MACA: A Residual Network with Multi-Attention and Core Attributes for Code Search

Lian Gu, Zihui Wang, Jiaxin Liu, Yating Zhang, Dong Yang, Wei Dong
College of Computer Science, National University of Defense Technology, Changsha, China
{gulian, wangzihui98, liujiaxin18, zhangyating18}@nudt.edu.cn, yangdong5002@163.com, wdong@nudt.edu.cn

***Abstract*—Code search technique has gradually become a key skill to accelerate software development. However, the current deep learning methods only use the encoded results and ignores the original content of the code. Besides, the feature expression of the code is too single, which makes the model's understanding insufficient. And the last problem is the lack of separate processing of core attributes, which will cause the model to lack differentiated learning of the attributes with different importance. Therefore, we propose a residual network based on Multi-Attention, so that the model can not only retain the original content of the code but also allow the code to perform a large number of combined learning in different aspects to obtain differentiated features. Then we treat three core attributes and specific implementation of the code differently so that the model can pay extra attention to the core attributes. We use 158,201 Java code-comment pairs for training. In our experimental results, our model is 9.5% higher than the existing method on the indicator of MRR and 12% higher on the SuccessRate@1.**

***Index Terms*—code search, deep learning, residual network, Multi-Attention**

I. Introduction

The code search field can be divided into two historical development stages. The first stage is based on information retrieval methods, and its main strategy is keyword matching; the second stage is based on deep learning methods, and its main strategy is to build a neural network model from query sentences to codes to bridge the semantic gap between the two.

On the one hand, the method based on information retrieval, because it relied on keywords, leads to limited correctness, and may not match codes that are highly similar to the query sentence due to different keywords. On the other hand, the current deep learning method has three problems. The first is that the model only uses the encoded result of the code and discards the original content, which will make the model lose a certain degree of stability. The second is that the model's distributed representation of code features is too single, which makes the model unable to fully learn the code features. The last problem is that these models lack separate processing of core attributes, which will allow code elements of different importance to be treated equally, so that core attributes cannot be expressed as they should in the modeling process. Therefore, we propose a residual network structure based on Multi-Attention. This structure can not only retain the original content of the code but also allows the code to learn different aspects of the combination through a large number of attention mechanisms [1], so as to obtain a more comprehensive representation of the code. Finally, we extracted the three core attributes of the method name, return type, and parameter list separately from the original code content. Then let the distributed representation of these three attributes concatenate the existing coding content so that the model can pay extra attention to the important features of the code.

Our experimental results show that our method is higher than other deep learning methods in both SuccessRate and MRR indicators. This shows that our method effectively improves the performance of code search.

Our contributions are as follows:

- We find that inputting the core attributes of the code separately into the model increases the performance of the model;
- We conduct a comparative experiment on the presence or absence of Multi-Attention and find that Multi-Attention can effectively mine the potential information of code and comments;
- We also set up an experiment, discarding one of the attributes each time, and find that the method name helps the model the most.

II. Related Work

The field of code search has always been a popular research content in academia and industry. It has gone through two research phases, one is realized by information retrieval technology, and the other is realized by deep learning.

At the stage where information retrieval technology is the main method, the code search method has already made many achievements. One of the typical methods is CodeBroker [2], which uses annotations to calculate similarity. Besides, Apache releases Lucene [3], which is an open-source full-text search engine toolkit, which can perform full-text indexing and search with high search efficiency. In addition to the full-text search engine Lucene, there are many code search engines based on information retrieval, such as Codase [4], Koders [5], Krugle [6].

Sachdev et al. set up an experiment to compare the effects of traditional information retrieval methods and deep learning methods on code search tasks [7]. The results show that methods based on deep learning can express more precise

Corresponding author: Wei Dong. This work was supported by National Natural Science Foundation of China (No.62032019, 61690203)
DOI reference number: 10.18293/SEKE2021-079

