Wuhan University
Journal of Natural Sciences

2023, Vol.28 No.3, 237-245

Article ID 1007-1202(2023)03-0237-09
DOI https://doi.org/10.1051/wujns/2023283237

Fine-Tuning Pre-Trained CodeBERT for Code Search in Smart Contract

JIN Huan, LI Qinying
Information Engineering College, Jiangxi University of Technology, Nanchang 330000, Jiangxi, China

Abstract: Smart contracts, which automatically execute on decentralized platforms like Ethereum, require high security and low gas consumption. As a result, developers have a strong demand for semantic code search tools that utilize natural language queries to efficiently search for existing code snippets. However, existing code search models face a semantic gap between code and queries, which requires a large amount of training data. In this paper, we propose a fine-tuning approach to bridge the semantic gap in code search and improve the search accuracy. We collect 80 723 different pairs of <comment, code snippet> from Etherscan.io and use these pairs to fine-tune, validate, and test the pre-trained CodeBERT model. Using the fine-tuned model, we develop a code search engine specifically for smart contracts. We evaluate the Recall@*k* and Mean Reciprocal Rank (MRR) of the fine-tuned CodeBERT model using different proportions of the fine-tuned data. It is encouraging that even a small amount of fine-tuned data can produce satisfactory results. In addition, we perform a comparative analysis between the fine-tuned CodeBERT model and the two state-of-the-art models. The experimental results show that the fine-tuned CodeBERT model has superior performance in terms of Recall@*k* and MRR. These findings highlight the effectiveness of our fine-tuning approach and its potential to significantly improve the code search accuracy.

Key words: code search; smart contract; pre-trained code models; program analysis; machine learning
CLC number: TP311

0 Introduction

The smart contract is a program that runs automatically on a decentralized platform such as Ethereum[1], and this program is usually written in Solidity program language (https://docs.soliditylang.org/). It is often used for financial transactions, for example, the smart contract has been used to implement crowdfunding initiatives that raised a total of US$6.2 billion from January to June of 2018. Such money-related activities require smart contracts with security and low-cost. The smart contract with the security vulnerabilities results in severe economic loss. In July 2016, attackers exploited the Decentralized Autonomous Organization (DAO) contract (https://etherscan. io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413) and stole more than $60 million. The high-cost smart contract also increases an unexpected enormous economic risk. In the Eplay contract (https://etherscan. io/address/0x41AeB72624f739281b12aDE663791254F32DB6693), more than 2 300 Ethers were not withdrawn because of the gas limitation, where the gas refers to the transaction cost.

To write a secure and low-cost smart contract, programmers usually reuse previously written code snippets by searching through a large-scale codebase[2-6] such as

Received date: 2022-09-23
Foundation item: Supported by Jiangxi Higher Education and Teaching Reform Project (JXJG-20-24-2), Science and Technology Project of Jiangxi Education Department (GJJ212023), and Jiangxi University of Technology Education and Teaching Reform Project (JY2104)
Biography: JIN Huan, female, Associate professor, research direction: service-oriented software engineering. E-mail: 281965782@qq.com

Github (https://github. com/), Etherscan (https://etherscan.io/), and Open Zeppelin (https://www.openzeppelin.com/). This leads to the increasing demand for a code search engine that could effectively find secure and low-cost code snippets for programmers. The code search engine matches natural language (i. e., query) to programming language (i. e., code snippet), which usually depends on models that can break the gap between natural language (NL) and programming language (PL).

The code search methods include information retrieval[7-10], embedding[11-15], and pre-trained language models[16,17]. Many studies[19-22] search the code using information retrieval methods that calculate the similarities between the query and code snippets. However, there is a distinction to be made between NL and PL. It is a challenge to find a relevant code snippet for a query in a large code base without using semantic analysis.

Instead, many studies[11-15] focus on word embedding and sequence embedding methods which train an embedding model using unsupervised and supervised learning methods. However, all these works necessitate a large dataset for training, and they do not obtain a large enough amount of training data (compared with pre-trained models), resulting in the models that do not adequately understand the relationship between NL and PL.

Recently, the pre-trained code model such as CodeBERT[16] has been able to understand the relationship between NL and PL by learning a giant amount of <comment, code snippet> pairs. However, the existing CodeBERT is only pre-trained for six languages (i. e., Java, Python, Ruby, Go, JavaScript, PHP) but has no PL about smart contract such as Solidity. Many studies[16-24] show that pre-trained models can quickly adapt to new tasks with a small amount of fine-tuning data. Based on these studies, we fine-tune the CodeBERT based on the code and comment of Solidity code and use this fine-tuned model to construct a code search engine for code search for smart contracts. We collect a substantial number of smart contract code files from Etherscan, forming a dataset for fine-tuning the pre-trained CodeBERT. Utilizing this fine-tuned CodeBERT, we develop a code search engine. We conduct a series of experiments on an evaluation dataset, demonstrating the effectiveness of few-shot learning. Our code search engine outperforms traditional code search methods .

