The Journal of Systems and Software 208 (2024) 111896

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Hierarchical features extraction and data reorganization for code search ☆

Fan Zhang, Manman Peng *, Yuanyuan Shen, Qiang Wu

College of Computer Science and Electronic Engineering, Hunan University, Changsha, 410082, China

ARTICLE INFO

Keywords:
AI in SE
Code search
Transformer-based architecture

ABSTRACT

According to a natural language query, code search aims to retrieve relevant code snippets from a codebase. Recent works mainly rely on transformer-based pretraining models to measure the matching degree of queries and codes. Compared with works that rely on earlier deep learning methods, such as LSTM and Attention, they can significantly improve the performance of code search tasks. However, the different layers of the transformer-based models have different features that are intuitive and efficient for understanding the semantics of codes and queries but are rarely considered. Moreover, existing methods do not consider further increasing the amount of training data during training to improve the model's performance.

Toward this end, we propose a novel method called HFEDR, which utilizes the hierarchical features of transformer-based models and reorganizes original training data during a training phase. Specifically, we first extract high-level and low-level features of queries and codes from the higher and lower layers of GraphCodeBERT, respectively, achieving multi-view and comprehensive semantic representation. After that, we organize the original training data into hierarchical-uncorrelated feature pairs and then reorganize them into hierarchical-correlated feature pairs, achieving training the model with more data. Finally, we update the model's parameters using a contrastive training method. We conduct extensive experiments on CodeSearchNet, demonstrating the effectiveness and rationality of our proposed approach.

1. Introduction

Code search is given a query to find the most relevant code snippets from a codebase. It can help developers reuse high-quality codes to improve their development efficiency. Therefore, improving the performance of code search tasks has attracted many researchers to study (Liu et al., 2021b).

A great effort has been made to address code search issues in the past few years. Traditional information retrieval (IR) based methods (Lv et al., 2015; Allamanis et al., 2015) are applied to code search tasks via text matching between queries and codes. However, they have not sufficiently narrowed the semantic gap between queries and codes. With the flourishing of deep learning technology, more new technologies are introduced to code search tasks to overcome traditional methods' limitations, such as LSTM (Graves, 2012), Attention (Vaswani et al., 2017), Pre-Train Devlin et al. (2018), Meta-Learning (Vilalta and Drissi, 2002), and Contrastive Learning (Chen et al., 2020). For example, Gu et al. (2018) proposed DeepCS, which first used a deep learning technique named LSTM to encode queries and codes as representations for code search tasks. Hu et al. (2023) proposed MESN-CS, which employed self-attention to learn the word-level representations and code unit-level representations for code search. Feng et al. (2020) introduced CodeBERT, boosting code search performance significantly with the help of pre-training techniques. Guo et al. (2021) proposed GraphCodeBERT, which considered the inherent structure of codes, such as using data flow to pre-train a model for better retrieval. In addition, researchers take advantage of meta-learning and contrastive learning to solve the problem of code search tasks with scarce data (Chai et al., 2022; Sun et al., 2022).

Overall, most existing deep learning-based approaches mainly adopt one of the deep learning paradigms to perform code search tasks. They encode queries and codes to feature vectors, make these feature vectors in the same vector space, and then employ a similarity function to measure whether they are close. Among them, the recent pre-training models based on transformer architecture, such as CodeBERT and GraphCodeBERT, have significantly improved the performance of code search tasks. However, they neglect to utilize hierarchical features extracted from queries and codes, which are extracted by transformer-based pre-trained models. For instance, if using CodeBERT or GraphCodeBERT to encode queries and codes, 12 layers of features will be output. Existing works have ignored that the features

☆ Editor: Xiao Liu.

* Corresponding author.
E-mail addresses: fanzhang@hnu.edu.cn (F. Zhang), pengmanman@hnu.edu.cn (M. Peng), shenyuanyuan@hnu.edu.cn (Y. Shen), wuqiang@hnu.edu.cn (Q. Wu).

https://doi.org/10.1016/j.jss.2023.111896
Received 2 April 2023; Received in revised form 7 September 2023; Accepted 2 November 2023
Available online 9 November 2023

in the lower layers capture low-level representations, and higher layers capture high-level representations (Tenney et al., 2019; Peters et al., 2018; Vig, 2019). In addition, although the existing works have prepared a large amount of data to train a model, they neglect to further increase the amount of training data during training to train the model better. For example, for a batch of original training data $\{\langle query,\ code,\ label\rangle_i\}_{i=1}^{B}$, where the B is the batch size, the *label* is 1 or 0, which means that $\langle query,\ code,\ label\rangle$ is a positive sample pair(matched) or a negative sample pair(unmatched). They directly feed these sample pairs into a model for training without considering constructing additional sample pairs during training.

To address the above issues, we propose a novel method named HFEDR (Hierarchical Features Extraction and Data Reorganization), which captures the complex relationships between queries and codes by exploiting hierarchical features and reorganizing original training data during training. To be specific, Firstly, in terms of the hierarchical features, we use GraphCodeBERT, a state-of-the-art code pre-training model with the transformer architecture, to extract high-level and low-level features from queries and codes to achieve a multi-view and comprehensive semantic representation. Since we are inspired by the model Hit, which uses the same principle to perform better on video-text retrieval tasks, and according to our experiments, we choose the first three layers and the last three layers as high-level and low-level features, respectively. Secondly, in terms of reorganizing the training data during training, we first organize a batch of original training data into hierarchical-uncorrelated feature pairs based on hierarchical features. We then reorganize them into hierarchical-correlated feature pairs, enabling the model to use additional training data better to capture the semantic relationships between queries and codes. We detail hierarchical correlated and uncorrelated feature pairs in Section 3.2. Finally, in a training phase, we use a contrastive loss function named CoSENT to ensure that similarities in positive feature pairs are greater than in negative samples. In the retrieval phase, to reduce storage space and achieve better and faster retrieval, we use a BERT-whitening operation to reduce the dimension and anisotropy of vectors.

To demonstrate the effectiveness and rationality of HFEDR, we conduct extensive experiments on CodeSearchNet, which includes six programming languages: Ruby, JavaScript, Go, Python, Java, and PHP. We use the same setting as Husain et al. (2019) to pre-process the dataset. It requires retrieving an answer for a query from 1000 candidates. We compare HFEDR with four state-of-the-art deep learning-based baseline models. Experimental results show that HFEDR outperforms state-of-the-art models with the evaluation metrics of MRR. Further micro-level analyses demonstrate how components of HFEDR affect the results, how predefined hyper-parameters affect our methods, and how to choose suitable layers to balance the retrieval performance and computing resources.

The main contributions of this work are summarized as follows:

1. To the best of our knowledge, this is the first work that attempts to perform the code search by investigating the hierarchical features of the transformer-based bimodal pre-training model, which can achieve multi-view and comprehensive semantic representation of queries and codes.
2. We propose a code search method HFEDR based on hierarchical features and reorganized training data. Specifically, HFEDR first extracts hierarchical features from the original training data and organizes them into hierarchical-uncorrelated feature pairs. Then HFEDR reorganizes each layer's uncorrelated feature pairs into correlated feature pairs, thereby increasing the amount of training data to improve the model's performance on code search tasks. In addition, the BERT-whitening operation is also incorporated in the code retrieval stage to obtain better performance and save storage space.
3. Extensive experiments are performed on a large-scale dataset to demonstrate the effectiveness and rationality of our method. Meanwhile, the codes are released to facilitate research communities.[1]

2. Related works

In this section, we present the related works of code search and then introduce some background knowledge of HFEDR, including recent research on code search, pre-training techniques, and contrastive learning.

2.1. Code search

For clarity, we divide major technology used in code search studies into two main branches, information retrieval (IR) based and deep learning (DL) based approaches.

In early studies, information retrieval approaches were applied to code search tasks based on text matching between queries and codes. For instance, Lv et al. (2015) proposed CodeHow, which measures the relevancy between a query and candidate codes based on TF-IDF (term frequency-inverse document frequency) (Wu et al., 2008). Allamanis et al. (2015) proposed building probabilistic models for computing the relevancy score between short natural language utterances and source code snippets.