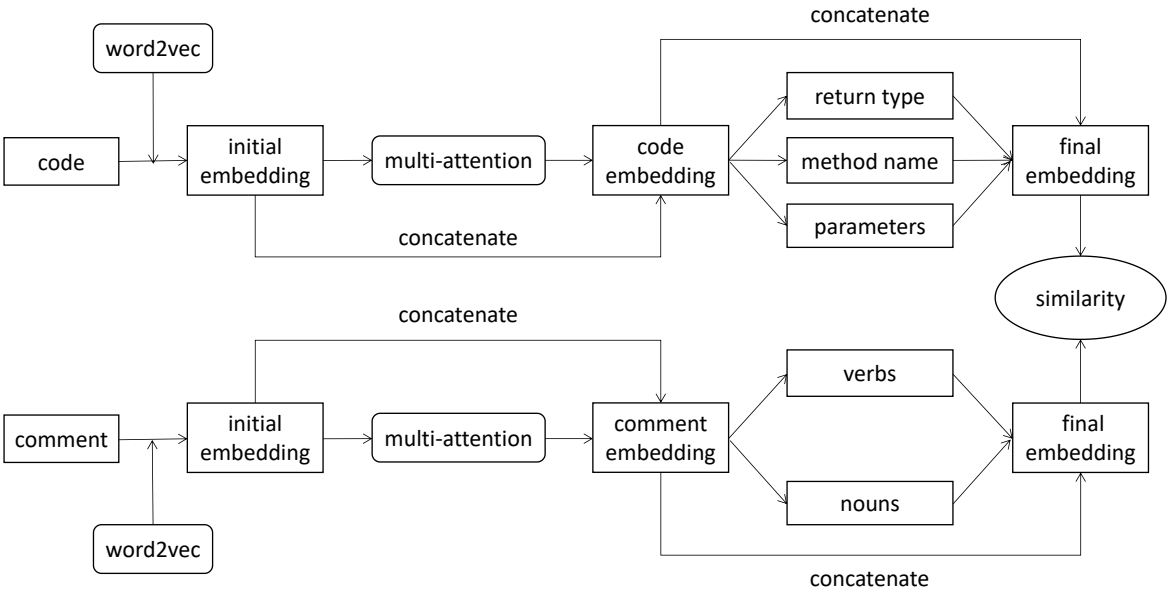

Fig. 1. The structure of the Code-Comment Embedding Neural Network

semantics and achieve better performance. In recent years, a large number of code search methods based on deep learning have emerged. A typical method is NCS [7], which trains the code and query sentence at the same time to obtain a fastText [8] embedding, and then calculates the weight of the code according to IF-IDF [9] to obtain the final code representation vector, and directly averages the embeddings of the query sentence to obtain its final representation. Cambronero et al. improves based on NCS and proposes UNIF [10], which trains a fastText embedding for the code and query sentence respectively, and this embedding can be fine-tuned in the later training phase, and the weighting of the code is changed from TF-IDF to the attention mechanism. Gu et al. split code into three parts according to the characteristics: method name, API sequence, and code tokens [11]. And they train an embedding for each of the three, which encodes the method name and API sequence through the Recurrent Neural Network, and encodes the code tokens through the Multi-Layer Perceptron. Haldar et al. propose a multi-perspective architecture, which calculates the similarity by capturing both global and local similarities [12]. Mou et al. embed codes by a tree-based convolutional neural network [13]. Chen et al. model code and natural language by training two VAEs [14].

III. Method

Our model is a twin tower model based on the deep structured semantic model(DSSM) framework [15]. The twin tower model divides the input into two independent terminals, one for codes and another for comments, and then processes the two different inputs separately. Among them, each end is composed of three layers. The first layer is the input layer, also called the embedding layer. Its function is to process the input into a numeric vector. The second layer is the presentation layer, also called the coding layer. Its function is to process the input vector into a single vector that can represent the entire input. The third layer is the matching layer. Its purpose is to score the similarity of the two final representations. The higher the score, the more similar the two inputs.

A. Input Layer

In the input layer, we usually choose some classic word vector representations. We choose Word2Vec [16] to learn word vectors for code and comments respectively. On the code side, in addition to the coding code itself, we also extracted the three attributes of the return value, method name, and parameter list; on the comment side, in addition to the coding comment itself, we also extracted the commented verbs and nouns. We believe that these separately extracted features can better help the model to express.

B. Presentation Layer

The presentation layer is the core of the entire model. Its role is to encode a collection of word vectors representing code and comments into a single word vector, which represents the entire code or comment.

From the framework of the presentation layer, both the code side and the comment side are two residual network structures. On the one hand, the model needs to learn new information from the original content, and on the other hand, it also needs to retain the original information to a certain

extent. Therefore, we use Multi-Attention to generate new content and then concatenate the original content. This is the first residual structure that learns new content by itself through the model. In the second stage, the model also needs to retain the existing content given artificially. On the code side, these contents are the return type, parameter list, and method name; on the comment side, these contents are the verbs and nouns that appear in the comment.

