the parameters of the attention mechanism, so the number of parameters may be relatively reduced. This method of parameter sharing can reduce the complexity of the model, and the multi-head attention mechanism can compute multiple attention heads in parallel when processing sequence data, so the computational efficiency is improved to some extent. In contrast, loop structures such as RNN and GRU typically require step-by-step computation in chronological order, making effective parallelization difficult. The parallel computing power of multi-head attention helps reduce overall computational complexity.

6. Validity Threats

In some cases, multi-head attention may be easier to overfit due to their complexity, especially if there is relatively little training data. Specifically, the accuracy rate, accuracy rate and F1 value of the modified model in code classification and code clone are slightly lower than the original model, which can be optimized by adjusting the number of heads of the multi-head attention mechanism and increasing the data in the dataset. While the attention mechanism provides a degree of explainability (for example, by looking at attention weights), its inner workings are often more difficult to explain than RNN-based models.

7. Conclusion

This paper presents a method of source code representation based on multi-head attention mechanism. SCRMHA converts a source code snippet into an AST, then splits the AST into small statement trees and encodes them into statement tree vectors, and finally, uses a multi-head attention mechanism to get the vector features of the entire code snippet. Finally, the characteristics of source code snippets are applied to code classification and code clone to evaluate SCRMHA. Experimental results show that SCRMHA performs better than traditional source code representation. The proposed method can effectively parallel process the features of the code. SCRMHA takes more time than ASTNN and the complexity is about 1/3 of ASTNN. The proposed method can effectively capture the syntax and semantic information in the code, and can well represent the features of the source code. The future can be optimized by adjusting the number of heads of multi-head attention mechanisms and increasing industrial datasets.

Acknowledgment

This work is supported by Natural Science Foundation of Fujian Province Science (No. 2022J011238) and Technology Program of Xiamen (No. 3502Z20231042) under grant.

References

[1] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377, 1998.

[2] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[3] Muslim Chochlov, Gul Aftab Ahmed, James Vincent Patten, Guoxian Lu, Wei Hou, David Gregg, and Jim Buckley. Using a nearest-neighbour, bert-based approach for scalable clone detection. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 582–591, 2022.

[4] Osama Ehsan, Foutse Khomh, Ying Zou, and Dong Qiu. Ranking code clones to support maintenance activities. *Empirical Softw. Engg.*, 28(3), apr 2023.

[5] Xin Guo, Ruyun Zhang, Lu Zhou, and Xiaozhen Lu. Precise code clone detection with architecture of abstract syntax trees. In Lei Wang, Michael Segal, Jenhui Chen, and Tie Qiu, editors, *Wireless Algorithms, Systems, and Applications*, pages 117–126, Cham, 2022. Springer Nature Switzerland.

[6] Yutao Hu, Deqing Zou, Junru Peng, Yueming Wu, Junjie Shan, and Hai Jin. Treecen: Building tree graph for scalable semantic code clone detection. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

[7] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[8] Lili Mou, Ge Li, Zhi Jin, Lu Zhang, and Tao Wang. Tbcnn: A tree-based convolutional neural network for programming language processing. *ArXiv*, abs/1409.5718, 2014.

[9] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480, 2014.

[10] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. *ArXiv*, abs/1503.00075, 2015.

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[12] Kesu Wang, Meng Yan, He Zhang, and Haibo Hu. Unified abstract syntax tree representation learning for cross-language program classification. In *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*, pages 390–400, 2022.

[13] Wenjie Wang, Zihan Deng, Yinxing Xue, and Yun Xu. Ccstokener: Fast yet accurate code clone detection with semantic token. *Journal of Systems and Software*, 199:111618, 2023.

[14] XU Yun WANG Wen-jie. Code clone detection based on token semantics. *Computer System Applications*, 31(60-67), 2022.

[15] Hui-Hui Wei and Ming Li. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI'17, page 3034–3040. AAAI Press, 2017.

[16] Zhipeng Xue, Zhijie Jiang, Chenlin Huang, Rulin Xu, Xiangbing Huang, and Liumin Hu. Seed: Semantic graph based deep detection for type-4 clone. In Gilles Perrouin, Naouel Moha, and Abdelhak-Djamel Seriai, editors, *Reuse and Software Quality*, pages 120–137, Cham, 2022. Springer International Publishing.

[17] Feng Zhang, Youliang Wei, and Yucheng Qin. Cross language code plagiarism detection based on program flow chart and graph attention network. *Journal of Chinese Computer Systems*, pages 1–9, 2024.

[18] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794, 2019.