Information and Software Technology 177 (2025) 107589

Contents lists available at ScienceDirect

Information and Software Technology

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

An intent-enhanced feedback extension model for code search

Haize Hu [a], Mengge Fang [a,*], Jianxun Liu [b]

[a] Guangxi Normal University, School of Computer Science and Engineering & School of Software, Guilin, 541001, Guangxi, China
[b] Hunan University of Science and Technology, School of Computer Science and Engineering, Xiangtan, 411100, Hunan, China

ARTICLE INFO

Keywords:
Code search
Query expansion
Intentional enhancement
Feedback mechanisms

ABSTRACT

Context: Queries and descriptions used for code search not only differ in semantics and syntax, but also in structural features. Therefore, solving the differences between them is of great significance to the study of code search.
Objective: This study focuses on the improvement of code search accuracy by exploring the expansion of query statements during the search process.
Methods: To address the disparities between description and query, the paper introduces the Intentional Enhancement and Feedback (QEIEF) query expansion model. QEIEF leverages the written description provided by developers as the source for query expansion. Furthermore, QEIEF incorporates theQEIEF method to enhance the semantic representation of the query. This involves utilizing the query output as the target for intent enhancement and integrating it back into the query.
Results: To assess the effectiveness of the proposedQEIEF in code search tasks, we conducted experiments using two base models (DeepCS and UNIF) along withQEIEF, as well as baseline models (WordNet and BM25). The experimental results indicate that QEIEF outperforms the baseline models in terms of query expansion accuracy and code search results.
Conclusion: QEIEF not only enhances the accuracy of query expansion but also substantially improves code search performance. The source code and data associated with our study can be accessed publicly at: The address of our new code and data is https://github.com/xiangzheng666/IST-IEFE.

1. Introduction

As the open-source community continues to expand, code search has become an integral part of the software programming process [1]. Developers frequently rely on code search to find similar code fragments in open-source repositories, which they can modify or reuse to improve software development efficiency. Studies have shown that more than 90% of developers actively search for existing code in open-source repositories for the purpose of modification and reuse [2]. Consequently, investigating code search practices holds paramount importance for the progress of software engineering.

In the initial stages of code search research, information retrieval techniques [3] predominantly relied on word-based matching, overlooking the semantic gap between code language and natural language [4]. Consequently, these techniques yielded less accurate code search results. To address this semantic divide, Gu et al. [5] introduced deep learning to code search tasks and proposed the DeepCS model. This pioneering model utilizes deep learning to embed code language and natural language into a shared vector space, effectively bridging the semantic gap that existed in traditional information retrieval techniques. The incorporation of deep learning has significantly improved the effectiveness and efficiency of code search, leading to its widespread adoption among code search researchers.

The accuracy of code search heavily relies on the query's ability to accurately represent the desired source code [6]. Previous studies in code search have incorporated code description and source code for co-embedding during the training of search models, allowing the acquisition of neural network model parameters [7]. In the search process, a query is employed to find matches with source code on a one-to-one basis [8]. However, there exist inherent differences between query and code description. Query typically consist of shorter sentences compared to code description, which may not precisely convey the intended code requirements [9]. Furthermore, developers employ different search methods that can result in varied search outcomes. To tackle these challenges, leveraging additional information to expand the query statement can enhance its accuracy and, consequently, improve the precision of code search results. Additionally, it is important

* Corresponding author.
E-mail addresses: HHZ@gxnu.edu.cn (H. Hu), fmg@stu.gxnu.edu.cn (M. Fang).

https://doi.org/10.1016/j.infsof.2024.107589
Received 4 November 2023; Received in revised form 20 September 2024; Accepted 23 September 2024
Available online 27 September 2024

to note that while the code description originates from the code developer, the query is generated by the searcher, and their understanding of the code may differ. Therefore, developers cannot expect to achieve identical results obtained from code description training when utilizing a query for search purposes in the context of code search.

To address the disparity between query statements and code description in existing code search research, query expansion techniques have been introduced. Query expansion refers to the automatic expansion of the developer's entered query, which helps to narrow the gap between the query and the code description, consequently enhancing the effectiveness of code search. By expanding the query, additional information can be incorporated that aligns more closely with the code description, leading to better matching and more accurate search results. Query expansion techniques aim to enrich the query with relevant terms, synonyms, related concepts, or code snippets to improve its representation of the desired code. This approach helps developers find more relevant code snippets and enhances the efficiency of the code search process.