Deep learning approaches have recently drawn great attention in code search tasks. It attempts to bridge the semantic gap between queries and codes by neural networks. With the development of deep learning, more new technologies have been introduced to code search tasks, such as LSTM (Graves, 2012), Pre-train (Devlin et al., 2018), Meta learning (Vilalta and Drissi, 2002), and Contrastive learning (Chen et al., 2020). Gu et al. (2018) proposed DeepCS, which first uses deep learning techniques for code search tasks. It encodes queries and codes to embeddings by LSTM, then makes them in the same semantic space to measure whether they are matched. Cheng and Kuang (2022) proposed CSRS, which combines attention mechanisms and information retrieval methods to improve performance. Xu et al. (2021) proposed a two-stage attention-based model for code search tasks named TabCS, which utilizes the attention and co-attention mechanism (Lu et al., 2016) to learn better query and code representation. Hu et al. (2023) propose a novel self-attention model called MESN-CS, which considers word-level attention and code unit-level attention for code search. Feng et al. (2020) proposed CodeBERT, which significantly boosts code search performance by using pre-training techniques. Guo et al. (2021) present GraphCodeBERT, which considers the inherent structure of codes, such as using data flow in the pre-training stage. It has achieved state-of-the-art performance on code search tasks. In addition, Chai et al. (2022) proposed CDCS, which employs a meta learning algorithm for code search tasks with relatively scarce and expensive data, such as SQL and Solidity. Sun et al. (2022) proposed a novel context-aware code translation technique that translates code snippets into natural language descriptions.

Although existing deep learning-based methods have achieved remarkable success in code search tasks, they ignore using the extracted hierarchical features and increasing the amount of training data during training. Therefore, we can further improve previous works to more fully capture the complicated relationships between queries and codes for better retrieval.

[1] https://github.com/cocola00232/HFEDR

2.2. Transformer-based architecture

We use a pre-trained model with transformer-based architecture named GraphCodeBERT (Guo et al., 2021) as an encoder to extract hierarchical features from queries and codes. Hence, we will introduce the concepts of its pre-trained technology and transformer-based architecture.

In the field of natural language processing, pre-trained models such as BERT (Devlin et al., 2018) have shown remarkable success. Similarly, many researchers proposed pre-trained models such as CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al., 2021) to improve the performance of programming language tasks. They mainly follow the BERT and use a transformer-based neural architecture as the model backbone.

The transformer-based architecture can be considered that its output has 12 layers of hidden states, also known as features. As studied in Tenney et al. (2019), Peters et al. (2018), Vig (2019), the hidden states in the lower layers capture low-level representations, and the higher layers capture high-level representations. However, most existing works do not fully utilize the extracted hierarchical features from queries and codes. Hence, we use hierarchical features to enhance sample pairs for better training.

2.3. Contrastive training

In this subsection, we introduce the concept of contrastive training and its application.

Contrastive training is a method in contrastive learning which has been widely applied in various domains, such as CV and NLP tasks (Chen et al., 2020, 2021; Gao et al., 2021). Its goal is to learn an embedding space where similar sample pairs remain close together while different sample pairs are far apart. For instance, Chen et al. (2020) presented SimCLR, a simple framework for contrastive learning of visual representations. It learns representations for visual inputs by maximizing agreement between differently augmented views of the same sample via a contrastive loss in the latent space. Gao et al. (2021) presented SimCSE: a simple contrastive learning framework for sentence embeddings. SimCSE treats dropout as data augmentation for text sequences in an unsupervised approach, and employs annotated pairs from natural language inference datasets to construct positive and negative sample pairs in a supervised approach. Liu et al. (2021a) propose a hierarchical transformer for video-text retrieval named HiT, which performs hierarchical cross-modal contrastive matching in both feature-level and semantic-level. Recently, contrastive learning has been used for code search tasks. For instance, Bui et al. (2021) proposed Corder, a self-supervised contrastive learning framework for the source code model. It is used to solve the code retrieval problem that does not have sufficient labeled data.

Constructing negative sample pairs within a batch is one of the methods for constructing training samples in contrastive training. They inspire the idea of reorganizing hierarchical-correlated feature pairs in our work. We believe it is a promising method for improving the performance of the code search task.

3. Approach

The framework of our proposed method HFEDR is shown in Fig. 1, which includes extracting hierarchical features, organizing hierarchical-uncorrelated feature pairs, reorganizing hierarchical-correlated feature pairs, and contrastive training. Specifically, HFEDR first extracts high-level and low-level features from queries and codes via GraphCodeBERT. Then HFEDR organizes hierarchical-uncorrelated feature pairs and reorganizes hierarchical-correlated feature pairs in two stages. Finally, HFEDR utilizes contrastive training to optimize the model's parameters, ensuring that the similarity of positive feature pairs is greater than negative feature pairs.

3.1. Extracting hierarchical features

To achieve multi-view and comprehensive semantic representation, we use the state-of-the-art code embedding method GraphCodeBERT to extract the hierarchical features of queries and codes.

GraphCodeBERT is a pre-trained model based on transformer architecture, trained by natural language and programming language, and achieves state-of-the-art performance on code retrieval tasks (Guo et al., 2021). Using GraphCodeBERT to encode queries and codes, 12 layers of hidden states, also known as features, will be output. Recent studies have shown that hidden states in lower layers in transformer-based architectures capture low-level features that can describe basic information. Hidden states in higher layers capture high-level features with more complex meanings (Tenney et al., 2019; Peters et al., 2018; Vig, 2019). Therefore, the low-level and high-level features can be combined to achieve multi-view and comprehensive semantic representation. In addition, we are also inspired by HIT, which uses the hidden states of the first two layers and the last two layers as low-level and high-level features, respectively, and has achieved better results in video-text retrieval tasks (Liu et al., 2021a).

Thus, for each $\langle query,\ code,\ label\rangle$ sample pair in the training corpus, where the *label* is 1 or 0, which means that $\langle query,\ code\rangle$ is a positive sample pair(matched) or a negative sample pair(unmatched). We feed the *query* and *code* into GraphCodeBERT, extracting the hidden states of the first three layers and the last three layers from the output as low-level and high-level features, respectively. The reason for choosing these layers is based on our experiments and HIT options (Liu et al., 2021a). That is, we use six layers of features to represent the semantics of a code or a query.

3.2. Data reorganization

To make the model better establish the semantic relationships between queries and codes, we reorganize the original training data to increase the amount of training data in two stages. In the first stage, we obtain hierarchical-uncorrelated feature pairs based on the hierarchical features extracted from the original training data. In the second stage, we reorganize each layer's uncorrelated feature pairs into multiple sets of hierarchical-correlated feature pairs.

In the first stage, we organize hierarchical-uncorrelated feature pairs, which have less correlation between the positive and negative feature pairs in each layer. Specifically, the original training corpus consists of positive and negative sample pairs. The positive sample pairs are matched queries and codes, and the negative sample pairs are randomly matched by these queries and codes. The number of positive sample pairs equals the number of negative sample pairs. We shuffle the original corpus's positive and negative sample pairs during loading data. After that, for the training data $\{\langle query,\ code,\ label\rangle_i\}_{i=1}^{B}$ in a batch, their positive and negative sample pairs can be unordered and have less correlation. We input them into GraphCodeBERT and obtain hierarchical feature pairs $\{\{\langle query_l,\ code_l,\ label\rangle_i\}_{i=1}^{B}\}_{l=1}^{6}$, which is still unordered and uncorrelated. Thus, these can be regarded as hierarchical-uncorrelated feature pairs. More formally, for feature pairs $\{\langle query_l,\ code_l,\ label\rangle_i\}_{i=1}^{B}$ in each layer, the query and code features of their positive feature pairs $\{<query_l,\ code_l,\ 1>_i\}_{i=1}^{M}$ are denoted as $\{pos\_query_{li}\}_{i=1}^{M}$ and $\{pos\_code_{li}\}_{i=1}^{M}$, respectively. The query and code features of their negative feature pairs $\{<query_l,\ code_l,\ 0>_i\}_{i=1}^{N}$ are denoted as $\{neg\_query_{li}\}_{i=1}^{N}$ and $\{neg\_code_{li}\}_{i=1}^{N}$, respectively. The M and N are the numbers of positive and negative feature pairs in a layer, respectively, and $M+N=B$. They satisfy the following requirements:

$$\begin{cases} |\{pos\_query_{li}\}_{i=1}^{M} \cap \{neg\_query_{li}\}_{i=1}^{N}| \ll B \\ |\{pos\_code_{li}\}_{i=1}^{M} \cap \{neg\_code_{li}\}_{i=1}^{N}| \ll B \end{cases}$$

