CUTE: A Collaborative Fusion Representation-Based Fine-Tuning and Retrieval Framework for Code Search

Qihong Song[1,2], Jianxun Liu[1,2(✉)], and Haize Hu[1,2]

[1] School of Computer Science and Engineering, Hunan University of Science and Technology, Xiangtan, China
904500672@qq.com

[2] Key Laboratory of Knowledge Processing and Networked Manufacturing, Hunan University of Science and Technology, Xiangtan, Hunan, China

Abstract. Code search aims at searching semantically related code snippets from the large-scale database based on a given natural descriptive query. Fine-tuning pre-trained models for code search tasks has recently emerged as a new trend. However, most studies fine-tune models merely using metric learning, overlooking the beneficial effect of the collaborative relationship between code and query. In this paper, we introduce an effective fine-tuning and retrieval framework called CUTE. In the fine-tuning component, we propose a Collaborative Fusion Representation (CFR) consisting of three stages: pre-representation, collaborative representation, and residual fusion. CFR enhances the representation of code and query, considering token-level collaborative features between code and query. Furthermore, we apply augmentation techniques to generate vector-level hard negative samples for training, which further improves the ability of the pre-trained model to distinguish and represent features during fine-tuning. In the retrieval component, we introduce a two-stage retrieval architecture that includes pre-retrieval and refined ranking, significantly reducing time and computational resource consumption. We evaluate CUTE with three advanced pre-trained models on CodeSearchNet consisting of six programming languages. Extensive experiments demonstrate the fine-tuning effectiveness and retrieval efficiency of CUTE.

Keywords: Code search · Collaborative fusion representation · Fine tuning · Hard negative sample · Data augmentation

1 Introduction

With the development of the open-source community, code search and reuse are becoming essential for improving coding efficiency. Giving a descriptive natural language query, how to accurately retrieve semantically relevant code snippets (positive samples) from others (negative samples) in large repositories has become a crucial challenge [1]. In early studies, code search models mainly

H. Gao et al. (Eds.): CollaborateCom 2023, LNICST 561, pp. 343–362, 2024.
https://doi.org/10.1007/978-3-031-54521-4_19

employed the Information Retrieval (IR) technique [2–4], based on superficial textual similarity, which was limited by the vast gap. The gap refers to code and query from different forms in grammar and expression [5], but we need to find code and query pairs that express the same semantics. With the development of Deep Learning (DL), Metric Learning (ML) [6] used in DL models can align code and query features in a high-dimensional vector space [7–9], reducing linguistic disparities and facilitating similarity calculation, because ML aims at shrinking the vector distance of positive pairs while enlarging the distance of negative pairs. As illustrated in Fig. 1, after the process of ML, the relevant code and query vectors are aggregated together, while the unrelated vectors are pushed apart. Recently, pre-trained models related to programming languages have been proposed, demonstrating excellent performance on code search tasks [10–12]. Pre-trained models are large DL models pre-trained on multiple pre-training tasks, such as masked language modeling, replaced token detection, etc., on large-scale datasets to learn universal features and patterns about programming and natural language. The pre-trained models can then be fine-tuned toward specific code search tasks. However, most existing research has focused on the pre-training phase, simply fine-tuned models through supervised ML in the fine-tuning phase for code search task, especially regarding the representation approach and the composition of training samples [13,14].

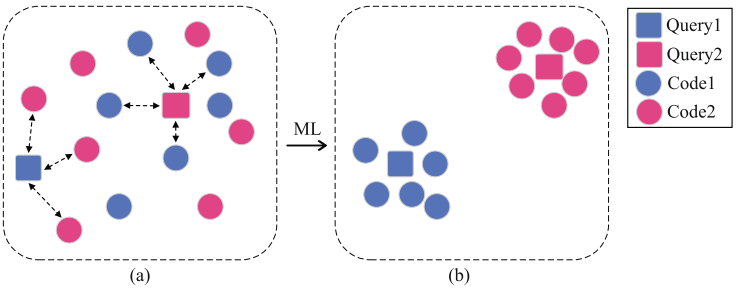

Fig. 1. (a) Vector distribution of code and query. (b) Vector distribution after ML.

For the representation approach, current studies [11–13] generally treat the pre-training model as an encoder to obtain the final representation of code and query. These vectors are directly used to calculate similarity for ML, ignoring the potential benefits of the token-level mapping relationships (collaborative features) between code and query. Hard negative samples are those negative samples similar to positive ones in vector space but should be far apart. Those samples can help guide the model to correct its mistakes more quickly and enhance the representational ability [15–17]. However, model training is performed using mini-batch data that consists of multiple code-query pairs in most code search studies [18,19], where each query has only one relevant code snippet as a positive sample. The remaining code samples within the mini-batch are considered

as negative samples, and they are randomly sampled. Therefore, the mini-batch typically does not contain or contain only a small number of corresponding hard negative samples. Most of these negative samples are easily distinguished from positive samples in vector space, becoming the model performance bottleneck [20].

In this work, we propose CUTE, a collaborative fusion representation-based fine-tuning and retrieval framework, which aims to provide a new paradigm for high-performance fine-tuning and efficient retrieval for code search. Performance improvement is mainly achieved via two key factors of the fine-tuning component: CFR and hard negative sample augmentation. ①There are token-level associations between code-query pairs. For example, as a code-query pair shown in Fig. 2, the token in the query like "contents" has a solid semantic association with the tokens like "file" and "input" in code. Inspired by the co-attention mechanism [21–23], we propose CFR to enhance the representative ability, which consists of three parts: pre-representation (PR), collaborative representation (CR), and residual fusion (RF). The PR stage first represents the code and query as original vectors. The CR stage captures the token-level collaborative features between the original vectors of code and query. The RF stage builds a residual structure [24] to fuse the collaborative vectors and original vectors, alleviating the information lost and benefiting the integrity and accuracy of the representations. ②Inspired by Generative Adversarial Networks (GAN) [25], we employ vector-level augmentation techniques to generate hard negative samples similar to the labeled positive samples for training, which can significantly increase the matching difficulty of code-query pairs. It is worth mentioning that vector-based augmentation does not require modifying the raw data but only involves augmentation based on the vector, which avoids the consumption of time and computational resources.

```
// Read the contents of the file by line
public static void ReadFile(String filePath)
{
    FileReader inputFile = new FileReader(filePath);
    BufferedReader bufferReader = new BufferedReader(inputFile);
    String line;
    while ((line = bufferReader.readLine()) != null){
            System.out.println(line);
      }
    bufferReader.close();
}
```

Fig. 2. An example of a code-query pair.