The main goal of current query expansion research is to expand the query to include additional terms or phrases, in order to enhance information retrieval accuracy and reduce the semantic gap between the user's query and the documents in the collection. However, there are still three primary problems associated with existing query expansion techniques.

(1) One of the main issues is that the existing datasets primarily concentrate on query-code pairs, neglecting the need for research datasets specifically designed for query expansion.

(2) The source code description reflects the comprehension of the code developer, while the query represents the understanding of the subsequent searcher. Consequently, there may be discrepancies between the two perspectives when describing the same code.

(3) Another problem is that query expansion techniques predominantly focus on expanding the query statements without adequately considering the intent of the searchers. This oversight disregards the crucial aspect of aligning the expanded query with the searchers' specific information needs.

To address the aforementioned problems of query expansion, we propose a novel query expansion model called QEIEF (Query Expansion with Intent Enhancement Feedback), which leverages search intent enhancement. Firstly, we create a research dataset (Query-Code-Description) specifically designed for query expansion. Secondly, QEIEF conducts intent ranking on the initial query results to identify the most relevant search results. Subsequently, QEIEF utilizes the search intent to semantically enrich the query statement and expand its scope. Lastly, QEIEF generates the optimal search results based on the expanded query.

(1) We introduce the CodesearchQE dataset (Query-Code-Description) specifically designed for query expansion research, which defined as CodesearchQE. We demonstrate the effectiveness and utility of this dataset in enabling comprehensive query expansion studies.

(2) We propose a novel query expansion model, QEIEF, which leverages search intent enhancement. This model addresses the limitations of existing query expansion techniques and aims to improve the accuracy of information retrieval.

(3) Using the CodesearchQE dataset (Query-Code-Description), we investigate the impact of QEIEF on code search effectiveness. We evaluate the performance of QEIEF in terms of retrieving relevant code snippets based on the expanded query.

(4) We conduct an in-depth analysis of the individual modules and parameters within QEIEF. By examining their contributions, we gain insights into the effectiveness and optimization of the model's components, aiding in further refinement and improvement.

The remaining sections of the manuscript are organized as follows: Section 2: Introduction of the QEIEF model. Section 3: Description of the experimental evaluation environment, including the construction of the data and the preparation for the experiments. Section 4: Presentation and discussion of the experiment results, including Structure Analysis, Comparative Experimental Analysis, and Parameter Adjustment Analysis. Section 5: Discussion of the experimental results. Section 6: Presentation of the related work on code search. Section 7: Discussion of the potential threats to the research validity of the paper. Section 8: Conclusion of the full-text work.

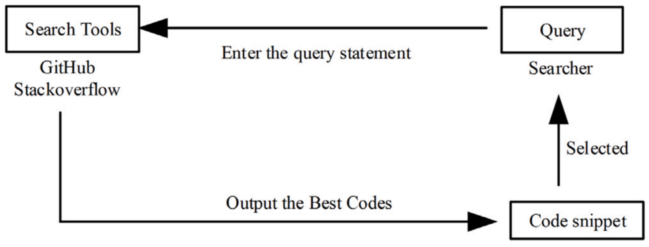


Fig. 1. Code search framework.

2. Backgrounds

2.1. Code search definition

During the software development process, developers often encounter two main types of tasks: modifying existing functionality and developing new functionality. To streamline the process of modifying existing functionality, developers can make use of open source code repositories or engage with open source communities [5]. By querying and refining the search results, developers can reduce the development time for existing code and allocate more time to the development of new functional code. This process, known as code search, is commonly performed by developers in open source repositories or open source communities, as depicted in Fig. 1. Developers input the description or specification of the desired code function into the search tool, which then matches the input against the available code sources. The search tool subsequently generates corresponding code snippets that match the query, enabling developers to select and reuse suitable code based on these results.