For example, as shown in Stage 1 of Fig. 1, the positive feature pairs of the 12th layer are $\langle Q9, C9\rangle$ and $\langle Q86, C86\rangle$. The negative feature pairs are $\langle Q21, C67\rangle$ and $\langle Q2, C10\rangle$. The batch size is 4. The number

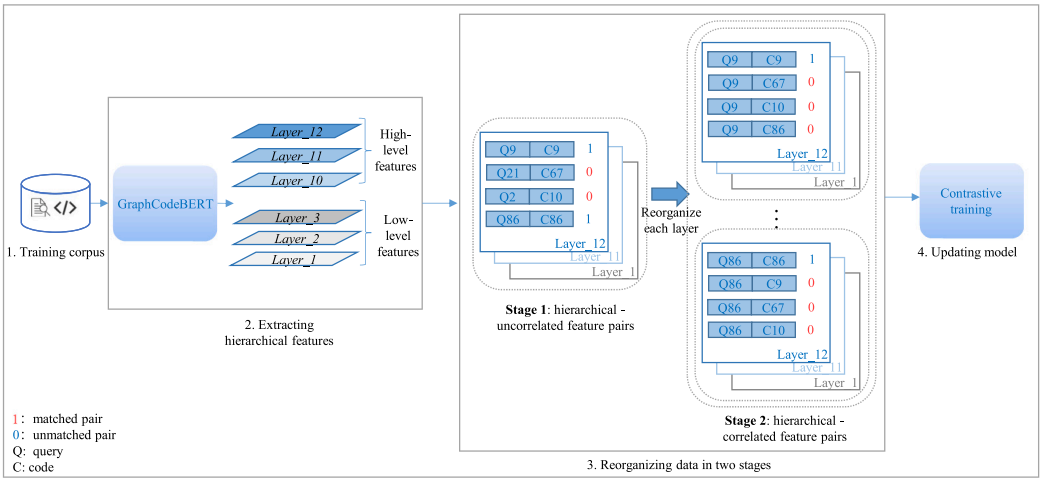


Fig. 1. The graphical representation of our proposed HFEDR framework. HFEDR first uses GraphCodeBERT to extract the hierarchical features of the sample pairs in the training corpus. Layers 10, 11 and 12 are regarded as high-level features and layers 1, 2 and 3 as low-level features. Then, to increase the amount of training data during training, HFEDR uses these hierarchical features to obtain hierarchical-uncorrelated feature pairs in the first stage; then reorganizes the uncorrelated feature pairs of each layer into hierarchical-correlated feature pairs in the second stage. Finally, HFEDR uses contrastive training with these feature pairs to update the parameters of the model.

of elements in the intersection of $\{Q9, Q86\}$ and $\{Q21, Q2\}$ equals 0, which is much smaller than the batch size, and so are $\{C9, C86\}$ and $\{C67, C10\}$. Therefore, we treat these feature pairs in the 12th layer as uncorrelated feature pairs in the layer; and such feature pairs in all layers as hierarchical-uncorrelated feature pairs.

Intuitively, HFEDR organizes hierarchical-uncorrelated training data, which can better complement each other with the reorganized hierarchical-correlated feature pairs in training time, enabling the model to build query and code relationships from uncorrelated and correlated dataset knowledge. Moreover, HFEDR can use more feature data better to learn the semantic relationships between queries and codes.

In the second stage, we reorganize the hierarchical-uncorrelated feature pairs $\{\{< query_l,\ code_l,\ label >_i\}_{i=1}^{B}\}_{l=1}^{6}$ to obtain hierarchical-correlated feature pairs. It means the positive and negative feature pairs have the same $query_l$ but different $code_l$ in each layer.

The main idea of constructing these feature pairs is that for all uncorrelated feature pairs $\{< query_l,\ code_l,\ label >_i\}_{i=1}^{B}$ in each layer, HFEDR uses each matched query feature and code feature to construct positive feature pairs. Then the query feature is paired with other unmatched code features in the same layer as negative feature pairs. Specifically, for each sample feature pair $< query_l,\ code_l,\ label >$, if it is a positive sample feature pair, then pair its query feature $query_l$ with the remaining $\{code_{li}\}_{i=1}^{B-1}$ to construct negative feature pairs. If not, new feature pairs are not reorganized.

For example, as shown in Stages 1 and 2 of Fig. 1, for the positive sample feature pair $\langle Q9,\ C9\rangle$ at layer 12 in Stage 1, $Q9$ is paired with other unmatched code features at layer 12, such as $C67$, $C10$ and $C86$, to construct negative feature pairs. For a negative sample pair $\langle Q21,\ C67\rangle$, $Q21$ is not paired with the remaining code features to build negative samples. For each sample feature pair in each layer, the above operation is followed.

Constructing positive and negative feature pairs within each layer is inspired by constructing negative sample pairs within a batch in contrastive training (Gao et al., 2021). Although this method makes the number of negative samples greater than the number of positive samples, training a model in this way yields better results for some tasks, such as sentence embeddings. Therefore, further constructing positive and negative feature pairs at each layer may achieve good results in code search tasks.

3.3. Contrastive training

Based on previous efforts, we convert sample pairs in the training corpus into hierarchical-correlated and hierarchical-uncorrelated feature pairs. We utilize these two types of hierarchical feature pairs and use a loss function to optimize the model's parameters.

We use a contrastive loss function named CoSENT (Jianlin, 2022) to ensure that similarities in positive sample pairs are greater than in negative samples. CoSENT has been shown to have faster convergence and better results on semantic similarity tasks than sentence vector schemes, such as InferSent (Conneau et al., 2017) and Sentence-BERT (Reimers and Gurevych, 2019), which employ other loss functions. Specifically, we compute the loss of feature pairs in each layer and then accumulate these losses to update the parameters of the model. Use *Pos* and *Neg* to denote the set of positive feature pairs and negative feature pairs in the same layer, respectively. CoSENT aims to let positive feature pair $\langle i,\ j\rangle \in Pos$ and negative feature pair $\langle k,\ l\rangle \in Neg$ achieve:

$$cos(q_i,\ c_j) > cos(q_k,\ c_l)\,, \tag{1}$$

where q_i and q_k are the features of different queries, c_j and c_l are the features of different codes. CoSENT employs cosine similarity to measure the semantic gap between queries and codes.

In order to achieve the goal of function (1), the formula for calculating the loss of a layer is:

$$loss\_layer = \log\left(1 + \sum_{(i,j)\in Pos,(k,l)\in Neg} e^{\tau(\cos(q_i,c_j)-\cos(q_k,c_l))}\right), \tag{2}$$

where τ is a hyperparameter.

After calculating the losses of all layers, we accumulate these losses as follows:

$$loss\_all = \sum_{i\in\{1,2,3,10,11,12\}} loss\_layer_i, \tag{3}$$

then we use $loss\_all$ to update the parameters of the model.

The procedure of our contrastive training is summarized in Algorithm 1.