To provide high-performance code search services, CUTE requires capturing the token-level collaborative features between a given query and all code snippets

in the code repository, which results in longer search time. To improve efficiency, we introduce a two-stage retrieval architecture for the retrieval component of CUTE. In the pre-retrieval stage, we conducted pre-representation to the whole codebase in advance. We thus only need to pre-represent the given query and then select the top-k candidate snippets by computing cosine distance. In the refined ranking stage, we conduct CR and RF to the candidate code vectors and query from the previous stage to get the final vectors with collaborative features. Finally, we rerank the obtained top-k candidate codes based on the final vector and return the re-fined results to the user.

The key contributions of this work can be summarized as follows:

- We propose CUTE, a fine-tuning and retrieval framework for code search, which can be directly applied to fine-tune any pre-trained language model, ensuring both retrieval performance and efficiency.
- For the fine-tuning component, we propose a three-stage representation method CFR, where the collaborative representation stage captures the token-level collaborative features between code and query. Unlike previous studies, we adopt vector-level data augmentation techniques to generate hard negative for better fine-tuning performance.
- For the retrieval component, a two-stage retrieval architecture is proposed, including pre-retrieval and refined ranking, which optimizes the retrieval process and improves the retrieval efficiency.
- We conduct extensive experiments on dataset CodeSearchNet for six programming languages with three advanced pre-training models. The results demonstrate the structural rationality and superior performance of the proposed CUTE.

2 Related Work

2.1 Deep Learning Models for Code Search

Deep learning models can represent code and query into vector space and compare the similarity of aligned feature vectors. Gu et al. [7] combined DL with code search for the first time and proposed the DeepCS, which embeds code and query respectively by two LSTMs and learns the semantic relationship among them. Based on DeepCS, researchers have built models based on DL technologies such as CNN, GNN, and self-attention mechanisms [9,18,26]. Regarding the processing of model inputs, some researchers have transformed code into different forms of Abstract Syntax Tree (AST) sequences for syntactic representation [27,28]. In addition, some models also use transfer learning, meta-learning, and other techniques to increase the model's generalizability for different databases [29,30].

Pretrained models are large DL models trained on large datasets using unsupervised learning techniques to learn statistical and structural features of the input data, which can be fine-tuned for specific downstream tasks. Recently, transformer-based code pretraining models have made significant progress in

tasks such as code search, code clone detection, and code completion, etc. They are pre-trained from different perspectives. CodeBERT, CodeT5, and RoBERTa (code) are pretrained by multiple tasks based on code semantic understanding [11,31]. To enhance the awareness of code structure features, GraphCodeBERT, StructCoder, and UniXcoder [10,12,32] introduce inputs and pretraining tasks related to data flow and AST.

In the fine-tuning for the code search task, ML merely directly obtains the final representation by the pre-trained model, with weaknesses in fine-tuning for the code search task. We thus propose a general fine-tuning and retrieval framework for code search, CUTE, with a better representation approach and sampling strategy.

2.2 Collaborative Attention Mechanism

The attention mechanism [21–23] is widely used in Computer Vision (CV) and Natural Language Processing (NLP), allowing deep learning models to focus on critical parts of the data. Since most models' inputs contain multimodal information, researchers have proposed a collaborative attention mechanism to capture the mapping association among them. This mechanism mainly uses attention mechanism to explore the essential parts of the interaction association between multiple inputs. For example, Ma et al. [33] proposed a multi-step co-attention model for multi-label classification, specifically using the collaborative attention mechanism to analyze the connection between the original text and the leading label, which filter out the error accumulation problem caused by the error prediction. Zhang et al. [34] proposed a network based on the collaborative mechanism to capture the aspect information and surrounding contexts to assign the correct polarity to a given sentence. Shuang et al. [35] applied the mechanism to a CNN model to capture the input connection and improve the code representation ability.

Inspired by the mechanism, we incorporate collaborative representation into the CFR component of CUTE to capture the token-level collaborative features between code and query, which serves as a crucial part for achieving high-performance code search.

3 Methodology

3.1 Overall Architecture

The overall architecture of CUTE is shown in Fig. 3, which contains two components: fine-tuning and retrieval component. In the fine-tuning component, we first use vector-level data augmentation techniques to generate hard negative samples to enlarge the mini-batch. We then propose a three-stage CFR method to represent code and query. Specifically, the PR captures the features of code and query, and the CR captures the token-level collaborative features between code and query, and finally, the RF integrates all features to obtain complete

representation vectors. In this component, the encoder (the pre-trained model) is fine-tuned using metric learning based on the representations obtained from CFR.

In the retrieval stage, we propose a two-stage retrieval architecture consisting of pre-retrieval and refined ranking. We compute the cosine similarity between the representations of code snippets in the codebase (conducted PR in advance) and the pre-representation vector of the given query. The top-k most similar code snippets are selected as the pre-retrieval results. In the refined ranking stage, we perform CR and RF on the candidate code list obtained from pre-retrieval and the given query statement to obtain more accurate representation vectors. The k candidate codes are rearranged based on vector distances to obtain the final recommended code list. This approach balances effectiveness and efficiency, enabling high-speed and high-performance code search.