2.2. LSTM model

LSTM (Long Short-Term Memory) is a type of recurrent neural network (RNN) that addresses the challenge of handling long-term dependencies [5]. It incorporates a specialized mechanism for selectively storing and retrieving important information. Fig. 2 illustrates the structure of an LSTM network.

$$\begin{aligned}
i_t &= \sigma\left(W_{ix}\cdot x_t + W_{ih}h_{t-1} + b_i\right)\\
c_t &= \tanh\left(W_{cx}\cdot x_t + W_{ch}h_{t-1} + b_c\right)\\
f_t &= \sigma\left(W_{fx}\cdot x_t + W_{fh}\cdot h_{t-1} + b_f\right)\\
c_{Lt} &= f_t c_{t-1} + i_t c_t\\
o_t &= \sigma(W_{ox}\cdot x_t + W_{oh}\cdot h_{t-1} + b_o)\\
h_t &= o_t \tanh(c_t)
\end{aligned} \tag{1}$$

where, i_t is the input gate, x_t is the current input information, c_t is the current information state, f_t is the output of the forgetting gate, c_{Lt} is the current added or deleted information, o_t is the output information of the output gate, h_t is the final output of the LSTM, $thah()$ is the activation function, and W_{ix}, W_{cx}, W_{fx}, and W_{ox} are the parameter moment matrices corresponding to the LSTM input gate, current information state, forgetting gate, and output gate, respectively.

3. Method

To address the limitation of existing query expansion studies that overlook query intent, we propose a QEIEF (Query Expansion with

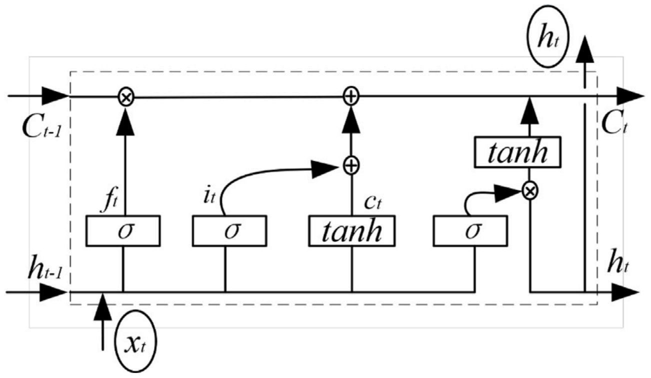


Fig. 2. LSTM network.

Intent-aware Embedding Fusion) query expansion model. The overall framework of the model is illustrated in Fig. 3.

Based on Fig. 3, the QEIEF framework is illustrated as consisting of three modules: Code Data, Offline Training, and Online Searching. The Code Data module serves as the dataset used for model training and testing, and it comprises the CodesearchQE dataset (Query-Code-Description). In this manuscript, we have constructed our own dataset, which is extensively described in Section 3. During Offline Training, the deep learning network parameters undergo training, resulting in the acquisition of a deep learning network specifically tailored for the code search task. In Online Searching, a portion of the new query is used for searching. Throughout the search process, the query is expanded, and the output corresponds to the optimal code fragment.

3.1. *Off-line training*

The purpose of offline training is to derive model parameters through feature extraction from a substantial volume of data. In the code search task, the offline training model emphasizes the joint embedding of the source code and natural language description, subsequently establishing the mapping relationship between them. The training framework is depicted in Fig. 4.

As depicted in the figure, the training of neural networks primarily involves similarity calculations using Code and Description. In this context, the $Description^{+}$ refers to the unique and positively correlated description language associated with the corresponding code fragment. Conversely, to enhance model training accuracy, multiple negative samples referred to as $Description^{-}$ are assigned to each code fragment. The objective of model training is to increase the calculated similarity value between code snippets and positive sample description, while decreasing the calculated similarity value between code snippets and negative samples. Continuous training is performed, wherein the model parameters are iteratively adjusted until the termination condition for model training is met.

3.1.1. *Data serialization*

One of the primary objectives of offline training is to extract the feature information present in the source code and establish an accurate mapping relationship between the source code and the query. The feature information within the source code encompasses not only token-level details but also logical structural information within the context. Consequently, we preprocess the source code by dividing it into four sequences: Methodname (represented by M), API (represented by A), Tokens (represented by T), and Description (represented by D) prior to training. To capture the information from each of these four sequences, we employ a BiLSTM network to generate the corresponding sequence vectors.