1 Fine-Tuning CodeBERT for Code Search

This section describes how to collect the dataset (Dataset Collection), process the dataset (Dataset Processing), fine-tune the pre-trained CodeBERT model (CodeBERT Fine-Tuning), and build the code search engine based on the fine-tuned model (Code Search Engine Construction).

1.1 Dataset Collection

We obtain 40 302 contract files (*. sol) from Etherscan. io. These files have been deployed online and audited to be secure and low-cost. Anyone could access them through certain contract addresses. These files are popular as they exist in Solidity documentation (https://docs. soliditylang. org/), Open Zeppelin (https://www.openzeppelin. com/), Video Tutorials (https://www. youtube. com/results? search_query=Solidity) and Examples on GitHub. After that, we filter out the contracts that cannot be compiled by Slither tool[25] since we could not extract the code and comment from them. Finally, we remain 30 844 reliable files.

Because the smart contract ' s execution unit is a function, we extract the comment and source code for each function from the reliable contract files. We do not consider functions without a body because they are not called during execution but overridden, and such functions are typically defined in an interface. We do not consider functions that do not have comments because they cannot compose the format of the dataset <comment, code snippet>. We use the Slither tool[25] to extract each function's source code and corresponding comment. Because a large amount of codes reuse in the Solidity code, many <comment, code snippet> pairs are identical in our collection of pairs. These identical pairs affect the authenticity of the evaluation, so we remove the duplicate ones. Finally, we obtain 80 723 <comment, code snippet> pairs as the dataset.

1.2 Dataset Processing

Because the model can only understand mathematical numbers, we must convert all of the dataset's comments and codes to mathematical numbers. We split all comments and code snippets into tokens and map the tokens into ids. Figure 1 depicts the process of tokenizing the comment associated with a code snippet and the code snippet itself, with the exclusion of any comments within the function body.

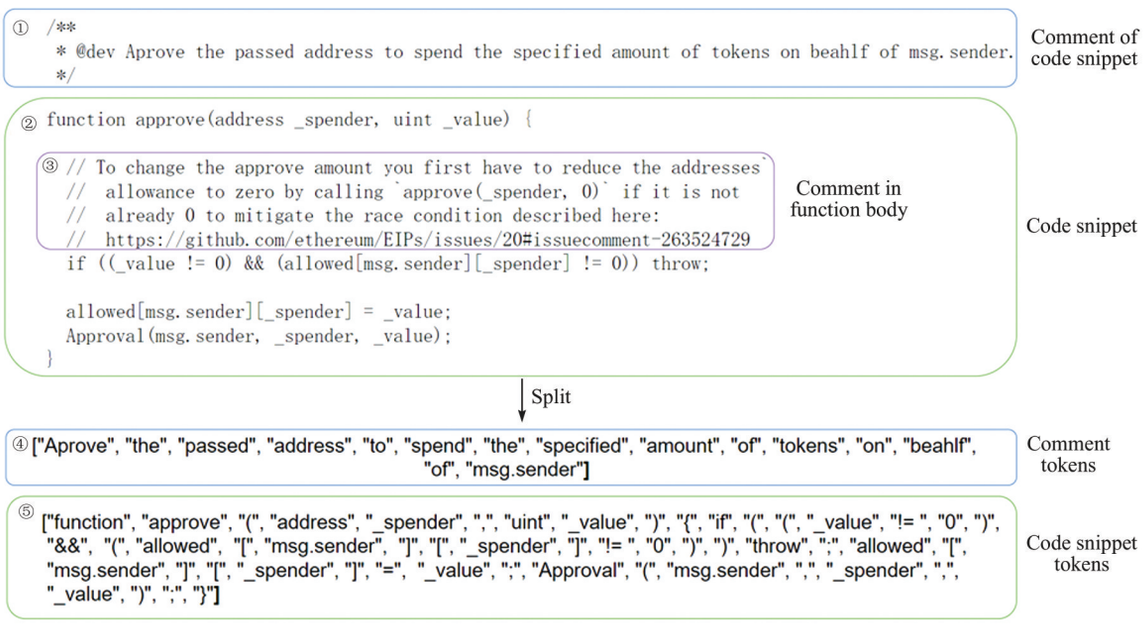

Fig. 1 Code snippet and comment

1.2.1 Split comments and code snippets to tokens

The comment is written in natural language, and we split it directly with spacers. As shown in Fig. 1-①, the *comment of code snippet* is converted to the *comment tokens* in the blue block. The code snippet is written in programming language, and we remove the comment from its body because the mixing of code and comment together to represent the code snippet will affect the semantics of the code itself. The *comment in function body* is removed. Then we split the code snippet with spacers directly. For example, the *code snippet* in Fig. 1-② is converted to the *code snippet tokens* in Fig. 1-⑤. Note that we keep the special symbols in the code snippet, such as {[,; because these symbols contain code semantics. For example, symbol ; is the end marker of a statement, and [] is the marker containing an element of an array.