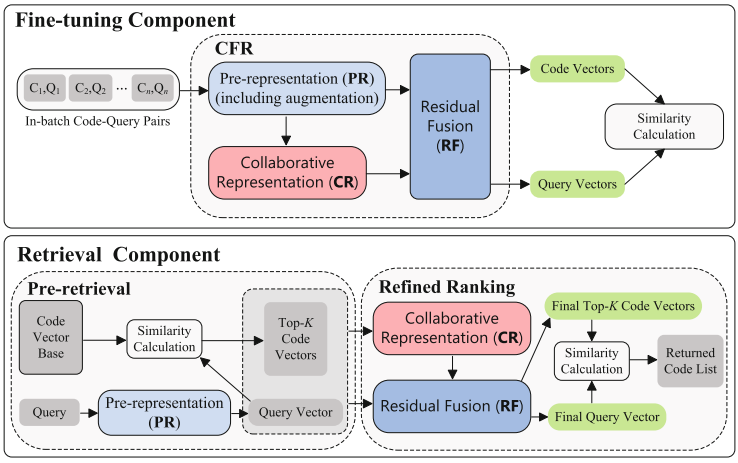

Fig. 3. Overall architecture of CUTE.

3.2 Fine-Tuning Component

In this section, we present the details of the fine-tuning component, including how to generate hard negative samples to enlarge mini-batch, CFR, and how to fine-tune the model using ML.

Generate In-Batch Hard Negative Samples. For a given query, code snippets related to it are called positive samples, and code snippets unrelated to it are called negative samples. Among the negative samples, there exist special samples which are close to the anchor sample but should be far away. Specifically, they are similar to the anchor sample vector but have different semantics.

The samples are defined as hard negative samples. They can help guide a model to correct its mistakes more quickly and effectively during training. However, most existing code search models have not paid enough attention to the samples. Specifically, most existing studies treat other samples in the mini-batch as negative samples during model training. These samples are randomly sampled from the training set (only with a few or no hard negative samples), which resulted in them being easily distinguishable and thus cannot fully optimize the model.

Inspired by GAN, we employ vector-level augmentation techniques to generate hard negative samples for the in-batch samples to enhance the fine-tuning performance. At the vector-level, data augmentation is achieved by perturbing the given code vector, we generate hard negative samples that are similar to the original samples by employing linear interpolation. Linear interpolation mainly uses the features of another sample to augment the given sample, and the method is calculated as Eq. 1.

$$V_i^{'} = \lambda V_i + (1 - \lambda) V_j \tag{1}$$

where V_i is the given sample vector, V_j is randomly sampled from other samples. λ is the interpolation coefficient sampled from a uniform distribution $U(\alpha, \beta)$, and α, β are mutable parameters near 1.0.

Collaborative Fusion Representation (CFR). Our proposed CFR consists of three stages: pre-representation (PR), collaborative representation (CR), and residual fusion (RF). The details are shown in Fig. 4.

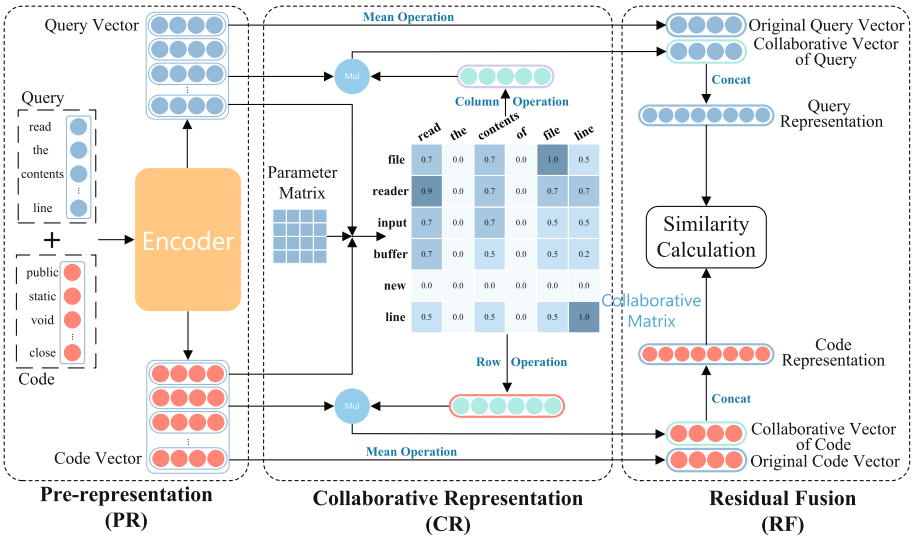

Fig. 4. The architecture of three-stages CFR.

Pre-representation (PR). PR is used to embed semantic features of code and query respectively into a fixed-length dense vector. In detail, given a code and a query of length m and n, we tokenize them into two sequences $S_C = \{C_1, C_2, ..., C_m\}$ and $S_Q = \{Q_1, Q_2, ..., Q_n\}$, then feed them into a encoder to capture the semantic features respectively. Since the encoder embeds each token as the same representation dimension d, we obtain the final representations $O_C \in \mathbb{R}^{d \times m}$ and $O_Q \in \mathbb{R}^{d \times n}$ by passing the tokenized sequences S_C and S_Q through the encoder, respectively, which are calculated as Eq. 2 and Eq. 3.

$$O_C = PR(S_C) \tag{2}$$

$$O_Q = PR(S_Q) \tag{3}$$

Collaborative Representation (CR). After PR, we obtain the collaborative vectors by conducting row & column operations on the matrix I. The matrix is computed by multiplying the two original vectors O_C and O_Q of code and query as Eq. 4.

$$I = tanh(O_C^T U O_Q) \tag{4}$$

where the parameter matrix $U \in \mathbb{R}^{d \times d}$ is introduced to construct collaborative matrix, which continuously be optimized during the training. We use the hyperbolic tangent function to normalize the collaborative value between code and query. The normalization is to facilitate the subsequent calculation.

The collaborative matrix represents the strength of the association between each token of code and query. Since the matrix I is obtained by multiplying the feature vectors of code and query from the PR stage, so each element I_{ij} of I represents the matching level of the ith token C_i in code and the jth token Q_j in query.