In our study, the primary focus lies in query expansion to enhance the accuracy of code search. The purpose of query expansion research is to expand the input query statement, focusing solely on natural language research. With the continuous application and development of deep learning models in code search, relying solely on a single deep learning model to enhance code search efficiency has certain limitations. Therefore, the article explores enhancing code search research from the perspective of combining natural language expansion on the basis of deep learning models. The article adopts a feedback enhancement method for natural language expansion, which does not rely on the learning and training of deep learning. However, the expansion method needs to be validated and analyzed based on the code search task. Therefore, the query expansion method proposed in the article needs to be based on a foundational code search model. However, existing code search research has limited content on query expansion research, with diverse datasets, which has prevented the application of new-generation technologies such as large models in code search query expansion research. Thus, the article adopts the DeepCS foundational code search model proposed by Gu et al. using bidirectional LSTM as the network for data feature representation and learning. The primary distinction with DeepCS lies in the input segment of the query statement. In DeepCS, the query is directly feature extracted and trained. In our article, the approach we advocate centers on expanding the query to minimize the disparity between the query and the description. Building on existing research, this serves as a foundational study for the application of more advanced deep learning models and new-generation technologies such as large models in the future.

To achieve this goal, we employ the widely utilized bi-directional long-short memory network (BiLSTM) for offline training of the data. The BiLSTM architecture, as shown in Fig. 5, is utilized to extract the sequence data features, thereby improving the accuracy of code feature characterization.

The source code is parsed to obtain four sequences, all of which exist as tokens in the composition of the sequence. Assume that the source code parsed Methodname sequence is $M = \{m_{1M}, m_{2M}, m_{3M}, \ldots, m_{nM}\}$, and the API sequence is $A = \{a_{1a}, a_{2a}, a_{3a}, \ldots, a_{nA}\}$, Tokens sequence is $T = \{t_{1T}, t_{2T}, t_{3T}, \ldots, t_{nT}\}$, Description sequence is $D = \{t_{1D}, t_{2D}, t_{3D}, \ldots, t_{nD}\}$.

3.1.2. *Feature extraction*

To further elaborate on feature learning, the paper selects the method sequence embedding as an example.

Camel split is used to slice the M sequences and extract word feature information from the sequences based on BiLSTM and maximum pooling. In the model feature extraction, the words in $M = \{m_{1M}, m_{2M}, m_{3M}, \ldots, m_{nM}\}$ in the words correspond to data 1, data 2, ..., data n in Fig. 6, respectively. $M = \{m_{1M}, m_{2M}, m_{3M}, \ldots, m_{nM}\}$ is viewed as a continuous sequence of words, and finally the model obtains the feature information of each word, outputs as hidden layer h_t, and uses maximum pooling to obtain the final feature volume. The results are as follows.

$$h_t = \tanh\left(W^M \left[h_{t-1}; m_{tM}\right]\right) \tag{2}$$

where, W^M is the parameter matrix of the Methodname sequence in the BiLSTM network, and m_{tM} is the vector value corresponding to the word input in the sequence. After the BiLSTM training, the hidden layer output sequence $h = h_1, h_2, h_3, \ldots, h_{nM}$, the optimal hidden layer is selected using the maximum pooling network to obtain the feature information m.

$$m = \max\text{pooling}\left(\left[h_1, h_2, h_3, \ldots, h_n\right]\right) \tag{3}$$

The same method can be used to obtain feature information of API sequence vector (a), Tokens sequence vector (t) and Description vector (d).

$$a = \max\ \text{pooling}\ \left(\left[h_1, h_2, h_3, \ldots, h_{nA}\right]\right) \tag{4}$$