1.2.2 Map tokens to ids

Figure 2 depicts the process of fine-tuning CodeBERT and using it for code search. Fig. 2-① shows how each pair of code snippets and comments in a <code snippet, comment> pair is cut into tokens and these tokens are converted into token ids for fine-tuning CodeBERT. Fig. 2-② illustrates how a query and all code snippets in code base are cut into tokens and these tokens are converted into token ids, as input to the fine-tuned CodeBERT. Fig. 2-③ explains how to use the vector obtained from the fine-tuned CodeBERT to calculate the similarity between the query and the code snippets in the codebase to recommend the top k most relevant code snippets.

We obtain a Tokenizer (shown in Fig. 2-①) that maps each token to a unique id. This Tokenizer is provided in the Microsoft CodeBERT base, which contains 50 265 <token, id> mappings. There are five special tokens: <*s*>, </*s*>, <unk>, <pad>, and <mask>. <*s*> represents the start of a sequence; </*s*> represents the end of a sequence; <unk> represents a token that not exists in the dictionary; <pad> represents a null character used to occupy a position and has no real meaning; <mask> is a special token for the masked language modelling (MLM). Each token sequence starts and ends with <*s*> and </*s*>, respectively, and each token in it is mapped to an id by *Tokenizer*. For example, <*s*> is mapped to 0, </*s*> is mapped to 2. If there is a token that does not belong to the dictionary, it is mapped to 4 (i.e., <unk>).

1.3 CodeBERT Fine-Tuning

Fine-tuning CodeBERT with the Solidity dataset is the incremental training on the original CodeBERT framework. We first introduce the CodeBERT framework and the training mechanism, then describe how to fine-tune CodeBERT.

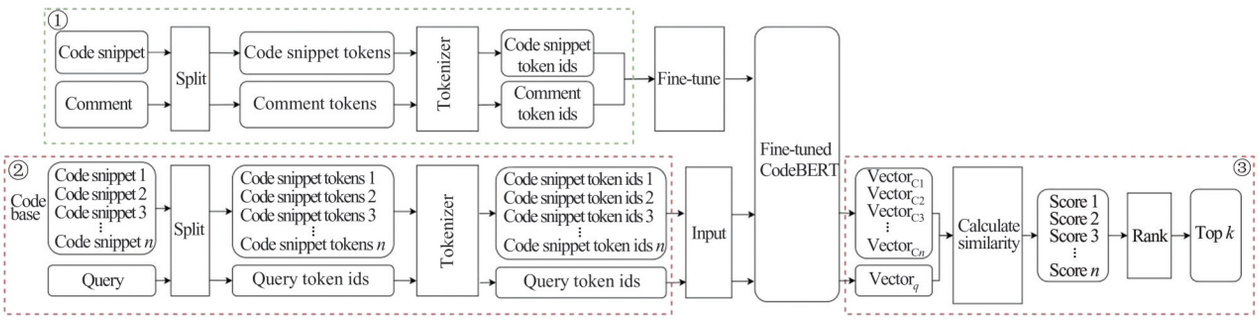

Fig. 2 Process of fine-tuning CodeBERT and using it for code search

1.3.1 CodeBERT framework

CodeBERT follows BERT [26] and RoBERTa [27], and uses a multi-layer bidirectional Transformer [28] as the model architecture. There are two objectives used for training CodeBERT [16]. The first objective is MLM, and the second one is replaced token detection (RTD). The total number of the model parameter is 125 million.

1.3.2 Training mechanism

1) MLM

Given an NL-PL pair datapoint ($x=\{w,\ c\}$) as input, where w is a sequence of NL and c is a sequence of PL, it first selects a random set of positions for both NL and PL to mask out (i.e., m^w and m^c, respectively), and then replaces the selected spots with a specific <mask> token. As mentioned in CodeBERT [16], 15% of the tokens from x are masked out.

$$m_i^w \sim \text{unif}\{1,|w|\} \text{ for } i=1 \text{ to } |w| \tag{1}$$

$$m_i^c \sim \text{unif}\{1,|c|\} \text{ for } i=1 \text{ to } |c| \tag{2}$$

$$w^{\text{masked}} = \text{REPLACE}(w,\ m^w,\ <\text{mask}>) \tag{3}$$

$$c^{\text{masked}} = \text{REPLACE}(c,\ m^c,\ <\text{mask}>) \tag{4}$$

$$x = w + c \tag{5}$$

The goal of MLM is to predict the original tokens that have been masked out, which is expressed as follows:

$$\text{Loss}_{\text{MLM}}(\theta) = \sum_{i \in m^w \cup m^c} -\log p^{D_1}(x_i | w^{\text{masked}}, c^{\text{masked}}) \tag{6}$$

The p^{D_1} is the discriminator that predicts a token from a wide vocabulary.