Because the attention mechanism is only an understanding of one aspect of the code, we have performed multiple attention calculations on the code, and we call this process Multi-Attention. The attention mechanism of each time is calculated as follows:

$$\mathrm{a}_{i,k} = \frac{\exp\left(a_k \cdot e_i^T\right)}{\sum_{i=1}^{n} \exp\left(a_k \cdot e_i^T\right)} \tag{1}$$

Among them, $a_{i,k}$ is the weight of each e_i vector, and a_k is the attention weight coefficient. The target combined vector can be calculated as follows:

$$\mathrm{v_k} = \sum_{i=1}^{n} a_{i,k} \mathrm{e_i} \tag{2}$$

where v_k is the k-th vector of the target vectors.

Finally, the word vector representing the entire code is cascaded to the new vector generated by Multi-Attention and is input to the encoding stage as detailed information together. When summing up the embedding of the four parts, we directly obtain the final code vector representation by averaging.

On the comment side, we first extract the verbs and nouns in the comment sentence through the Natural Language Toolkit(NLTK) and input the word vectors of the two separately into the final representation. The encoding of the entire sentence is consistent with the code side. First, additional information about the comment is obtained through Multi-Attention, then these vectors are concatenated to the original vector, and finally, the average is taken.

C. Match Layer

The presentation layer has coded the code and the comment into a vector respectively, and the function of the matching layer is to score the similarity of the two vectors representing the code segment and the natural language comment. For two vectors with equal dimensions, we generally use cosine similarity for calculation. The higher the cosine similarity score, the closer the two vectors are. The calculation formula of cosine similarity is as follows:

$$\cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}} \tag{3}$$

Among them, A and B respectively represent a vector, A_i represents the i-th element in the A vector, and B_i represents the i-th element in the B vector.

IV. Experiment

A. Dataset

a) Data Collection: We first obtain Java projects with a stat greater than or equal to 10 on GitHub through the crawler. Then we parse each Java file in the project through the AST parser of Java Development Tools(JDT) to get information such as comment, method name, return type, parameter list, and method body. Among them, each function corresponds to a piece of data. We finally get 158,201 pieces of data, and then randomly selected 500 pieces of data as the test set.

b) Preprocessing: Because the naming convention in Java follows the camel case principle, and the actual semantics is a single word in the variable name instead of the entire, so we also de-camelize the function name and variable name according to the regular expression and keep the content Words with precise semantics. At the same time, to be more stable in the subsequent training of word vectors, we convert all words to lowercase.

c) Data set training: In the training phase, our training set is a triple which consists of the following parts: a code segment, a natural language description, and an integer tag. The value of this tag is either 1 or 0. 1 means that the code segment and the natural language description are the data in the original data set. And 0 means that this code segment does not match the natural language description, which is generated by random negative sampling. In our experimental data, the number of our negative samples is equal to the number of original samples.

B. Experimental Setup

In our experiment, our data set is trained for 10 epochs, and the batch size is set to 100. In the setting of the count of Multi-Attention, we find that 30% of the number of original embedding vector sets is the best. If the original vector set has 100 vectors, then 30 vectors will be generated after the Multi-Attention. The code segment and the Word2Vec vector described by natural language are trained separately, and the dimension of the word vector of both is 128. Our model is implemented on the TensorFlow framework, and the optimizer selected during training is Adam.

Our similarity is calculated by cosine similarity. For a training data triple <code, description, flag>, our loss function is defined as follows:

$$\mathcal{L}(\theta) = \frac{\sum_{i=1}^{n} \left(\cos\left(a_i, b_i\right) - flag_i\right)^2}{N} \tag{4}$$

Where N represents the number of training samples, $cos(\cdot)$ is the cosine similarity mentioned above, the value of the flag is 0 or 1, 0 means randomly sampled data, 1 means original sample data.

C. Evaluation Index

In the test indicators, we adopt the commonly used SuccessRate@k and MRR. MRR is the average value of the inverse of the ranking of all test data in the sample, and SuccessRate@k is the ratio of the number of all samples ranked before K.

D. Baseline

We chose UNIF, a classic model among the code search models. The model is very lightweight. On the code side, UNIF transmits the token vector of the entire code to the encoder in the form of a bag of words. The encoder completes the final representation of the code through the attention mechanism; on the comment side, UNIF directly averages the word vectors of all words to get the final representation of the comment. Because the model is relatively simple, the actual effect of each sub-module of our model can be observed more clearly in the experiment.