3.4. Code retrieval

In terms of code retrieval, we use not only the trained HFEDR model but also the BERT-whitening operation (Su et al., 2021), which can be

Algorithm 1: Training of HFEDR

Input: Dataset D^{train}, $MaxEpoch$, Encoder GraphCodeBERT
Output: Trained HFEDR model

```
for epoch ∈ MaxEpoch do
    for each D_i ∈ D^train do
        loss_all = init (0); # Initialization loss_all
        selected_hidden_states ← Encoder(D_i) ; # Encoding
          the queries and codes, then extracting the features
          of the selected layers.
        for hidden_state ∈ selected_hidden_states do
            Calculate loss_layer by Eq. (2) using hidden_state;
            loss_all += loss_layer;
            for x ∈ D_i do
                if x is a positive pair then
                    Use x's query feature to match the code
                      features of the other feature pairs in Di
                      to form a new set of training data
                      D_new;
                    Calculate loss_layer by Eq. (2) using
                      D_new;
                    loss_all += loss_layer;
        use loss_all to update the parameters of the encoder
```

used to reduce the anisotropy and the dimension of vectors so that the similarity calculation is accurate enough and reduce storage space.

Specifically, the code retrieval process with the following steps:

1. Suppose we have a natural language query Q and n codes in a codebase $C = \{C_1, C_2, C_3, \ldots, C_k, C_n\}$.
2. Let $Q\_Feature_{layer\_i}$ and $C\_Feature_{layer\_i}$ denote the hierarchical features from the query Q and the code C_k, respectively. These features are extracted by HFEDR, where $layer\_i$ is $L \in \{1, 2, 3, 10, 11, 12\}$.
3. We employ the BERT-whitening operation to process $Q\_Feature_{layer\_i}$ and $C\_Feature_{layer\_i}$, it outputs low-dimensional vector $Q\_LowFea_{layer\_i}$ and $C\_LowFea_{layer\_i}$, the dimension is reduced to 512 in our experiment.
   In the BERT-whitening operation, we concatenate $Q\_Feature_{layer\_i}$ and $C\_Feature_{layer\_i}$ as $\{X_i\}_{i=1}^{n}$, where n denotes the sum of the number of $Q\_Feature_{layer\_i}$ and $C\_Feature_{layer\_i}$. $\{X_i\}_{i=1}^{n}$ perform linear transformation as

$$\widetilde{X}_i = (X - \mu)W \tag{4}$$

   The transformation is to make the *mean* of $\{\widetilde{X}_i\}_{i=1}^{n}$ is 0, and the covariance matrix is an identity matrix. To make the *mean* equal to 0, μ can be solved as

$$\mu = \frac{1}{n}\sum_{i=1}^{N} x_i. \tag{5}$$

   In order to solve W, the covariance of the original data is denoted as

$$\Sigma = (W^{-1^T})W^{-1}, \tag{6}$$

   where Σ is a positive definite symmetric matrix, it can be solved by SVD (Singular Value Decomposition) as

$$\Sigma = U \wedge U^T, \tag{7}$$

   where U is an orthogonal matrix and $\wedge$ is a diagonal matrix, and the diagonal elements are all positive. Therefore, W can be solved as

$$W = U\sqrt{\wedge^{-1}[:, :d]}, \tag{8}$$

   where d is the reserved dimensionality.
   After the above BERT-whitening, we separate the obtained $\widetilde{X}_i$ into the low-dimensional vector $Q\_LowFea_{layer\_i}$ and the $C\_LowFea_{layer\_i}$, where $Q\_LowFea_{layer\_i}$ and $C\_LowFea_{layer\_i}$ denote low-dimensional queries vector and codes vector, respectively. They will be used to calculate the matching score.
4. The matching score of each $< Q, C >$ is calculated by

$$\text{score} = \sum_{layer\_i \in L} \cos\left(Q_{LowFea_{layer\_i}}, C_{LowFea_{layer\_i}}\right). \tag{9}$$

5. After calculating the *score* between the query and each code, the code snippet with the highest *score* in the codebase is selected as the returned result.

It is worth mentioning that we can prepare the hierarchical features of the codes before retrieval. During retrieval, we only extract the hierarchical features of a query and calculate their similarity with the prepared hierarchical features of the codes to save retrieval time.

Table 1
Data statistics about the processed dataset.

Programming language	Training examples	Dev queries	Testing queries
Ruby	58 889	4417	2279
JavaScript	149 675	16 505	6483
Go	382 296	28 483	14 291
Python	499 231	46 213	22 176
Java	546 690	30 655	26 909
PHP	629 781	52 029	28 391

4. Experimental setups

In this section, we conduct extensive experiments on the CodeSearchNet dataset to answer the following four research questions:

RQ1 How does our proposed HFEDR framework perform as compared to other state-of-the-art competitors?

RQ2 How do different components in the HFEDR framework contribute to its performance?

RQ3 How do different hyperparameters affect the performance of HFEDR?

RQ4 How do the different layers of hidden states affect our framework?

RQ5 How time efficient is the HFEDR framework compared to other methods?

RQ6 Are the components proposed in the HFEDR framework model-free?

4.1. Datasets

We experiment with a publicly accessible dataset named CodeSearchNet (Husain et al., 2019), which includes six programming languages: Ruby, JavaScript, Go, Python, Java, and PHP. For each programming language, it contains parallel data of ⟨description, code⟩ pairs, where the description represents the ground truth of the code. The statistics of the dataset are shown in Table 1. We follow Husain et al. (2019) approach to pre-process the testing set. Specifically, each pair of test data $\langle query, code\rangle$ over a fixed set of 999 distractor codes.

In addition to CodeSearchNet, we selected two additional datasets, CoSQA (Huang et al., 2021) and AdvTest (Lu et al., 2021). The former can test the performance of HFEDR in real-world scenarios, and the latter can test the generalization ability of HFEDR. CoSQA is more representative of real-world code search scenarios. It includes 20,604 labels for pairs of natural language queries and codes, each annotated by at least three human annotators. AdvTest is constructed using CodeSearchNet's Python corpus and is mainly used to evaluate the model's generalization ability. Unlike CodeSearchNet, AdvTest replaces function and variable names in the test set with special markers to prevent overfitting.

4.2. Evaluation metrics

The performance of HFEDR is measured using a popular quantitative metric named MRR (Mean Reciprocal Rank).

MRR (Craswell, 2009) is a measure to evaluate systems that return a ranked list of answers to queries. A higher MRR score indicates better performance in the code search task. Given a query, the reciprocal rank is $\frac{1}{rank}$, where the rank is the position of the correct code snippet (1, 2, 3, ..., N for N code snippets returned in a query). For multiple queries Q, the MRR is the mean of the Q reciprocal ranks. It can be calculated by

$$MRR = \frac{1}{Q}\sum_{i=1}^{Q}\frac{1}{\text{rank}_i}. \tag{10}$$

4.3. Baselines

The effectiveness of our method is verified by comparing it with several state-of-the-art methods: RoBERTa, RoBERTa(code), CodeBERT, GraphCodeBERT, CodeT5, SPT-Code, Plbart, MoCoCS, SYNCOBERT, and Unixcoder.

- **RoBERT** (Liu et al., 2019) is pre-trained on natural language corpus. It builds on BERT and modifies key hyperparameters. Meanwhile, removing the next-sentence pretraining objective and training with much larger mini-batches and learning rates.
- **RoBERTa(code)** (Feng et al., 2020) architecture is the same as RoBERTa's. The difference is that RoBERTa(code) is trained from a codebase.

- **CodeBERT** (Feng et al., 2020) is a bimodal pre-trained model for programming languages (PL) and natural language (NL). It learns general-purpose representations that can be used in NL-PL applications like code search, code clone detection, and code summarization.
- **GraphCodeBERT** (Guo et al., 2021) is a pre-trained model for programming languages that considers the inherent structure of codes. It uses data flow in a pre-training stage, which is a semantic-level structure of codes that encodes the relation of "where-the-value-comes-from" between variables.
- **CodeT5** (Wang et al., 2021a) is the first unified encoder–decoder model to support code-related understanding and generation tasks and also supports multi-task learning. CodeT5 can better capture code semantics through the proposed identifier-aware pre-training and dual-mode dual generation, which mainly benefits NL-PL tasks.
- **SPT-Code** (Niu et al., 2022) is a seq2seq pre-trained model for source code that is built with the encoder–decoder architecture and is applicable to both classification and generation tasks.
- **Plbart** (Ahmad et al., 2021) is a bidirectional and autoregressive transformer pre-trained on unlabeled data across PL and NL to learn multilingual representations applicable to a broad spectrum of PLUG applications.
- **Unixcoder** (Guo et al., 2022) is a multi-modal code pre-training model which utilizes multi-modal data, such as code comments and AST, to pre-train code representation. UniXcoder supports code-related comprehension, generation tasks, and autoregressive tasks.
- **MoCoCS** (Shi et al., 2023) is a multi-modal momentum contrastive learning method for code search. It can improve the representation of queries and codes by constructing large-scale multi-modal negative samples.
- **SYNCOBERT** (Wang et al., 2021b) is a multi-modal contrastive pre-training framework for code representation. SYNCOBERT uses two pre-trained targets to encode symbolic and syntactic information of programming languages.