2) RTD

In the MLM objective, only bimodal data (i.e., datapoints of NL-PL pairs) is used for training. CodeBERT presents the objective of RTD, and adapts it with the advantage of using both bimodal and unimodal data for training. Specifically, there are two data generators here, an NL generator p^{G_w} and a PL generator p^{G_c}, both for generating plausible alternatives for the set of randomly masked positions.

$$\bar{w}_i \sim p^{G_w}(w_i | w^{\text{masked}}) \text{ for } i \in m^w \tag{7}$$

$$\bar{c}_i \sim p^{G_c}(c_i | c^{\text{masked}}) \text{ for } i \in m^c \tag{8}$$

$$w^{\text{corrupt}} = \text{REPLACE}(w,\ m^w, \bar{w}) \tag{9}$$

$$c^{\text{corrupt}} = \text{REPLACE}(c, m^c,\ \bar{c}) \tag{10}$$

$$x^{\text{corrupt}} = w^{\text{corrupt}} + c^{\text{corrupt}} \tag{11}$$

The discriminator is trained to determine whether a word is the original one or not, which is a binary classification problem. The loss function of RTD regarding the discriminator parameterized by θ is given below, where $\delta(i)$ is an indicator function and p^{D_2} is the discriminator that predicts the probability of the i-th word being original.

$$\begin{aligned}\text{Loss}_{\text{RTD}}(\theta) = \sum_{i=1}^{|w|+|c|} \{&\delta(i) \log p^{D_2}(x^{\text{corrupt}}, i) \\ &+[1-\delta(i)][1-\log p^{D_2}([x^{\text{corrupt}}], i)]\}\end{aligned} \tag{12}$$

$$\delta(i) = \begin{cases} 1, & \text{if } x_i^{\text{corrupt}} = x_i \\ 0, & \text{otherwise} \end{cases} \tag{13}$$

There are many ways to implement the generators. CodeBERT implements two efficient n-gram language models with bidirectional contexts, one for NL and one for PL, and learn them from corresponding unimodal datapoints, respectively. The final loss function is given below.

$$\min_{\theta}\ \text{Loss}_{\text{MLM}}(\theta) + \text{Loss}_{\text{RTD}}(\theta) \tag{14}$$

1.3.3 Fine-tuning based on Solidity dataset

We fine-tune the pre-trained CodeBERT MLM and RTD objectives as mentioned in Section 1.3.1. We randomly divide the fine-tuned dataset, validation dataset and test dataset into 8∶1∶1 and obtain 64 579 fine-tuning pairs, 8 072 validation pairs and 8 072 test pairs. As shown in Fig. 2-①, we regard comments as NL and code snippets as PL, split them into tokens, and map them into ids. The length of each id sequence is N. If the length of the ids of a sequence exceeds N, we take the

first $N-1$ ids of the sequence and add the id (i.e., 3) representing the end symbol (i.e., special token </*s*>) to the last id of the sequence. If the length of ids of a sequence is less than *N*, we use id 2 as a pad to fill the sequence to length *N*. Then we get the fine-tuned pair ids <code snippet token ids, comment token ids> and input them to pre-trained CodeBERT for fine-tuning.

In the fine-tuning process, we input the pair ids <code snippet token ids, comment token ids> into the model, and get the NL vector and PL vector. We calculate the score of these two vectors, compute the loss based on the score, and update the parameters based on this loss. We set the model's initial optimal mean reciprocal rank (MRR) score (details in Section 2.1) to 0. After each batch, we calculate the MRR score of the model on the validation set, and if the MRR score is greater than the best score, the model is updated. After ten epochs, we get the fine-tuned CodeBERT.

1.4 Code Search Engine Construction

We use the fine-tuned CodeBERT to search for the best-matched code snippet in the code base for the query (Fig. 2-②③). We convert all code snippets in the code base into ids by splitting, tokenizing, and inputting them into fine-tuned CodeBERT to get the corresponding vectors $\text{Vector}_{C_1}, \text{Vector}_{C_2}, \cdots, \text{Vector}_{C_n}$, where *n* is the number of code snippets in the code base. Each vector dimension is *D*. We stitch these vectors to get a matrix $\boldsymbol{M}$ of size $n \times D$ that represents the whole code base. We transform the query into ids and input them into fine-tuned CodeBERT to get the corresponding Vector_q.

We compute the similarities of Vector_q with the matrix $\boldsymbol{M}$ to obtain an *n*-dimensional vector. The value of each dimension of this vector represents the similarity of the query to the code snippet in the code base. Finally, we select the top-*k* values with the highest scores, get their indexes, and get the corresponding code snippets in the code base based on the indexes.