Specifically, the middle part of Fig. 4 shows the details of the generated collaborative matrix I from the code-query example shown in Fig. 1. The ith row represents the match level between the ith token in the code and each token in the query, and the jth column represents the match level between the jth token in the query and each token in the code. The color of each square in the matrix represents the mapping association strength of the corresponding tokens. The higher the association strength of two tokens, the darker the color of the corresponding square. Therefore, the collaborative matrix is essential to determine whether the code's function satisfies the requirements of the query. We should primarily consider the tokens with high association strength and pay less attention to irrelevant tokens for the similarity calculation. In this way, we can focus on the crucial semantics for code-query matching.

To obtain the weight that should be assigned to each token in query/code for matching with the whole code/query segment, we perform a series of row & column operations on the collaborative matrix. Since each element in the ith row of the matrix represents the association strength between the ith token in code and each token in query, the vector consisted of the matrix I's ith row thus shows the strength of the association between the ith token in code and the

corresponding whole query segment. We conduct the max-pooling operation on the rows of the matrix as Eq. 5. Specially, we take the maximum value of the ith row vector as the strength of the ith token in code matched with the whole given query. Similarly, for the column vectors of the matrix, we do the same operation as Eq. 6 to get the strength of the jth token in query.

$$d_i^C = maxpooling(I_{i,1}, I_{i,2}, \ldots, I_{i,n}) \tag{5}$$

$$d_j^Q = maxpooling(I_{1,j}, I_{2,j}, \ldots, I_{m,j}) \tag{6}$$

where d_i^C denotes the association strength value of the ith token in the code, and d_j^Q denotes the strength value of the jth token in the query. Therefore, the strength values of the code and query's tokens can be expressed as follows:

$$d^C = [d_1^C, d_2^C, \ldots, d_m^C] \tag{7}$$

$$d^Q = [d_1^Q, d_2^Q, \ldots, d_n^Q] \tag{8}$$

To unify the scale, we use the *softmax* function to normalize d_i^C and d_i^Q to w_i^C and w_i^Q as Eq. 9 and Eq. 10. They denote the matching weight of each token in the code and query.

$$w_i^C = \frac{exp(d_i^C)}{\sum_{k=1}^{m} exp(d_k^C)} \tag{9}$$

$$w_i^Q = \frac{exp(d_i^Q)}{\sum_{k=1}^{n} exp(d_k^Q)} \tag{10}$$

The final token-level weight vectors $W^C \in \mathbb{R}^m$ and $W^Q \in \mathbb{R}^n$ for code and query are shown in Eq. 11 and Eq. 12.

$$W^C = [w_1^C, w_2^C, \ldots, w_m^C] \tag{11}$$

$$W^Q = [w_1^Q, w_2^Q, \ldots, w_n^Q] \tag{12}$$

Based on the weight vectors of the code and query obtained from the collaborative matrix, we combine the original vectors O_C and O_Q obtained from the PR stage with corresponding weights W^C and W^Q by dot product operations as Eq. 13 and Eq. 14. Finally, we obtain the collaborative vectors V_C and V_Q of the code and query, which highlight the token-level crucial collaborative features.

$$V_C = O_C W^C \tag{13}$$

$$V_Q = O_Q W^Q \tag{14}$$

Residual Fusion (RF). The original vectors are semantic features of code and query from the PR stage. Collaborative vectors are obtained by adding token-level collaborative features to the original vectors of code and query. The obtained collaborative vectors will lose some original features in the CFR operation flow. In order to alleviate the information lost during the CFR processes, we treat the collaborative vectors generated in the previous stage as semantic residuals. Then, we fuse original vectors O_C and O_Q, which contain relatively complete semantic information, with the semantic residuals V_C and V_Q to generate the final representation vectors C and Q of code and query as Eq. 15 and Eq. 16. In this way, it not only alleviates the representation loss but also makes the model easier to train by the residual structure. The final generated representations and not only highlight the crucial semantic information by the CFR, but also maximize the integrity of the semantic features.

$$C = V_C \oplus O_C \tag{15}$$

$$Q = V_Q \oplus O_Q \tag{16}$$

Whole Fine-Tuning Process. The fine-tuning process of our proposed CUTE is shown in the fine-tuning component of Fig. 3. Randomly sampled code-query pairs form a mini-batch, they can be represented by CFR, and we can compute the similarity of the obtained vectors to get the matching code snippets. The optimization of the entire fine-tuning process follows ML, aiming to minimize the distance between relevant code and query vectors while maximizing the distance between irrelevant code and query vectors.

In the CFR, we first perform PR on the samples within the mini-batch, generating the original code and query vectors. Next, we apply vector-level augmentation techniques to perturb the original vectors, generating M hard negative samples for each original sample. By augmenting the mini-batch with hard negative samples, we can enhance the efficiency and effectiveness of fine-tuning. Subsequently, we perform CR on the pre-represented and augmented sample vectors. This process incorporates token-level collaborative features between code and query, facilitating the matching between relevant code and query pairs. Finally, we construct a residual structure to fuse the original and collaborative vectors, generating the final representation vectors of code and query.

In a mini-batch with the size of B, there exists a unique code positive sample vector C_r that is semantically related to a given query vector Q_r. Since each sample is augmented to generate M corresponding hard negative samples, the mini-batch size becomes $(M+1)B$, with the remaining $MB+B-1$ code snippets vectors C_s being negative samples semantically unrelated to Q_r. Given a query, ML aims to minimize the distance between the query and its positive sample while maximizing the distance between the query and its negative samples. The ML loss value of the query Q_r in a mini-batch can be calculated as equation Eq. 17, where the dot product is used to calculate similarity. The total loss value is the average of each query loss value.

$$L_r = -log\frac{exp(Q_r \cdot C_r)}{exp(Q_r \cdot C_r) + \sum_{s=1,s\neq r}^{B(M+1)} exp(Q_r \cdot C_s)} \tag{17}$$

3.3 Retrieval Component

The two-stage retrieval architecture of our proposed CUTE is shown in the retrieval component of Fig. 3, which includes pre-retrieval and refined ranking.