4.4. Implementation details

To conduct our experiments, we use a default GraphCodeBERT tokenizer with the same vocabulary size (50265) to tokenize queries and codes. The maximum sequence length of queries and codes is both set to 128. The default batch size is set to 64. We update the parameters via AdamW (Kingma and Ba, 2014) optimizer with the learning rate 2e-5 in the training phase, which warms up in the first 5% of all steps and linearly decays. We use an automatic mixed precision mechanism to train neural networks to reduce training time and memory requirements without affecting the model's performance. All models in this paper are trained for 10 epochs. The contrastive loss function's hyperparameter τ is set to 20.

We implemented our method based on PyTorch 1.12 framework with Python 3.8, and the experiments are conducted on a computer with a GeForce RTX 2080 Ti GPU with 11 GB memory, running on Ubuntu 18.04.

5. Experimental results

In this section, we present and analyze the experimental results to answer the research questions. Then, we provide illustrative examples to demonstrate the effectiveness of HFEDR.

5.1. RQ1: Effectiveness of HFEDR

Table 2 shows the performance of HFEDR and other state-of-the-art methods. We have the following observations: (1) With the metric of MRR, HFEDR achieves 0.742, 0.739, 0.853, 0.911, 0.798, and 0.771 on Ruby, JavaScript, Go, Python, Java, and PHP, respectively. (2) Compared to GraphCodeBERT and Unixcoder, HFEDR improves MRR scores in Python, Java, and PHP by a significant margin. The result shows that our approach HFEDR obtains better performance than state-of-the-art baselines, proving our framework is effective. (3) On the CoSQA and AdvTest datasets, we can see that the performance of HFEDR exceeds other baselines, which shows that HFEDR performs well in real-world scenarios and has good generalization ability.

Result 1: HFEDR outperforms state-of-the-art models with the evaluation metrics of MRR.

5.2. RQ2: Contribution of key components

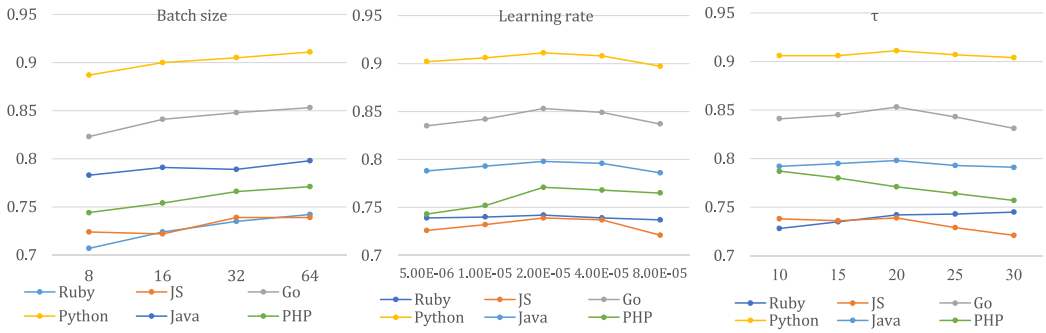
We conduct related ablation studies to understand the contribution of different components in our framework. Specifically, HFEDR compares with its variants, which remove or replace its components.

- **w/o whitening**: We removed the BERT-whitening module during the retrieval phase. The BERT-whitening module reduces the dimensionality and anisotropy of vectors for better retrieval performance.
- **w/o Uncor_Pairs**: We remove the constructing hierarchical-uncorrelated pairs module, which will show the effect of organizing hierarchical-uncorrelated feature pairs.

- **w/o Cor_Pairs**: We remove the constructing hierarchical-correlated pairs module, which will show the effect of reorganizing uncorrelated feature pairs at each layer into correlated feature pairs during training.
- **w/o Multi_Lay**:We replace the setting of extracting hierarchical features and only use the last layer of hidden states to represent sample pairs. This setting demonstrates the effect of using hierarchical features to represent codes and queries.

Table 2
Overall performance comparison among various methods.

	CoSQA	AdvTest	CodeSearchNet						
Model	Python	Python	Ruby	JS	Go	Python	Java	PHP	Avg
RoBERTa	0.603	0.183	0.625	0.606	0.82	0.809	0.666	0.658	0.697
RoBERTa(code)	0.362	0.424	0.661	0.64	0.819	0.844	0.721	0.671	0.726
CodeBERT	0.657	0.272	0.693	0.706	0.84	0.869	0.748	0.706	0.760
GraphCodeBERT	0.684	0.352	0.732	0.711	0.841	0.879	0.757	0.725	0.774
CodeT5	0.678	0.393	0.711	0.697	0.833	0.872	0.736	0.738	0.764
SPT-Code	0.634	0.386	0.689	0.673	0.816	0.845	0.701	0.706	0.742
Plbart	0.650	0.347	0.645	0.632	0.803	0.815	0.687	0.709	0.715
Unixcoder	0.701	0.413	0.734	0.721	0.836	0.884	0.782	0.739	0.783
SyncoBERT	-	0.381	0.728	0.719	0.831	0.868	0.771	0.725	0.774
MoCoCS	-	-	0.731	0.725	0.833	0.891	0.766	0.745	0.782
HFEDR	**0.752**	**0.445**	**0.742**	**0.739**	**0.853**	**0.911**	**0.798**	**0.771**	**0.802**



(a) Impact of batch size (b) Impact of Learning Rate (c) Impact of hyperparameter τ

Fig. 2. Parameter tuning in terms of MRR. The x-axis is the tuning range of batch size, learning rate, and hyperparameter τ, while the y-axis is the MRR for performance evaluation.

Table 3
Performance comparison of HFEDR and its variants.

	Ruby	JS	Go	Python	Java	PHP	AVG
w/o whitening	0.715	0.708	0.836	0.907	0.788	0.765	0.787
w/o Uncor_Pairs	0.697	0.701	0.835	0.899	0.775	0.762	0.778
w/o Cor_Pairs	0.673	0.703	0.824	0.891	0.776	0.73	0.766
w/o Multi-Lay	0.721	0.718	0.843	0.903	0.773	0.735	0.782
HFEDR	0.742	0.739	0.853	0.911	0.798	0.771	0.802

Table 3 presents the performance of HFEDR and its variants. We have the following observations: (1) HFEDR exceeds HFEDR without the BERT-whitening operation, which shows that the BERT-whitening component to reduce anisotropy is conducive to the code search task. (2) The performance of the w/o Uncor_Pairs has worsened, which proves that organizing hierarchical-uncorrelated sample features is practical. (3) Meanwhile, the performance of the w/o Cor_Pairs is meaningfully worsening, indicating that carefully reorganizing data during training is effective. (4) The w/o Multi_Lay has poor performance. It reveals the necessity of the extracting hierarchical features module proposed in our framework. (5) In addition, in descending order of improved average degree are the Cor_Pairs module, the Uncor_Pairs module, the multi-Lay module, and the BERT-whitening module. (6) In each programming language, different modules have different effects on search performance.

Result 2: Each component in the HFEDR can improve the performance of the code search, while combining these modules yields the best results.

5.3. RQ3: Impact of hyperparameters

In this subsection, we investigate the impacts of several factors to prove the robustness and effectiveness of our proposed HFEDR framework. These factors are the batch size, the learning rate, and the contrastive loss function's hyperparameter τ.