2 Evaluation

The experiments in this paper are implemented with Python 3.7, and are run on NVIDIA GeForce RTX 3090 GPU. The training epoch is 10; the PL max length is 256; the NL max length is 128; the training batch size is 32; the eval batch size is 64; the learning rate is 2E−5; the *transformers* version is 4.18.0. The hyper-parameters are set as in CodeXGLUE (https://github.com/microsoft/CodeXGLUE1), the loss function is *CrossEntropyLoss*(), the optimizer is *AdamW*, and the tokenizer function is the *RobertaTokenizer*.

To assess the effectiveness of our approach, we have devised the following experiments:

1) Evaluation of results with limited fine-tuned data: We aim to determine if promising results can be achieved even when using a small quantity of fine-tuned data. This experiment will shed light on the potential efficiency of our approach with limited training resources.

2) Comparison with the traditional code search methods: In this experiment, we will evaluate the performance of our fine-tuned CodeBERT model against the traditional code search techniques, such as the information retrieval and the word embedding. By comparing the outcomes, we can ascertain if our model surpasses these conventional methods in terms of code search accuracy.

2.1 Evaluation Metrics

The performance of a code search engine can be measured by Recall@*k* and MRR. Recall@*k* represents the percentage of the relevant items in top-*k* search results to all relevant items in the item set, the higher the value of Recall@*k*, the better the code search engine's retrieval capability. The formulate is as follows:

$$\text{Recall@}k = \frac{R_k}{R_t} \tag{15}$$

where R_k is the number of relevant items in the top-*k* recommended items, R_t is the number of relevant items in the item set.

MRR measures the order of the recommended items. The higher the value of MRR is, the more relevant items returned by the search engine comes near the top. The formulate is as follows:

$$\text{MRR} = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{\text{Rank}_{Q_i}} \tag{16}$$

where Rank_{Q_i} refers to the rank of the first hit result of Q_i among all search results. If there is no code snippet corresponding to Q_i among the top-*k* results, then let $\frac{1}{\text{Rank}_{Q_i}} = 0$. Likewise, following the general settings in baselines, we calculate the MRR values at *k*=10.

2.2 Evaluation of Results with Limited Fine-Tuned Data

Recent studies have found that pre-trained models can achieve a good result by fine-tuning with a small amount of data. We investigate whether fine-tuning

CodeBERT based on the smart contract dataset can also fit this phenomenon.

We randomly take 1%, 5%, 10%, 20%, 50%, and 100% of the fine-tuned dataset, and fine-tune the models based on these six fine-tuned datasets to obtain six models, which are Fine-tuned CodeBERT-1, Fine-tuned CodeBERT-5, Fine-tuned CodeBERT-10, Fine-tuned CodeBERT-20, Fine-tuned CodeBERT-50, Fine-tuned CodeBERT-100. We test these models on the set of Recall@1, Recall@5, Recall@10, Recall@20, Recall@50, Recall@100 and MRR.

As shown in Table 1, as the fine-tuning data increases, the Recall@1-100 and MRR gradually increases. This suggests that the more fine-tuned the data, the better the model can be adapted to a specific downstream task. Surprisingly, the MRR of pre-trained CodeBERT is only 0.263 7, indicating that CodeBERT's generalization ability without fine-tuning is too strong to handle specific tasks like Code Search. However, the Recall@*k* of pre-trained CodeBERT exceeds 0.7, implying that the characteristics of Solidity code may be like some code (e.g., java) that has seen during pre-training, allowing pre-trained CodeBERT to understand some of the Solidity code as well.

When only 1% of the data is used for fine-tuning, the model's Recall@1-100 and MRR increase dramatically (Recall@*k* increases by 0.210 9-0.355 1, MRR increases by 0.330 5). This demonstrates how a small amount of fine-tuned data can significantly improve the model's ability to adapt to a particular task. When we provide more than 10% fine-tuning data for the model, the Recall@1 is greater than 0.6, indicating that a small amount of fine-tuning data is sufficient to make the model perform well on a specific task.