E. Results

We set up two comparative experiments, one of which is used to prove the effectiveness of the module we designed, and the other experiment is used to compare the degree of improvement of each artificially extracted core attribute on the model effect. The first experiment is to prove the effectiveness of manually extracting core attributes and Multi-Attention separately:

TABLE I
EFFECT OF EACH MODALITY. (BEST SCORES ARE IN BOLDFACE.)

Model	*SR@1*	*SR@5*	*SR@10*	*MRR*
UNIF	0.536	0.764	0.796	0.632
MACA(Multi-Attention)	0.57	0.754	0.8	0.652
MACA(Core Attributes)	0.608	0.79	0.85	0.691
MACA(MA+CA)	**0.656**	**0.808**	**0.868**	**0.727**

It can be seen from the experimental data from table I that our model performs better than the UNIF model. In terms of all indicators, our model is 10% higher than UNIF as a whole. In particular, on the SuccessRate@1 indicator, our model effect has increased by 12%, and on the MRR indicator, our model effect has increased by 9.5%. , Which shows that our model is very accurate in calculating the similarity between comments and code.

Then we remove the core attributes and Multi-Attention respectively. We find that the effect of removing these core attributes is worse than removing Multi-Attention, but the effect of the two is still better than UNIF. After removing the core attributes, MRR dropped by 7.5%; after removing Multi-Attention, MRR dropped by 3.6%. Therefore, for the entire model, human input of core attributes can greatly improve the model's effectiveness.

TABLE II
THE EFFECT OF REMOVING ONE OF THE ATTRIBUTES

Model	*SR@1*	*SR@5*	*SR@10*	*MRR*
MACA	0.656	0.808	0.868	0.727
MACA-w/o.MethodName	0.598	0.774	0.816	0.676
MACA-w/o.ReturnType	0.644	0.806	0.856	0.718
MACA-w/o.Parameter	0.64	0.788	0.848	0.711
MACA-w/o.Verbs	0.65	0.802	0.858	0.721
MACA-w/o.Nouns	0.652	0.796	0.854	0.719

We finally set another comparative experiment, which was to remove one of the core attributes to see how the model's effect declined. It can be seen from table II that when the core attribute of the method name is removed, the MRR index drops the most, up to 5.1%. When the parameter list is removed, MRR drops by 1.6%. Among other core attributes, the decline is minimal. This shows that it is necessary to input the two core attributes of the method name and parameter list to the model.

V. CONCLUSION

We propose a model based on the residual network, which deliberately encodes the return type, parameter list, and method name of the code on the basis of encoding the entire code. The core attributes are entered separately to enhance the modeling of the code by these attributes. In addition, in the process of encoding the entire code and natural language description, our model generates fresh vectors through Multi-Attention, and these new vectors again input to the presentation layer in the form of residuals as important content. Our experiments show that adding these important contents separately will enhance the effect of the model.

REFERENCES

[1] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1 – 29, 2019.

[2] Y. Ye and G. Fischer, "Supporting reuse by delivering task-relevant and personalized information," in *ICSE '02*, 2002.

[3] Lucene, https://lucene.apache.org/.

[4] Codase, https://www.codase.com/.

[5] Koders, https://www.koders.com/.

[6] Krugle, https://www.krugle.com/.

[7] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra, "Retrieval on source code: a neural code search," *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2018.

[8] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017.

[9] W. Frakes and R. Baeza-Yates, "Information retrieval: Data structures and algorithms," 1992.

[10] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[11] X. Gu, H. Zhang, and S. Kim, "Deep code search," *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 933–944, 2018.

[12] R. Haldar, L. Wu, J. Xiong, and J. Hockenmaier, "A multi-perspective architecture for semantic code search," *ArXiv*, vol. abs/2005.06980, 2020.

[13] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *AAAI*, 2016.

[14] Q. Chen and M. Zhou, "A neural framework for retrieval and summarization of source code," *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 826–831, 2018.

[15] P.-S. Huang, X. He, J. Gao, L. Deng, A. Acero, and L. Heck, "Learning deep structured semantic models for web search using clickthrough data," *Proceedings of the 22nd ACM international conference on Information Knowledge Management*, 2013.

[16] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *ArXiv*, vol. abs/1310.4546, 2013.