In the refined ranking stage, we obtained the top-k most relevant code snippets from the previous stage. Then we perform CR and RF on these pre-representation vectors, which yields representation vectors that consider the collaborative features of given query and candidate code snippets. Next, we rearrange the code snippets and return the final results. Specifically, we have the K candidate original code vectors and the original given query vector from the pre-retrieval stage. We capture the token-level collaborative features between the candidate code snippets and the query by employing CR. We then fuse these features with the original features obtained from previous stage using RF, resulting in the final vectors for candidate code and query. Finally, we rearrange the code snippets from the pre-retrieval stage and provide the user with the final retrieval results.

4 Experimental Evaluation

In order to investigate the structural rationality and the performance of CUTE on code search, several experiments are conducted to answer the following research questions:

- **RQ1:** How effective is the fine-tuning effectiveness of CUTE in code search task, compared with traditional fine-tuning method?
- **RQ2:** How do the different components of CUTE affect the retrieval efficiency and effectiveness?
- **RQ3:** How to better implement CFR in practice to capture collaborative features between code and query?
- **RQ4:** How to choose the optimal parameters for augmentation to ensure high performance of CUTE?

4.1 Experimental Setup

Dataset. In this work, we use large-scale dataset CodeSearchNet [31] to evaluate the effectiveness of the proposed CUTE. The dataset contains six different programming language. We removed low-quality data from the dataset following the method of Guo et al. [12], and we extracted some code snippets from the entire dataset used to construct a search codebase for validation and testing. The details of the preprocessed dataset are shown in Table 1.

Table 1. Details about the preprocessed CodeSearchNet corpus.

Language	Training	Validation	Test	Search Codebase
Python	251,820	13,914	14,918	43,827
PHP	241,241	12,982	14,014	52,660
Go	167,288	7,325	8,122	28,120
cre Java	164,923	5,183	10,955	40,347
JS	58,025	3,885	3,291	13,981
Ruby	24,927	1,400	1,261	4,360

Evaluation Metrics. To evaluate the effectiveness of CUTE, we utilize the most widely used mean reciprocal rank (MRR) for code search [7]. It measures the ranking of the target code in the returned code list, and only cares about the ranking of the most relevant code. The higher the MRR value, the higher the first hit code is ranked. It can be calculated as follows:

$$MRR = \frac{1}{|N|} \sum_{j=1}^{|N|} \frac{1}{Rank_j} \tag{18}$$

where N represents the number of query in the validation/test set, $Rank_j$ denotes the ranking position of the most relevant code in the returned list for the jth query.

Pre-trained Models. This work fine-tunes the three advanced pre-trained models related to programming language. The details are presented as follows.

RoBERTa (Code) [31] is a transformer architecture-based pre-trained model, an improved version of Bert [36]. It is pre-trained on the CodeSearchNet corpus with masked language modeling (MLM) task.

CodeBERT [11] is a bimodal pre-trained model for programming language and natural language. It is pre-trained with two tasks: MLM and replaced token detection.

GraphCodeBERT [12] is a pre-trained model that incorporates code structure information, and it is pre-trained with three tasks: MLM, code structure edges prediction, and alignment representations of source code and code structure.

Implementation Details. All experiments were conducted in a Linux environment using Nvidia GTX 3090 24GB. We employed a traditional fine-tuning approach as our baseline method. This method directly computes the ML loss by leveraging the code and query vectors encoded by the pre-trained model. During training, we input the model with all training data as code-query pairs. We set the augmentation count M to 5 for each code in a mini-batch, the interpolation

coefficient λ is 0.85, the number of training epochs to 10, the batch size to 32, and the learning rate values for RoBERTa(Code), CodeBERT, and GraphCodeBERT are 1e–5, 1e–5, and 2e–5. During validation/test, we randomly sample 1000 query statements from the validation/test set, search the codebase in the dataset for relevant code snippets, and return the top 100 code snippets for pre-retrieval stage. After refined ranking stage, we return the top 20 code snippets to evaluate the value of MRR.

4.2 RQ1: The effectiveness of CUTE

In this section, we evaluate the performance of pre-trained models fine-tuned with both traditional methods and the proposed CUTE in code search tasks. The traditional approach typically represents queries and candidate code snippets as vectors using pre-trained models, then adjusting the distances between vectors through ML for fine-tuning. In contrast, our proposed CUTE considers the collaborative features between code and query. It captures and represents more complete semantic features using a three-stage CFR. Finally, the pre-trained model is also fine-tuned by ML. We select three advanced pre-trained models for programming languages, RoBERTa(Code), CodeBERT, and GraphCodeBERT. We compare the performance of the traditional fine-tuning method with the proposed CUTE on code search tasks across six programming languages, using MRR as the evaluation metric. The results are presented in Table 2.

Table 2. Effectiveness of CUTE compared with the traditional fine-tuning method.

Model		Ruby	Python	Java	PHP	Go	JS	Average
RoBERTa(Code)	traditional	0.625	0.620	0.654	0.588	0.865	0.575	0.655
	CUTE	**0.645**	**0.656**	**0.682**	**0.615**	**0.879**	**0.591**	**0.678↑3.5%**
CodeBERT	traditional	0.633	0.642	0.674	0.619	0.878	0.595	0.674
	CUTE	**0.683**	**0.671**	**0.695**	**0.635**	**0.885**	**0.634**	**0.701↑4.0%**
GraphCodeBERT	traditional	0.709	0.690	0.689	0.648	0.896	0.637	0.712
	CUTE	**0.736**	**0.715**	**0.705**	**0.685**	**0.905**	**0.654**	**0.733↑2.9%**

The results indicate that, compared to traditional fine-tuning method, the proposed CUTE fine-tuning approach leads to performance improvements of 3.5%, 4.0%, and 2.9% for the three pre-trained models, RoBERTa(Code), CodeBERT, and Graph-CodeBERT, respectively. The results demonstrate the superiority of CUTE in fine-tuning models and further highlight the positive impact of collaborative features between query and code matching.

4.3 RQ2: Ablation experiments

We construct ablation experiments to verify the structural rationality of CUTE's fine-tuning and retrieval components.