Table 1 Recall@*k* and MRR of the models obtained based on fine-tuning of different scaled dataset

Metrics	Pre-trained CodeBERT	Fine-tuned CodeBERT-1	Fine-tuned CodeBERT-5	Fine-tuned CodeBERT-10	Fine-tuned CodeBERT-20	Fine-tuned CodeBERT-50	Fine-tuned CodeBERT-100
Recall@1	0.187 6	0.542 7	0.588 6	0.609 6	0.613 1	0.621 3	0.623 7
Recall@5	0.356 0	0.652 8	0.753 2	0.778 1	0.781 6	0.795 4	0.797 2
Recall@10	0.446 9	0.744 1	0.801 8	0.845 5	0.868 8	0.870 9	0.872 3
Recall@20	0.528 5	0.759 4	0.874 2	0.904 5	0.918 9	0.920 3	0.921 4
Recall@50	0.629 8	0.852 9	0.876 2	0.914 8	0.924 7	0.929 4	0.940 1
Recall@100	0.703 9	0.914 8	0.930 3	0.947 1	0.954 8	0.961 3	0.969 8
MRR	0.263 7	0.594 2	0.624 7	0.634 6	0.648 1	0.650 2	0.651 5

2.3 Comparison with Traditional Code Search Methods

There are various methods for code search, including information retrieval and word embedding. We want to investigate if our approach can outperform traditional approaches in terms of Recall@*k* and MRR metrics. We use the best model (Fine-tuned CodeBERT-100) obtained in Section 2.2 to represent the Fine-tuned CodeBERT.

We choose BM25 and word2vec as the baselines for information retrieval and word embedding, respectively.

1) BM25 for code search

As shown in Fig. 3, inputting a query and *n* code snippets, we split them into tokens. We calculate the similarities between query tokens and code snippet tokens based on the BM25 model, and get the scores. The scores represent the similarity between each query and code snippet. Then we select the top-*k* values with the highest scores, get their indexes, and get the corresponding code snippets in the code base based on the indexes. We implement BM25 by using the Python package *gensim.summ arization.bm*25.*BM*25.

2) Word embedding for code search

As shown in Fig. 4, we split the code snippet and comment on the fine-tuning dataset into tokens and train a word2vec model on these tokens. For testing, we enter a query and *n* code snippets and split them into tokens. We use the word2vec model to map each token to a vector, then sum and average all the vectors, and then collect all the code vectors into a matrix. Finally, we compute the cosine similarities between the query vector and each code snippet vector and arrange them in reverse order based on the cosine similarity to obtain the

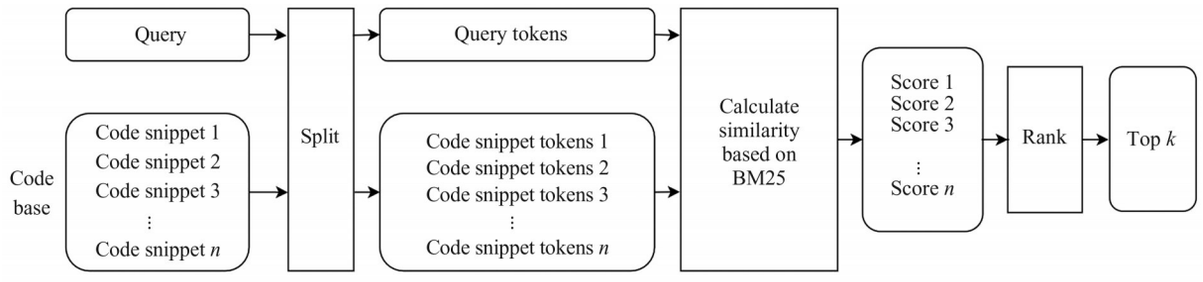

Fig. 3 Code search based on BM25

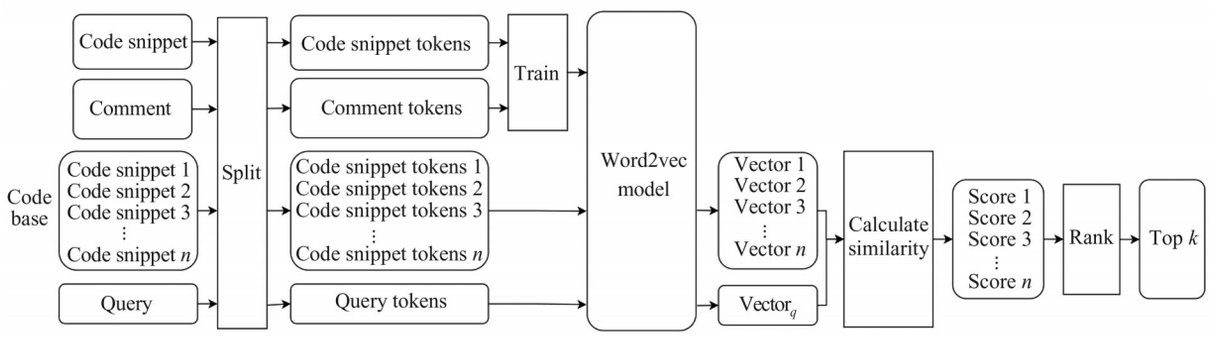

Fig. 4 Code search based on Word2vec

top-*k* values. Then we choose the top-*k* values with the highest scores, obtain their indexes, and locate the corresponding code snippets in the code base. We take the top-*k* values with the highest scores, get their indexes, and find the corresponding code snippets in the code base based on the indexes. The hyperparameters of training the Word2vec are as follows: *training_epoch* is 20; *vec_dim* is 768.