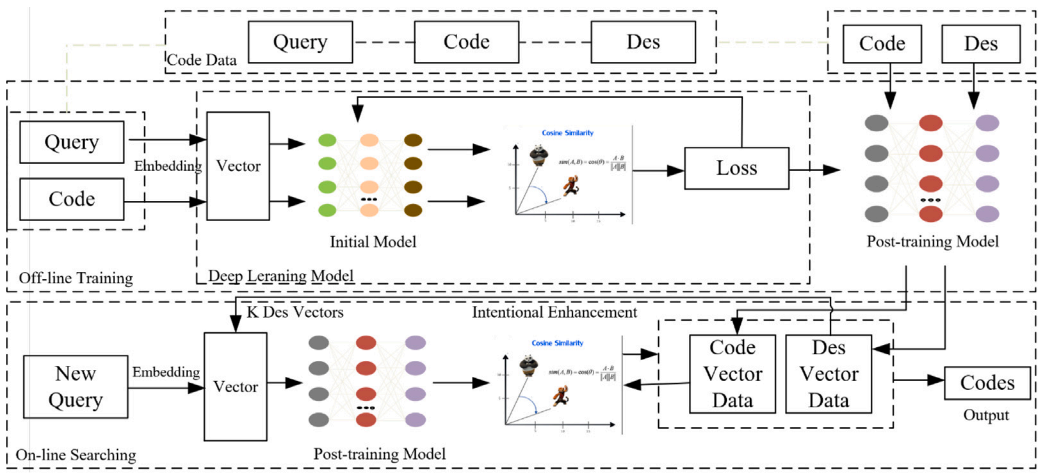


Fig. 3. QEIEF model.

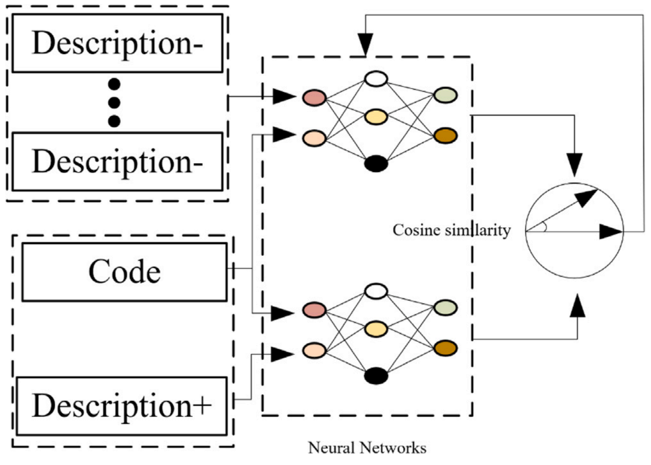


Fig. 4. Training framework.

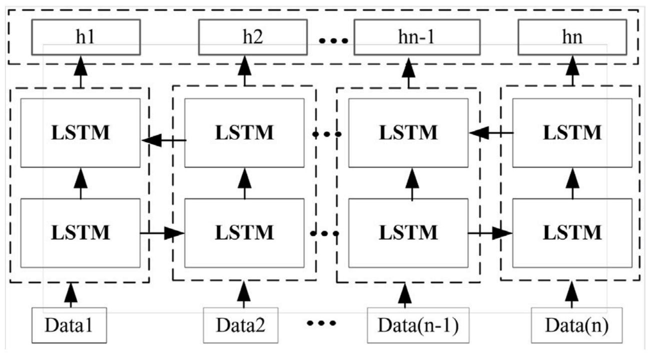


Fig. 5. LSTM network.

$$t = \max\ \text{pooling}\ \left(\left[h_1, h_2, h_3, \ldots, h_{nT}\right]\right) \tag{5}$$

$$d = \max \text{pooling}\left(\left[h_1, h_2, h_3, \ldots, h_{nD}\right]\right) \tag{6}$$

The feature information m, a, t is concatenated using concat network to obtain the source code fragment vector (c) feature information, as follows.

$$c = \text{concat}(m, a, t) \tag{7}$$

3.1.3. Model training

During the training process, the description are divided into a positively correlated description vector (d^+) and a negatively correlated description vector (d^-) to facilitate correct matching between the source code and the corresponding description. A ternary vector set $(c,\ d^+,\ d^-)$ is constructed for training purposes, where d^+ represents the description vector associated with the code fragment, and d^- represents the description vector selected from other code fragments. The model is trained using similarity calculations, aiming for high similarity between $(c,\ d^+)$ and low similarity between $(c,\ d^-)$ simultaneously. Consequently, the loss function for model training is defined as follows.

$$\text{Loss}(\theta) = \sum_{\{c,d^+,d^-\}} \max\left(0, \epsilon - \cos\left(c, d^+\right) + \cos\left(c, d^-\right)\right) \tag{8}$$

where, θ is the model adoption number, ϵ is a constant margin.

3.2. On-line searching

The model's network parameters are trained and validated through offline training. Simultaneously, the code fragments are transformed into intermediate vector representations using the trained model, resulting in the creation of a dataset comprising code fragment intermediate vectors. During online search, when a searcher submits a query, the query is treated as a description for offline search. The trained model converts the query into a query vector, and similarity calculations are employed to match the query vector with the code fragment vectors one by one. Subsequently, the most similar code fragment vector is recommended. The online search framework is depicted in Fig. 6.