- **Impact of batch size**: Fig. 2(a) shows the performance of HFEDR with the different batch size, which is set to 8, 16, 32, and 64. It is obviously observed that the more batch size is set, the better performance we achieved.
- **Impact of Learning Rate**: Fig. 2(b) demonstrates the parameter tuning results of the learning rate, which is set from 5e-6 to 8e-5 for training. We can observe that the results remain stable in most cases. Among them, when the learning rate is 2e-5, the effect is the best.
- **Impact of hyperparameter τ**: In Fig. 2(c), we reveal the performance of HFEDR with respect to different hyperparameter τ settings. We use a step size of 5 for fine-tuning and the results obtained have fewer changes. Among them, when the τ is 20, the effect is the best.

Result 3: The performance of HFEDR is influenced by all hyperparameters and can be improved by increasing batch size. It is possible to obtain stable performance if the learning rate is set between 5e-6 and 5e-4. For most programming languages, it can achieve better performance when τ is set to 20.

5.4. RQ4: Impact of choosing different layers

To show the impacts of choosing different layers of hidden states, we design to gradually increase the number of layers to verify the effect.

Table 4
Comparison of the effects when choosing different layers.

	Ruby	JS	Go	Python	Java	PHP	AVG
Last-Layer	0.721	0.718	0.843	0.903	0.773	0.735	0.782
First-Last-Layers	0.736	0.712	0.839	0.895	0.773	0.748	0.784
First-Last-Two	0.742	0.731	0.837	0.891	0.769	0.762	0.793
First-Last-Three	0.742	0.739	0.853	0.911	0.798	0.771	0.802

It is worth mentioning that both retrieval and training phases use the same layers. We design the following variants within the memory range of the machine.

- **Last-Layer**: We choose layer-12 to represent positive and negative sample pairs. It reveals whether using a layer to conduct experiments is sufficient or not.
- **First-Last-Layers**: We only choose the first and last layers to conduct experiments, such as layer-1 and layer-12. It shows whether it is effective to use the features of layer-1 and layer-12 as low-level and high-level features, respectively.
- **First-Last-Two**: Compared with the setting of First-Last-layers, we choose the first two layers and the last two layers from hidden states to represent low-level features and high-level features, respectively, i.e., layer-1, layer-2, layer-11, and layer-12.
- **First-Last-Three**: We select the first three layers and the last three layers from hidden states. This setting is used by HFEDR.

Table 4 presents the impact of choosing different layers to conduct experiments. We have the following observations: (1) The performance of First-Last-Three is better than First-Last-two and First-Last-layers, which implies that the more layers are extracted, the better performance we can achieve. (2) The First-Last-layers have little improvement compared with the Last-layer, which shows that using a layer to represent low-level features is less effective. (3) Adding more layers is only guaranteed to improve experimental performance in some programming languages. The performance of some programming languages, such as Go, gets worse since low-level features may be low-quality representations (Tenney et al., 2019). Therefore, instead of improving performance, it reduces performance. (4) Better performance requires more layers, which means more computing resources are consumed. The HFEDR chooses layer-1, layer-2, layer-3, layer-10, layer-11, and layer-12 to provide a good trade-off between performance and computing resources.

It is worth mentioning that during the training phase, there is no need to worry about obtaining more layer features need much time, as 12 layers of features can be extracted by encoding queries and codes at once. During the inference phase, the more layers are extracted, the more resources we need to calculate the cosine similarity.

Result 4: The more layers are used to represent initial sample pairs, the better performance we can achieve. Meanwhile, it is worth considering how to provide a good trade-off between performance and computing resources.

5.5. RQ5: Time efficiency

Since code search is very sensitive to time cost, we tested and compared the time efficiency of HFEDR with other baselines. To this end, we define the following experimental setup: (1) Since the maximum sequence length is a key factor affecting the time efficiency of models, to make a fair comparison, we set the maximum sequence length of all methods to 128. (2) The test datasets we choose are the Python and Java sub-datasets in the CodeSearchNet because these two programming languages are relatively popular. (3) We choose baselines with different architectures for comparison, such as GraphCodeBERT with only encoder architecture, CodeT5 with both encoder and decoder architectures, and Unixcoder with contrastive training and encoder architecture. (4) We found that code retrieval can be divided into two stages, namely the similarity calculation stage and the sorting stage. Therefore, we will divide the retrieval time into two stages. (5) For each method, we performed 1000 retrievals and used their total retrieval times for comparison.

Table 5 shows the time efficiency comparison results of other code search models and HFEDR. We have the following findings: (1) The time efficiency of code search is close for code pre-trained models with only encoder architecture, such as GraphCodeBERT and Unixcoder. (2) The CodeT5 model with both encoder and decoder architecture consumes the highest code search time. The time efficiency of HFEDR is close to that of other models with only encoder architecture because HFEDR is based on the encoder architecture model, and it extracts multi-layer features in one encoding rather than layer-by-layer extraction.

Result 5: The time efficiency of HFEDR is close to that of models with only encoder architecture, and their time efficiency is better than that of models with both encoder and decoder architectures.

5.6. RQ6: model-free components

To verify that the components in the framework are model-free, we first use the most classic pre-training model, CodeBERT, to replace the GraphCodeBERT model in our framework. Then, we perform the same ablation experiment on CodeBERT as in Section 5.2.

Specifically, these experiments include removing the BERT-whitening module (w/o whitening), removing the hierarchical-uncorrelated pairs module (w/o Uncor_Pairs), removing the hierarchical-correlated pairs module (w/o Cor_Pairs), and removing extracting hierarchical features module (w/o Multi_Lay).

Table 6 shows our experimental results by replacing the pre-trained model and performing ablation experiments to verify the model-free components. We have the following findings: (1) HFEDR (CodeBERT) outperforms the other four variants, demonstrating these components' usefulness in the CodeBERT model. (2) Removing the Cor_Pairs component has the greatest impact on the framework's performance, which shows that it is the most critical component. (3) By combining these components to improve the performance of GraphCodeBERT and CodeBERT, we can infer that the proposed components are model-free.

Result 6: Experiments demonstrate that the components in our framework are model-free.

5.7. Case study

As revealed in Fig. 3, we show the search result of HFEDR, Unixcoder, and GraphCodeBERT for a query sentence "Disconnects and kill the associated network manager". Fig. 3(a) shows that HFEDR and Unixcoder can retrieve a matched code snippet. While GraphCodeBERT returns an unmatched code snippet. The keywords in the query are disconnects and kill. It can be seen that both approaches return results that hit some of the keywords. However, the result returned by HFEDR and Unixcoder is more relevant than GraphCodeBERT.

As revealed in Fig. 4, we show the search result of HFEDR and Unixcoder for a query sentence "Appends other buffer to this one". We can find that both models establish a relationship between the query and the code through the keyword "Buffer append". However, Unixcoder does not retrieve the most matching code snippet. It proves that our method has more robust retrieval capabilities. As shown in Table 7, We use matching rankings calculated by these models for these queries and codes to explain the model's capabilities quantitatively. The

Table 5
Comparison of time efficiency between HFEDR and other baselines.(second)

	Similarity Calculation		Array Sorting		Total		AVG
Model	Python	Java	Python	Java	Python	Java	-
GraphCodeBERT	1121	1137	1.616	1.612	1122.616	1138.612	1130.614
CodeT5	1750	1767	1.622	1.653	1751.622	1768.653	1760.138
Unixcoder	1132	1125	1.628	1.648	1133.628	1126.648	1130.138
HFEDR	1296	1225	1.632	1.663	1297.632	1226.663	1262.148

```
public void dispose() {
  super.dispose();
  if(fields != null){
    Iterator<FieldEditor> e = fields.iterator();
    while (e.hasNext()){
      FieldEditor pe = e.next();
      pe.setPage(null);
      pe.setPropertyChangeListener(null);
      pe.setPreferenceStore(null);
    }
  }
}
```

(a) The top 1 result of HFEDR and Unixcoder.

```
public void kill( final OClientConnection connection){
  if(connection != null){
    final ONetworkProtocol protocol
                = connection.getProtocol();
  try{
    interrupt();
  }catch(Exception e){
    OLogManager.instance().error(this,"Error",e);
  }
  disconnect(connection);
  protocol.sendShutdown();
  }
}
```

(b) The top 1 result of GraphCodeBERT.