We test BM25, Word2vec, Pre-trained CodeBERT and Fine-tuned CodeBERT on the set of Recall@1, Recall@5, Recall@10, Recall@20, Recall@50, Recall@100 and MRR.

As shown in Table 2, BM25 outperforms Word2vec and Pre-trained CodeBERT in terms of Recall@*k* and MRR. This is because Solidity's code snippets are collected from Etherscan.io, which are written in a standardized manner and can express certain semantic information, and help query to match part of the codes. However, the results of BM25 are still not good enough, the Recall@1 is below 0.25, and the MRR is below 0.27.

Word2vec performs the worst, with a recall@100 of less than 0.35. We examine the Word2vec output and discover that the top-*k* results recommended by Word2vec are nearly identical regardless of the input query. This could be due to a lack of training data, which causes the model to fail to understand the relationship

Table 2 Comparison of BM25, Word2vec and Fine-tuned CodeBERT in terms of Recall@*k* and MRR

Metrics	BM25	Word2vec	Pre-trained CodeBERT	Fine-tuned CodeBERT-100
Recall@1	0.242 7	0.117 6	0.187 6	0.623 7
Recall@5	0.358 8	0.135 6	0.356 0	0.797 2
Recall@10	0.504 1	0.246 9	0.446 9	0.872 3
Recall@20	0.559 4	0.298 5	0.528 5	0.921 4
Recall@50	0.652 9	0.319 8	0.629 8	0.940 1
Recall@100	0.714 8	0.343 9	0.703 9	0.969 8
MRR	0.264 2	0.123 7	0.263 7	0.651 5

between natural language and programming language. The Word2vec model also fails to understand the relationship between two different codes because it generates nearly the same vector for two tokens.

In Recall@*k* and MRR, fine-tuned CodeBERT outperforms BM25, Word2vec, and pre-trained CodeBERT. This demonstrates that fine-tuning large training models can stimulate the model's potential. Fine-tuned CodeBERT has a Recall@100 of nearly 0.97, which means that the correct answer must be among the top 100 results recommended by the model. The Recall@1 of fine-tuned CodeBERT is approximately 0.62, indicating that the model can effectively recommend codes even in the most difficult cases. The MRR is around 0.65, indicating that the model can first rank the correct answers in many cases.

3 Conclusion

This paper describes a method that searches smart contract code snippets for a query. We begin by gathering a comprehensive dataset comprising 40 302 source code files from Etherscan.io. Leveraging this extensive dataset, we proceed to fine-tune the pre-trained CodeBERT model, enhancing its performance and applicability. Based on the fine-tuned CodeBERT, we construct a powerful code search engine. To assess the efficacy of our code search engine, we conduct a series of experiments to validate its effectiveness and performance. Based on previous research, this paper uses different percentages of data for fine-tuning and shows that a small amount of fine-tuned data can help the fine-tuned CodeBERT achieve a good result. The fine-tuned CodeBERT outperforms traditional models such as information retrieval and word embedding on Recall@*k* and MRR.

In the future, we will add more high-quality data to fine-tune pre-trained CodeBERT and build a better model for smart contract code search. Furthermore, because fine-tuning pre-trained CodeBERT with solidity code yields an efficient model, we will extend the fine-tuning method to other solidity code tasks, such as clone detection and code summarization.

References

[1] Wood D D. Ethereum: A secure decentralised generalised transaction ledger[J]. *Ethereum Project Yellow Paper*, 2014 (1):1-32.

[2] Chen X P, Liao P Y, Zhang Y X, *et al*. Understanding code reuse in smart contracts[C]//2021 *IEEE International Conference on Software Analysis*, *Evolution and Reengineering* (*SANER*). New York: IEEE, 2021: 470-479.

[3] He N Y, Wu L, Wang H Y, *et al*. Characterizing code clones in the Ethereum smart contract ecosystem[C]//*Financial Cryptography and Data Security*. Berlin: Springer-Verlag, 2020: 654-675.

[4] Vacca A, Fredella M, Di Sorbo A, *et al*. An empirical investigation on the trade-off between smart contract readability and gas consumption[C]//*Proceedings of the* 30*th IEEE/ACM International Conference on Program Comprehension*. New York: ACM, 2022: 214-224 .

[5] Guida L C, Daniel F. Supporting reuse of smart contracts through service orientation and assisted development[C]// 2019 *IEEE International Conference on Decentralized Applications and Infrastructures* (*DAPPCON*). New York: IEEE, 2019: 59-68.

[6] Shi C C, Xiang Y, Yu J S, *et al*. Semantic code search for smart contracts[EB/OL]. [2022-09-12]. https://arxiv.org/abs/2111.14139.