Based on Fig. 6, a notable distinction between the online search phase and the offline training phase is that the former utilizes query instead of description, wherein both the query statements and description are expressed in natural language. During online search, developers input a query statement, which is subsequently characterized by the trained model. The characterized query statement is then matched individually with the code vector library, resulting in the retrieval of the optimal output result.

3.3. Intentionally enhanced feedback

During the offline training process, description within the code is utilized to match and train the model parameters with the corresponding source code. However, in the online searching phase, query is used as input to the search tool for similarity matching with the source code. There are two key distinctions between description and query. Firstly, description is written by code developers during code

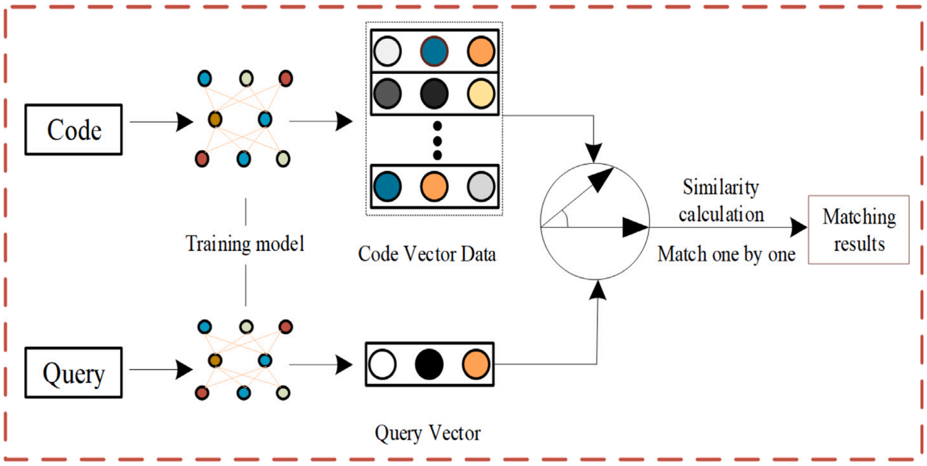
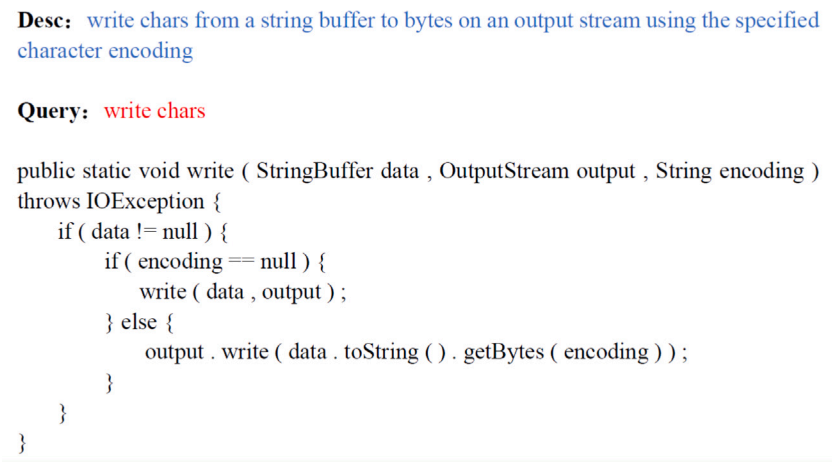


Fig. 6. On-line searching framework.



Desc: write chars from a string buffer to bytes on an output stream using the specified character encoding

Query: write chars

```
public static void write ( StringBuffer data , OutputStream output , String encoding )
throws IOException {
    if ( data != null ) {
        if ( encoding == null ) {
            write ( data , output ) ;
        } else {
            output . write ( data . toString ( ) . getBytes ( encoding ) ) ;
        }
    }
}
```

Fig. 7. Examples of differences.

composition and serve to interpret the code. On the other hand, query is inputted by searchers into the search tool to locate specific functionality within the code, representing the searcher's understanding of the code's purpose. Consequently, the description provided by code developers and searchers differ significantly in terms of semantic and syntactic composition. Secondly, description typically have longer lengths compared to query. On average, description consist of 20 to 30 words, while query typically consist of 2 to 3 words. Fig. 7 illustrates this point, where the same code fragment is described as "write chars from a string buffer to bytes on an output stream using the specified character encoding", containing 17 words. In contrast, the query "write chars" consists of only 2 words.