Ablation Experiment for CUTE's Fine-Tuning Component. For the fine-tuning component, we remove the residual fusion (RF), collaborative representation (CR), and the generated negative samples (Hard), respectively, to get the code search performance of three pre-trained models after fine-tuning. The results are shown in Table 3. It is worth noting that if the CR is removed, the RF will also be eliminated. Therefore, in practice, removing the CR means removing both the CR and the RF (CR+RF).

Table 3. The ablation experiment results of CUTE's fine-tuning component.

Model		Ruby	Python	Java	PHP	Go	JS	Average
RoBERTa(Code)	CUTE	**0.645**	**0.656**	**0.682**	**0.615**	**0.879**	**0.591**	**0.678**
	-Hard	0.637	0.649	0.675	0.609	0.875	0.588	0.672↓**0.9%**
	-RF	0.631	0.636	0.669	0.601	0.873	0.584	0.666↓**1.8%**
	-CR-RF	0.628	0.635	0.660	0.598	0.869	0.585	0.663↓**2.2%**
CodeBERT	CUTE	**0.683**	**0.671**	**0.695**	**0.635**	**0.885**	**0.634**	**0.701**
	-Hard	0.675	0.664	0.691	0.629	0.881	0.625	0.694↓**1.0%**
	-RF	0.667	0.661	0.685	0.627	0.881	0.623	0.691↓**1.4%**
	-CR-RF	0.652	0.654	0.681	0.625	0.880	0.615	0.685↓**2.3%**
GraphCodeBERT	CUTE	**0.736**	**0.715**	**0.705**	**0.685**	**0.905**	**0.654**	**0.733**
	-Hard	0.725	0.711	0.699	0.675	0.904	0.651	0.728↓**0.7%**
	-RF	0.722	0.702	0.698	0.669	0.900	0.645	0.723↓**1.4%**
	-CR-RF	0.719	0.698	0.690	0.659	0.899	0.642	0.718↓**2.0%**

Table 3 shows that removing the RF, CR (RF+CR), and Hard, respectively, all decrease the MRR of the three pre-trained models. The results indicate that each part of CUTE's fine-tuning component plays a crucial role, confirming that the fine-tuning architecture is reasonable.

After removing the generated hard negative samples, the performance of fine-tuned model RoBERTa(Code), CodeBERT, and GraphCodeBERT decreased by 0.9%, 1.0%, and 0.7%, respectively. The results indicate that the generated hard negative samples benefit model optimization during training. After removing the RF, the code search performance of the fine-tuned models decreased by 1.8%, 1.4%, and 1.4%, respectively. The changes indicate that RF compensates for the loss of semantic features of the original code and query vectors during the network propagation process, generating code and query representations with relatively complete semantic features. After removing RF+CR, the fine-tuned models' performance decreased by 2.2%, 2.3%, and 2.0%, respectively. Furthermore, the performance decrease was even greater compared to when only RF was removed under the same conditions. The results show that the token-level collaborative feature captured by CR is beneficial for matching code and query.

Ablation Experiment for CUTE's Retrieval Component. We pre-represent the code snippets in the codebase in advance, avoiding the time-consuming of the PR stage during user retrieval. However, performing CR and RF operations on all code snippets and the given query would significantly increase the search time because the CR and RF operations in CFR require operating on the given query original vector and all candidate code original vectors. To balance search effectiveness and efficiency, the retrieval component of our proposed CUTE framework adopts a two-stage retrieval architecture consisting of pre-retrieval and refined ranking.

We compare the retrieval performance and time by removing the pre-retrieval (PRR) and refined ranking (PR) stages separately. Where the retrieval time is the total elapsed time in milliseconds for recommending relevant code for 1000 query statements. Since the retrieval time is only affected by the retrieval operations and not the language type of the retrieved code base, we thus only use the Java sub-dataset of CodeSearchNet for our experiments, and the results are shown in Table 4.

Table 4. The ablation results of CUTE's retrieval component.

Model	RoBERTa(Code)			CodeBERT			GraphCodeBERT		
	CUTE	RR	PRR	CUTE	RR	PRR	CUTE	RR	PRR
Time (ms)	**7249**	5178	8802	**6802**	5233	9053	**7275**	5389	9055
		↓28.6%	**↑21.4%**		**↓23.1%**	**↑33.1%**		**↓25.9%**	**↑24.5%**
MRR	**0.682**	0.660	0.686	**0.695**	0.681	0.699	**0.705**	0.690	0.707
		↓3.2%	**↑0.6%**		**↓2.0%**	**↑0.6%**		**↓2.1%**	**↑0.3%**

The results demonstrate that our proposed two-stage retrieval architecture significantly reduces retrieval time while maintaining relative high performance in code search. Specifically, when we eliminate the RR and conduct retrieval using original vectors of code and query after PR, the retrieval time for the three models decreases by an average of 25.9%. However, due to the lack of collaborative features between the code and the query, the performance drops by an average of 2.4%. It is not cost-effective to trade such a high-performance loss for retrieval efficiency. When we eliminate PRR (calculating the collaborative features between the given code and all candidate code snippets), the average retrieval performance of the three models improves by 0.5%, and the average retrieval time increases by 26.3%. Trading retrieval time for a slight performance improvement is also inappropriate. Through the above analysis, we can conclude the superiority of the two-stage retrieval architecture of the proposed CUTE.

4.4 RQ3: The implementation of CFR

In CFR, the CR stage involves a series of row & column operations on the collaborative matrix to obtain collaborative vectors. The RF stage combines the

original vectors after PR stage with the collaborative vectors. In this section, we aim to experimentally determine the most appropriate implementation approach for them.

The Implementation of Collaborative Representation (CR). From the above mentioned, CR stage conducts a series of row & column operations on the collaborative matrix to generate the weight scores representing the association strength between code and query, where the matching strength of each token in code/query with the whole query/code is obtained by pooling operations. In this section, we compare the effects of the max-pooling operation (Max) and the average-pooling operation (Avg) of CUTE on the code search performance.

Table 5. Performance of different pooling operations.