Fig. 3. The top-1 results of HFEDR, Unixcoder and GraphCodeBERT for a query "Disconnects and kill the associated network manager".

Table 6
Verify components are model-free on the CodeBERT model.

	Ruby	JS	Go	Python	Java	PHP	AVG
w/o whitening	0.731	0.702	0.834	0.899	0.775	0.711	0.775
w/o Uncor_Pairs	0.728	0.691	0.837	0.875	0.768	0.704	0.767
w/o Cor_Pairs	0.713	0.685	0.822	0.881	0.763	0.685	0.758
w/o multi-Lay	0.725	0.692	0.844	0.881	0.764	0.692	0.766
HFEDR(CodeBERT)	0.741	0.715	0.852	0.893	0.785	0.724	0.785

Table 7
Different models on query-1 and query-2 calculate the Matching rankings. The query-1 refers to "Disconnects and kill the associated network manager," and the query-2 refers to "Appends other buffer to this one.".

	matching rankings	
Model	query-1	query-2
GraphCodeBERT	2	5
Unixcoder	1	2
HFEDR	1	1

matching ranking refers to the ranking of ground truth in the model's retrieval list.

As shown in Fig. 5, we show a case where HFEDR retrieval fails. A query statement is "converts excel rows into a list of objects". The ground truth of the query is in Fig. 5(a), but the result retrieved by HFEDR is in Fig. 5(b). We can see that some of the tokens in the query, such as "excel", "list", and "objects", are the same as some of the tokens in the code in Fig. 5(a) and 5(b). After various experimental verifications, we found that the main reason why HFEDR obtained this search result is that the number of keywords in the query hit by the code in Fig. 5(b) is greater than that in Fig. 5(a). In addition, we obtain some other reasons for retrieval failure after many experiments, and we will discuss these in Section 6.

6. Threats to validity

Although effective, we recognize that HFEDR may suffer from the following threats to validity:

- **Time-consuming**: During training, we use GraphCodeBERT to extract multiple layers of features from both query and code, which means that the memory footprint of features and the training time of the model will increase. In addition, although the BERT-whitening operation reduces some memory usage and retrieval time consumption during retrieval, these problems still need to be entirely resolved. In the future, we will explore whether it is possible to represent high-level and low-level features by decision-making to select representative layers in lower layers and higher layers.
- **Choosing different layers**: Due to the restriction of computing resources, we only combine a few layers to represent low-level and high-level features. We have yet to try various combinations of layers to perform experiments. We will explore these issues further in the future.
- **Long queries and long codes**: The length of queries or codes also affects the model's performance because the maximum sequence length that the pre-trained model can receive is limited, and long codes may reduce the model's performance. Therefore, long code searches need to be avoided for pre-trained based models. Fortunately, there are many approaches to deal with this problem in NLP, which we can apply to our domain in the future.
- **Identical tokens between queries and codes**: If queries and codes have some identical tokens, it is easier for our framework to determine that they are matched. The main reason for this is that most matching queries and codes have the same tokens in the training dataset. And we also found this with other baselines.
- **Impact between similar candidate codes**: A query may match other code fragments that are similar to the original matched code fragment. This phenomenon is difficult to avoid because similar codes have similar semantics. Fortunately, the original matching code is also ranked higher in retrieval lists.

7. Conclusions and future work

In this paper, we propose a novel solution named HFEDR to improve code search performance by extracting hierarchical features and reorganizing training data during training. Specifically, HFEDR first extracts low-level and high-level features from queries and codes through the advanced pre-training model GraphCodeBERT to achieve multi-view and comprehensive semantic representation. Then, HFEDR constructs hierarchical-uncorrelated feature pairs based on these hierarchical features and reorganizes these feature pairs in each layer into multiple

```
public Buffer append(final Buffer buffer){
    if(buffer.list.isEmpty())
    { // nothing to append return buffer ;
    }
    list.addAll(buffer.list);
    last = buffer.last;
    size += buffer.size;
    return this;
}
```

```
public static void list(JBossStringBuilder buffer, Collection objects
)
{
  if(objects == null) return;
  buffer.append(' ');
  if(objects.isEmpty() == false)
  {
    for(Iterator i = objects.iterator(); i.hasNext();)
    {
      Object object = i.next();
      if(object instanceof JBossObject)((JBossObject)
object).toShortString(buffer);
      else buffer.append(object.toString());
      if(i.hasNext()) buffer.append(", ");
    }
  }
  buffer.append(' ');
}
```

Fig. 4. The top-1 results of HFEDR and GraphCodeBERT for a query "Appends other buffer to this one".

```
public static synchronized < T > void fromExcel(final File file,
 final Class < T > type, final PoijiOptions options, final Consumer
< super T > consumer)
{
  final Unmarshaller unmarshaller = deserializer(file, options);
  unmarshaller.unmarshal(type, consumer);
}
```

(a) the ground truth of the query:
"converts excel rows into a list of objects"

```
private Object generifyList(final List list, final Class
 componentType)
{
  for(int i = 0; i < list.size(); i++)
  {
    Object element = list.get(i);
    if(element != null)
    {
      if(element instanceof Map)
      {
        Object bean = map2bean((Map) element, componentType
);
        list.set(i, bean);
      }
      else
      {
        Object value = convert(element, componentType);
        list.set(i, value);
      }
    }
  }
  return list;
}
```

(b) The top 1 result of HFEDR.

Fig. 5. HFEDR retrieval failure case.

sets of hierarchical-correlated feature pairs to achieve better training models with more data. Finally, HFEDR updates the model's parameters using contrastive training during the training phase, ensuring the similarity of the matched query and code is greater than the unmatched. BERT whitening operation is also incorporated in the code retrieval stage to obtain better performance and save storage space. To validate the effectiveness of HFEDR, we perform extensive experiments on CodeSearchNet. The results show that HFEDR achieves state-of-the-art performance for code search tasks. Further micro-level analyses demonstrate how predefined hyper-parameters affect our methods, how components of HFEDR affect the results, and how to choose the different layers to get a good trade-off between performance and computing resources.

In the future, we plan to extend our work in the following two directions. First, we will study how to reduce code search time consumption. It would be beneficial to do an online search. Second, we consider incorporating external knowledge, such as git commit logs, to improve the performance of code search.

CRediT authorship contribution statement

Fan Zhang: Conceptualization, Methodology, Software, Writing – original draft. **Manman Peng:** Supervision, Project administration, Writing – review & editing. **Yuanyuan Shen:** Software, Validation, Formal analysis. **Qiang Wu:** Supervision, Methodology.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

I have shared the link to my code/data in the manuscript footnote.

References

Ahmad, W.U., Chakraborty, S., Ray, B., Chang, K.-W., 2021. Unified pre-training for program understanding and generation. arXiv preprint arXiv:2103.06333.

Allamanis, M., Tarlow, D., Gordon, A., Wei, Y., 2015. Bimodal modelling of source code and natural language. In: International Conference on Machine Learning. PMLR, pp. 2123–2132.

Bui, N.D., Yu, Y., Jiang, L., 2021. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In: Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 511–521.