To address the disparities between description and query statements found in existing query methods and enhance the accuracy of code search, we introduce a query expansion method aimed at improving feedback. The query expansion framework is illustrated in Fig. 8.

In Fig. 8, the process begins by inputting the query into the trained model to obtain a vector representation of the query. Upon completing the model training, all code fragments within the source code dataset are transformed into code vectors using the trained model, forming the code vector dataset. During online searching, the query vectors are obtained and matched individually with the vectors in the code vector dataset using cosine similarity calculation.

To enhance the accuracy of online search, we incorporate intent enhancement feedback as a query expansion method. Intent enhancement feedback involves a judgment module that evaluates the results of the similarity matching. The judgment module assesses whether the current optimal accuracy value has improved compared to the previous accuracy value. If an improvement is observed, the top three code vectors from the matching results (determined through experimentation) are selected. The corresponding code description associated with these code vectors are then retrieved and concatenated with the query. This process generates extended query statements. The extended query statements are subsequently fed into the trained network model, which produces vector representations for similarity matching with the source code. Intent enhancement feedback is iteratively applied until the conditions specified by the judgment module is no longer satisfied. At that point, the source code vector with the best matching result is outputted. The use of intent-enhanced feedback for query expansion offers two advantages. Firstly, the source of query expansions in intent-enhanced feedback is derived from description within the source code. Description is written by code developers during the code-writing process to facilitate subsequent inspection and learning by others. Thus, they effectively capture the functionality of the code. Secondly, the query expansion approach based on intent enhancement feedback improves the precision of search results by incorporating relevant information from the code description.

The corresponding description of the optimal code from the matching output is utilized as the source and method for query expansion in the intent-enhanced feedback expansion. Intent-enhanced feedback expansions provide two advantages. Firstly, the query expansions in intent-enhanced feedback are derived from description within the source code. These description are authored by code developers during the code-writing process to facilitate subsequent inspection and

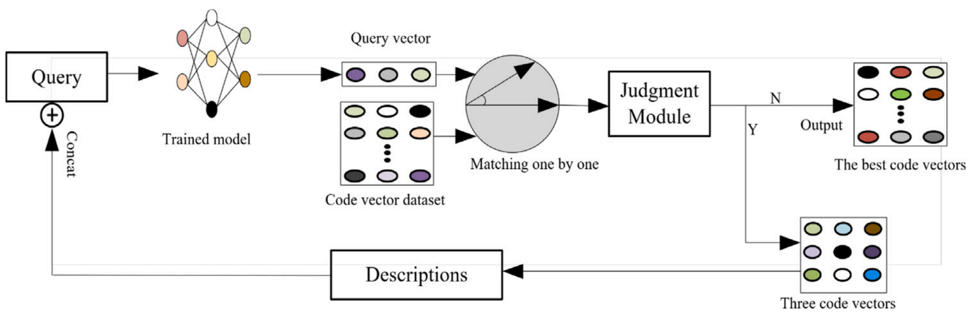


Fig. 8. Intentionally enhanced feedback model.

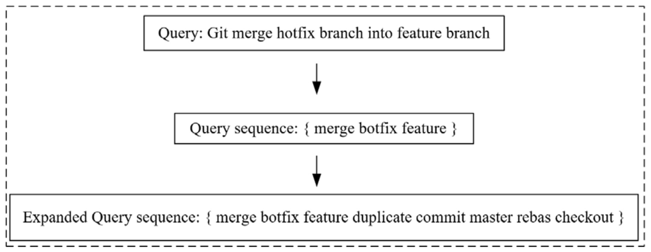

Query: Git merge hotfix branch into feature branch
Query sequence: { merge botfix feature }
Expanded Query sequence: { merge botfix feature duplicate commit master rebas checkout }

Fig. 9. Query extension example.

learning by others, effectively capturing the functionality of the code. Secondly, the query expansion approach, where the intent-enhanced feedback expansion is based on in-source code description, aligns with the natural language used in the offline training process. This consistency contributes to a search model with high accuracy. Hence, the intent-enhanced feedback expansion method possesses scaling advantages in terms of both the expansion source and method. Fig. 9 illustrates an example of query extension.