[7] Smirnov Y V. Subject search in modern library information retrieval systems[J]. *Scientific and Technical Libraries*, 2021, **1**(7): 87-96.

[8] Holzbaur L, Hollanti C, Wachter-Zeh A. Computational code-based single-server private information retrieval[C]// 2020 *IEEE International Symposium on Information Theory* (*ISIT*). New York: IEEE, 2020: 1065-1070.

[9] Trotman A, Lilly K. JASSjr: The minimalistic BM25 search engine for teaching and learning information retrieval[C]// *Proceedings of the* 43*rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. New York: ACM, 2020: 2185-2188.

[10] Jiang H, Nie L M, Sun Z Y, *et al*. ROSF: Leveraging information retrieval and supervised learning for recommending code snippets[J]. *IEEE Transactions on Services Computing*, 2019, **12**(1): 34-46.

[11] Cambronero J, Li H Y, Kim S, *et al*. When deep learning met code search[C]//*ESEC/SIGSOFT.* New York: ACM, 2019: 964-974.

[12] Gu X D, Zhang H Y, Kim S. Deep code search[C]//2018 *IEEE/ACM* 40*th International Conference on Software Engineering* (*ICSE*). New York: IEEE, 2018: 933-944.

[13] Pour M V, Li Z, Ma L, *et al*. A search-based testing framework for deep neural networks of source code embedding [C]//2021 14*th IEEE Conference on Software Testing*, *Verification and Validation* (*ICST*). New York: IEEE, 2021: 36-46.

[14] Shuai J H, Xu L, Liu C, *et al*. Improving code search with co-attentive representation learning[C]//*Proceedings of the* 28*th International Conference on Program Comprehension*. New York: ACM, 2020: 196-207.

[15] Yang J, Fu C, Liu X Y, *et al*. Codee: A tensor embedding scheme for binary code search[J]. *IEEE Transactions on Software Engineering*, 2022, **48**(7): 2224-2244.

[16] Feng Z Y, Guo D Y, Tang D Y, *et al*. CodeBERT: A pre-trained model for programming and natural languages[EB/OL]. [2020-12-25]. https://arxiv.org/abs/2002.08155.

[17] Guo D Y, Ren S, Lu S, *et al*. GraphCodeBERT: Pre-training code representations with data flow[EB/OL]. [2020-12-25]. https://arxiv.org/abs/2009.08366.

[18] Wan Y, Zhao W, Zhang H Y, *et al*. What do they capture? : A structural analysis of pre-trained language models for source code[C]//*Proceedings of the* 44*th International Conference on Software Engineering*. New York: ACM, 2022: 2377-2388.

[19] Yuan X E, Lin G J, Tai Y H, *et al*. Deep neural embedding for software vulnerability discovery: Comparison and optimization[J]. *Security and Communication Networks*, 2022, **2022**: 1-12.

[20] Karmakar A, Robbes R. What do pre-trained code models know about code? [C]//2021 36*th IEEE/ACM International Conference on Automated Software Engineering* (*ASE*). New York: IEEE, 2022: 1332-1336.

[21] Wang C Z, Yang Y H, Gao C Y, *et al*. No more fine-tuning? An experimental evaluation of prompt tuning in code intelligence [EB/OL]. [2021-02-25]. https://arxiv.org/abs/2207.11680.

[22] Han X, Zhang Z Y, Ding N, *et al*. Pre-trained models: Past, present and future[J]. *AI Open*, 2021, **2**: 225-250.

[23] Bisht M, Gupta R. Fine-tuned pre-trained model for script recognition[J]. *International Journal of Mathematical*, *Engineering and Management Sciences*, 2021, **6**(5): 1297-1314.

[24] Liu B Y, Cai Y F, Guo Y, *et al*. TransTailor: Pruning the pre-trained model for improved transfer learning[J]. *Proceedings of the AAAI Conference on Artificial Intelligence*, 2021, **35** (10): 8627-8634.

[25] Feist J, Grieco G, Groce A. Slither: A static analysis framework for smart contracts[C]//2019 *IEEE/ACM* 2*nd International Workshop on Emerging Trends in Software Engineering for Blockchain* (*WETSEB*). New York: IEEE, 2019: 8-15.

[26] Devlin J, Chang M W, Lee K, *et al*. BERT: Pre-training of deep bidirectional transformers for language understanding [EB/OL]. [2020-12-25]. https://arxiv.org/abs/1810.04805.

[27] Liu Y H, Ott M, Goyal N, *et al*. Roberta: A robustly optimized bert pretraining approach[EB/OL]. [2020-12-25]. https://arxiv.org/abs/1907.11692.

[28] Vaswani A, Shazeer N, Parmar N, *et al*. Attention is all you need[C]// *NIPS*'17: *Proceedings of the* 31*st International Conference on Neural Information Processing Systems.* New York: ACM, 2017: 6000-6010.

□