Laguage	RoBERTa(Code)		CodeBERT		GraphCodeBERT	
	Avg	Max	Avg	Max	Avg	Max
Ruby	0.639	**0.645**	0.677	**0.683**	**0.737**	**0.736**
Python	0.654	**0.656**	0.668	**0.671**	0.708	**0.715**
Java	**0.683**	0.682	0.685	**0.695**	0.702	**0.705**
PHP	0.614	**0.615**	**0.635**	**0.635**	0.683	**0.685**
Go	0.866	**0.879**	0.880	**0.885**	0.901	**0.905**
JS	**0.593**	0.591	0.629	**0.634**	0.649	**0.654**
Average	0.675	**0.678**	0.696	**0.701**	0.730	**0.733**

In Table 5, we can find that the max-pooling outperforms the average-pooling operation. Before the pooling operation, a particular code/query token has matching scores with each token in the query/code, we need to get the matching score of this token and the whole query/code through a pooling operation. The difference between choosing max-pooling and average-pooling is to choose the highest or the average of matching scores as the weight to measure the association strength. The results in Table 5 show that the highest score is more suitable to be the weight. We conjecture that max-pooling works best because the highest score indicates the highest level of association, which is the most appropriate indicator for semantic association strength.

The Implementation of Residual Fusion (RF). In CFR, the RF stage involves the fusion operation between the original and collaborative vectors. The fusion operation can be performed using concatenation (Con), add (Add), or average (Avg). Table 6 presents the impact of these three operations on code search performance.

The results in Table 6 demonstrate that the concatenation operation yields the best average performance across six programming languages and three pre-trained models. The add and average operations exhibit similar performance,

Table 6. Performance of different fusion operations.

Language	RoBERTa(Code)			CodeBERT			GraphCodeBERT		
	Avg	Add	Con	Avg	Add	Con	Avg	Add	Con
Ruby	0.642	0.641	**0.645**	0.680	**0.684**	0.683	0.731	0.734	**0.736**
Python	0.650	0.651	**0.656**	0.670	0.669	**0.671**	0.714	0.713	**0.715**
Java	**0.682**	0.681	**0.682**	0.693	0.693	**0.695**	**0.708**	0.707	0.705
PHP	**0.616**	0.615	0.615	0.632	0.631	**0.635**	0.677	0.680	**0.685**
Go	0.877	**0.881**	0.879	0.882	0.884	**0.885**	0.903	0.904	**0.905**
JS	0.588	0.590	**0.591**	0.628	0.631	**0.634**	0.651	0.650	**0.654**
Average	0.676	0.677	**0.678**	0.698	0.699	**0.701**	0.731	0.731	**0.733**

lower than the concatenation operation. Based on the results, we adopt the concatenation operation as the specific implementation for CFR.

4.5 RQ4: Determining the Optimal Parameters for Augmentation

The augmentation operation has two parameters: interpolation coefficient λ and augmentaion count M. In this section, we aim to experimentally determine the optimal parameters of CUTE. We conduct experiments using the Java dataset of CodeSearchNet.

Determine the Interpolation Coefficient. Since the larger the value of λ, the more similar the hard negative samples generated by linear interpolation will be to the original samples, we take a value of λ as 0.7, 0.75, 0.8, 0.85, 0.9, 0.95 to determine the optimal parameter values. The experimental results are shown in Table 7.

Table 7. The performance of CUTE with different interpolation coefficient.

λ	RoBERTa(Code)	CodeBERT	GraphCodeBERT
0.75	0.676	0.693	0.699
0.8	0.680	**0.695**	0.701
0.85	**0.682**	**0.695**	0.705
0.9	0.681	0.692	**0.706**
0.95	0.673	0.691	0.697

The results indicate that when λ is set to 0.85, the generated hard negative samples are most beneficial for fine-tuning the pre-trained models. Specifically, when λ is less than 0.85, the generated hard negative samples have limited

similarity to the original samples, resulting in a suboptimal utilization of hard negative samples for model optimization. On the other hand, when λ is greater than 0.85, the fine-tuning performance gradually declines. We speculate that this is due to the excessively high similarity between the generated hard negative samples and the original samples, making it challenging for the model to distinguish features and reducing its feature extraction capability.

Determine the Augmentaion Count. To enhance the fine-tuning effect of CUTE, we augmented each code in the mini-batch M times using vector-level data augmentation techniques. These generated hard negative samples were then included in the mini-batch for training. We experimented with different values of M to determine the optimal value, the results are presented in Table 8.

Table 8. The performance of CUTE with different augmentaion count.

M	RoBERTa(Code)	CodeBERT	GraphCodeBERT
3	0.677	0.692	0.700
5	0.682	0.695	0.705
10	**0.683**	0.695	**0.706**
15	**0.683**	**0.696**	**0.706**

The results demonstrate that the higher the augmentation count M, the better the fine-tuning performance. By analyzing the results in Table 8, we observe that the fine-tuning performance improvement becomes increasingly slow and even stagnates when M exceeds 5. Additionally, more hard negative samples would increase the resource consumption during training. Therefore, we determine M as 5 to generate hard negative samples to balance effectiveness and efficiency.

5 Conclusions

In this paper, we propose CUTE, a collaborative fusion representation-based fine-tuning and retrieval framework for code search tasks. Specifically, we introduce a three-stage collaborative fusion representation method in the fine-tuning component, which captures the token-level collaborative features between code and query, significantly improving representation accuracy. Moreover, we also utilize vector-level augmentation techniques to generate hard negative samples, further enhancing the fine-tuning effect. In the retrieval component, we propose a two-stage retrieval architecture, including pre-retrieval and refined ranking, ensuring retrieval quality while greatly reducing retrieval time. We conducted extensive experiments on CodeSearchNet comprising six programming languages

with three pre-trained models, determining the specific implementation and optimal parameters for CUTE. Experiment validation results demonstrate the proposed CUTE's superior fine-tuning and retrieval capabilities. For future work, we plan to delve deeper into the collaborative features between code and query, aiming to achieve higher performance code search.