Chai, Y., Zhang, H., Shen, B., Gu, X., 2022. Cross-domain deep code search with few-shot meta learning. arXiv preprint arXiv:2201.00150.
Chen, T., Kornblith, S., Norouzi, M., Hinton, G., 2020. A simple framework for contrastive learning of visual representations. In: International Conference on Machine Learning. PMLR, pp. 1597–1607.
Chen, X., Xie, S., He, K., 2021. An empirical study of training self-supervised vision transformers. In: Proceedings of the IEEE/CVF International Conference on Computer Vision. pp. 9640–9649.
Cheng, Y., Kuang, L., 2022. CSRS: Code search with relevance matching and semantic matching. arXiv preprint arXiv:2203.07736.
Conneau, A., Kiela, D., Schwenk, H., Barrault, L., Bordes, A., 2017. Supervised learning of universal sentence representations from natural language inference data.
Craswell, N., 2009. Mean reciprocal rank.. In: Encyclopedia of database systems, vol. 1703.
Devlin, J., Chang, M.-W., Lee, K., Toutanova, K., 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.
Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M., 2020. CodeBERT: A pre-trained model for programming and natural languages. In: Cohn, T., He, Y., Liu, Y. (Eds.), Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020. In: Findings of ACL, vol. EMNLP 2020, Association for Computational Linguistics, pp. 1536–1547. http://dx.doi.org/10.18653/v1/2020.findings-emnlp.139.
Gao, T., Yao, X., Chen, D., 2021. Simcse: Simple contrastive learning of sentence embeddings. In: Moens, M., Huang, X., Specia, L., Yih, S.W. (Eds.), Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing. EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021, Association for Computational Linguistics, pp. 6894–6910. http://dx.doi.org/10.18653/v1/2021.emnlp-main.552.
Graves, A., 2012. Long short-term memory. Supervised Seq. Label. Recurr. Neural Netw. 37–45.
Gu, X., Zhang, H., Kim, S., 2018. Deep code search. In: 2018 IEEE/ACM 40th International Conference on Software Engineering. ICSE, IEEE, pp. 933–944.
Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., Yin, J., 2022. Unixcoder: Unified cross-modal pre-training for code representation. arXiv preprint arXiv:2203.03850.
Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S.K., Clement, C.B., Drain, D., Sundaresan, N., Yin, J., Jiang, D., Zhou, M., 2021. GraphCodeBERT: Pre-training code representations with data flow. In: 9th International Conference on Learning Representations. ICLR 2021, Virtual Event, Austria, May 3-7, 2021, OpenReview.net, URL https://openreview.net/forum?id=jLoC4ez43PZ.
Hu, H., Liu, J., Zhang, X., Cao, B., Cheng, S., Long, T., 2023. A mutual embedded self-attention network model for code search. J. Syst. Softw. 198, 111591. http://dx.doi.org/10.1016/j.jss.2022.111591, URL https://www.sciencedirect.com/science/article/pii/S0164121222002679.
Huang, J., Tang, D., Shou, L., Gong, M., Xu, K., Jiang, D., Zhou, M., Duan, N., 2021. Cosqa: 20,000+ web queries for code search and question answering. arXiv preprint arXiv:2105.13239.
Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., Brockschmidt, M., 2019. Codesearchnet challenge: Evaluating the state of semantic code search. arXiv preprint arXiv:1909.09436.
Jianlin, S., 2022. CoSENT:A more efficient sentence vector scheme than sentence-BERT. URL https://kexue.fm/archives/8847.
Kingma, D.P., Ba, J., 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.
Liu, S., Fan, H., Qian, S., Chen, Y., Ding, W., Wang, Z., 2021a. Hit: Hierarchical transformer with momentum contrast for video-text retrieval. In: Proceedings of the IEEE/CVF International Conference on Computer Vision. pp. 11915–11925.
Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., Stoyanov, V., 2019. Roberta: A robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692.
Liu, C., Xia, X., Lo, D., Gao, C., Yang, X., Grundy, J., 2021b. Opportunities and challenges in code search tools. ACM Comput. Surv. 54 (9), 1–40.
Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., et al., 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. arXiv preprint arXiv:2102.04664.
Lu, J., Yang, J., Batra, D., Parikh, D., 2016. Hierarchical question-image co-attention for visual question answering. Adv. Neural Inf. Process. Syst. 29.
Lv, F., Zhang, H., Lou, J.-g., Wang, S., Zhang, D., Zhao, J., 2015. 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, IEEE, pp. 260–270.
Niu, C., Li, C., Ng, V., Ge, J., Huang, L., Luo, B., 2022. Spt-code: Sequence-to-sequence pre-training for learning source code representations. In: Proceedings of the 44th International Conference on Software Engineering. pp. 2006–2018.
Peters, M.E., Neumann, M., Zettlemoyer, L., Yih, W., 2018. Dissecting contextual word embeddings: Architecture and representation. In: Riloff, E., Chiang, D., Hockenmaier, J., Tsujii, J. (Eds.), Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing. Brussels, Belgium, October 31 - November 4, 2018, Association for Computational Linguistics, pp. 1499–1509. http://dx.doi.org/10.18653/v1/d18-1179.
Reimers, N., Gurevych, I., 2019. Sentence-BERT: Sentence embeddings using siamese BERT-networks.
Shi, Z., Xiong, Y., Zhang, Y., Jiang, Z., Zhao, J., Wang, L., Li, S., 2023. Improving code search with multi-modal momentum contrastive learning. In: 2023 IEEE/ACM 31st International Conference on Program Comprehension. ICPC, IEEE, pp. 280–291.
Su, J., Cao, J., Liu, W., Ou, Y., 2021. Whitening sentence representations for better semantics and faster retrieval. arXiv preprint arXiv:2103.15316.
Sun, W., Fang, C., Chen, Y., Tao, G., Han, T., Zhang, Q., 2022. Code search based on context-aware code translation. In: 44th IEEE/ACM 44th International Conference on Software Engineering. ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022, ACM, pp. 388–400. http://dx.doi.org/10.1145/3510003.3510140.
Tenney, I., Das, D., Pavlick, E., 2019. BERT rediscovers the classical NLP pipeline. In: Korhonen, A., Traum, D.R., Màrquez, L. (Eds.), Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28-August 2, 2019, Volume 1: Long Papers. Association for Computational Linguistics, pp. 4593–4601. http://dx.doi.org/10.18653/v1/p19-1452.
Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I., 2017. Attention is all you need. Adv. Neural Inf. Process. Syst. 30.
Vig, J., 2019. A multiscale visualization of attention in the transformer model. In: Costa-jussà, M.R., Alfonseca, E. (Eds.), Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28 - August 2, 2019, Volume 3: System Demonstrations. Association for Computational Linguistics, pp. 37–42. http://dx.doi.org/10.18653/v1/p19-3007.
Vilalta, R., Drissi, Y., 2002. A perspective view and survey of meta-learning. Artif. Intell. Rev. 18 (2), 77–95.
Wang, Y., Wang, W., Joty, S., Hoi, S.C., 2021a. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859.
Wang, X., Wang, Y., Mi, F., Zhou, P., Wan, Y., Liu, X., Li, L., Wu, H., Liu, J., Jiang, X., 2021b. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. arXiv preprint arXiv:2108.04556.
Wu, H.C., Luk, R.W.P., Wong, K.F., Kwok, K.L., 2008. Interpreting tf-idf term weights as making relevance decisions. ACM Trans. Inf. Syst. (TOIS) 26 (3), 1–37.
Xu, L., Yang, H., Liu, C., Shuai, J., Yan, M., Lei, Y., Xu, Z., 2021. Two-stage attention-based model for code search with textual and structural features. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 342–353.

Fan Zhang received the bachelor's degree from Anhui Polytechnic University, China, in 2017. He received the master's degree from Hunan University, China, in 2020. He is currently pursuing the Ph.D. degree from Hunan University, Changsha, China. His research interests include program analysis and machine learning.

Manman Peng received her bachelor's degree, master's degree, and a Ph.D. degree in computer science from Hunan University, in 1985, in 1988, and 2006, respectively. In 1988, she joined the College of Computer Science and Electronic Engineering. She is currently a Professor and doctoral supervisor. Her research interests include computer architecture and high performance computing. Dr. Peng is a member of the Professional Committee of Computer Architecture of China Computer Federation.

Yuanyuan Shen received the Ph.D. degree from Hunan University, China, in 2023. She is currently join School of Data Science and Technology, North University of China, Taiyuan, China. Her research interests include computer architecture, program analysis, and machine learning.

Qiang Wu received the master's degree in computer science from Tsinghua University, in 2002, and the Ph.D. degree from the Imperial College of Technology, in 2009. He is currently an Associate Professor at the College of Computer Science and Electronic Engineering. His main areas of research interests are computer architecture, machine learning, and Heterogeneous computing.