References

1. Liu, C., Xia, X., Lo, D., Gao, C., Yang, X., Grundy, J.: Opportunities and challenges in code search tools. ACM Comput. Surv. (CSUR) **54**(9), 1–40 (2021)
2. Linstead, E., Rigor, P., Bajracharya, S., Lopes, C., Baldi, P.: Mining internet-scale software repositories. In: Advances in Neural Information Processing Systems, vol. 20 (2007)
3. Lu, M., Sun, X., Wang, S., Lo, D., Duan, Y.: Query expansion via wordnet for effective code search. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 545–549. IEEE (2015)
4. Lv, F., Zhang, H., Lou, J.G., Wang, S., Zhang, D., Zhao, J.: Codehow: effective code search based on api understanding and extended boolean model (e). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 260–270. IEEE (2015)
5. Biggerstaff, T.J., Mitbander, B.G., Webster, D.E.: Program understanding and the concept assignment problem. Commun. ACM **37**(5), 72–82 (1994)
6. Bellet, A., Habrard, A., Sebban, M.: A survey on metric learning for feature vectors and structured data. arXiv preprint arXiv:1306.6709 (2013)
7. Gu, X., Zhang, H., Kim, S.: Deep code search. In: Proceedings of the 40th International Conference on Software Engineering, pp. 933–944 (2018)
8. Cambronero, J., Li, H., Kim, S., Sen, K., Chandra, S.: When deep learning met code search. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 964–974 (2019)
9. Fang, S., Tan, Y.S., Zhang, T., Liu, Y.: Self-attention networks for code search. Inf. Softw. Technol. **134**, 106542 (2021)
10. Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., Yin, J.: Unixcoder: unified cross-modal pre-training for code representation. arXiv preprint arXiv:2203.03850 (2022)
11. Feng, Z., et al.: Codebert: a pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155 (2020)
12. Guo, D., et al.: Graphcodebert: pre-training code representations with data flow. arXiv preprint arXiv:2009.08366 (2020)
13. Liu, S., Wu, B., Xie, X., Meng, G., Liu, Y.: Contrabert: enhancing code pre-trained models via contrastive learning. arXiv preprint arXiv:2301.09072 (2023)
14. Niu, C., Li, C., Luo, B., Ng, V.: Deep learning meets software engineering: a survey on pre-trained models of source code. arXiv preprint arXiv:2205.11739 (2022)
15. Ge, W.: Deep metric learning with hierarchical triplet loss. In: Proceedings of the European Conference on Computer Vision (ECCV), pp. 269–285 (2018)
16. Robinson, J., Chuang, C.Y., Sra, S., Jegelka, S.: Contrastive learning with hard negative samples. arXiv preprint arXiv:2010.04592 (2020)
17. Harwood, B., Kumar BG, V., Carneiro, G., Reid, I., Drummond, T.: Smart mining for deep metric learning. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 2821–2829 (2017)

18. Ling, X., et al.: Deep graph matching and searching for semantic code retrieval. ACM Trans. Knowl. Disc. Data (TKDD) **15**(5), 1–21 (2021)
19. Wang, X., et al.: Syncobert: syntax-guided multi-modal contrastive pre-training for code representation. arXiv preprint arXiv:2108.04556 (2021)
20. Suh, Y., Han, B., Kim, W., Lee, K.M.: Stochastic class-based hard example mining for deep metric learning. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 7251–7259 (2019)
21. Yu, Z., Yu, J., Xiang, C., Fan, J., Tao, D.: Beyond bilinear: Generalized multimodal factorized high-order pooling for visual question answering. IEEE Trans. Neural Netw. Learn. Syst. **29**(12), 5947–5959 (2018)
22. Li, L., Dong, R., Chen, L.: Context-aware co-attention neural network for service recommendations. In: 2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW), pp. 201–208. IEEE (2019)
23. Li, B., Sun, Z., Li, Q., Wu, Y., Hu, A.: Group-wise deep object co-segmentation with co-attention recurrent neural network. In: Proceedings of the IEEE/CVF International Conference on Computer Vision, pp. 8519–8528 (2019)
24. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)
25. Goodfellow, I., et al.: Generative adversarial networks. Commun. ACM **63**(11), 139–144 (2020)
26. Wang, H., Zhang, J., Xia, Y., Bian, J., Zhang, C., Liu, T.Y.: Cosea: convolutional code search with layer-wise attention. arXiv preprint arXiv:2010.09520 (2020)
27. Gu, J., Chen, Z., Monperrus, M.: Multimodal representation for neural code search. In: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 483–494. IEEE (2021)
28. Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X.: A novel neural source code representation based on abstract syntax tree. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 783–794. IEEE (2019)
29. Ling, C., Lin, Z., Zou, Y., Xie, B.: Adaptive deep code search. In: Proceedings of the 28th International Conference on Program Comprehension, pp. 48–59 (2020)
30. Chai, Y., Zhang, H., Shen, B., Gu, X.: Cross-domain deep code search with meta learning. In: Proceedings of the 44th International Conference on Software Engineering, pp. 487–498 (2022)
31. Husain, H., Wu, H.H., Gazit, T., Allamanis, M., Brockschmidt, M.: Codesearchnet challenge: evaluating the state of semantic code search. arXiv preprint arXiv:1909.09436 (2019)
32. Tipirneni, S., Zhu, M., Reddy, C.K.: Structcoder: structure-aware transformer for code generation. arXiv preprint arXiv:2206.05239 (2022)
33. Ma, H., Li, Y., Ji, X., Han, J., Li, Z.: Mscoa: multi-step co-attention model for multi-label classification. IEEE Access **7**, 109635–109645 (2019)
34. Zhang, P., Zhu, H., Xiong, T., Yang, Y.: Co-attention network and low-rank bilinear pooling for aspect based sentiment analysis. In: ICASSP 2019–2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 6725–6729. IEEE (2019)
35. Shuai, J., Xu, L., Liu, C., Yan, M., Xia, X., Lei, Y.: Improving code search with co-attentive representation learning. In: Proceedings of the 28th International Conference on Program Comprehension, pp. 196–207 (2020)
36. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)