Fig. 9 depicts the extension of the query statement "Git merge hotfix branch into feature branch". The process involves two steps. Firstly, the query statement is serialized to obtain the word sequence "merge hotfix feature". Secondly, the query expansion model is employed to expand the query statement, resulting in the sequence "merge hotfix feature duplicate commit master rebase checkout". In the code search task, the expanded query statement sequence is treated as a new query statement to be searched.

4. Experiment preparation

4.1. Selection and construction of datasets

Existing code search studies primarily rely on two datasets: the dataset crawled by Gu et al. [5] and the CodesearchNet dataset [10]. The dataset crawled by Gu et al. exclusively consists of Java language data, comprising over 18 million items. On the other hand, the CodesearchNet dataset encompasses six programming languages, namely Go, Java, JavaScript, PHP, Python, and Ruby, with a total of 2 million items. Notably, the Java portion of the dataset contains 496,688 items, while the Python portion includes 457,461 items, accounting for 46.1% of the total papers in the dataset. Consequently, for the dataset construction in our study, we have initially focused on Python and Java. However, in future research, we intend to expand the dataset to include additional programming languages and explore the search performance of models in those languages.

To evaluate the effectiveness of the proposed model in the code search task, we conducted experiments using two datasets. Firstly, we utilized the CodesearchNet dataset for training and testing the Query Expansion with Intent-Enhanced Feedback (QEIEF) model. This experimental analysis aimed to assess the performance of QEIEF using the CodesearchNet dataset. Additionally, to further validate the effectiveness of QEIEF, we constructed a ternary dataset known as CodesearchQE (Query-Code-Description). The CodesearchQE dataset was primarily built based on Q&A interactions from Stack Overflow, ensuring its relevance and applicability to a broad range of researchers. However, due to time constraints during data construction, the proposed model was primarily tested on the Python and Java languages. Details of the dataset and its language distribution can be found in Table 1, where the numbers indicate the data count.

Table 1
CodesearchQE dataset.

Data	Python	Java	Total
Number	37,234	31,297	68,531

During the construction of the CodesearchQE dataset, a selective approach was employed to enhance the data's effectiveness. Not all questions and code snippets from Stack Overflow were chosen for inclusion. This decision was made to ensure the dataset's suitability for code search research. It should be noted that not all questions on Stack Overflow contain code snippets, rendering them unsuitable for our research purposes. Additionally, discussions surrounding such questions may contain irrelevant information, which could introduce noise and adversely impact the model training process. To improve the quality of the CodesearchQE dataset, the following methods were implemented during its construction:

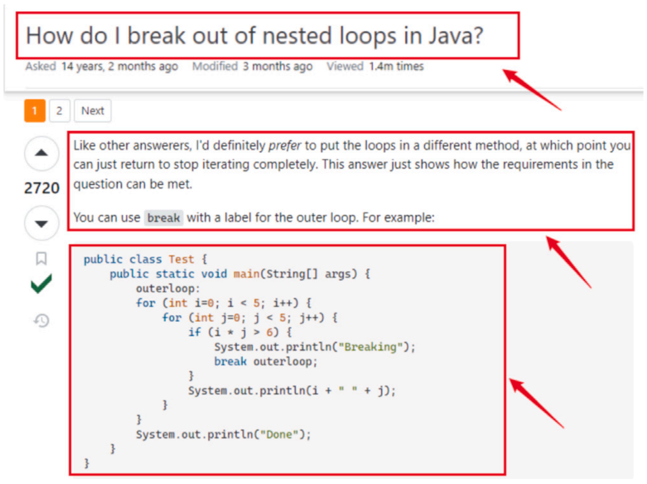
(1) In the initial stage of our research, we focused on data from Stack Overflow spanning the period from 2013 to 2023. During the construction of the CodesearchQE dataset, our primary selection was limited to Python and Java.

(2) Query selection: The selection of query, which serve as input for code search and reflect searchers' intent, followed two principles:

(1) Exclusion of 'what' questions: Through our analysis of Stack Overflow questions, we observed that questions starting with 'what' typically seek definitions rather than code-related information. For instance, a question like "What are classes in Java?" seeks a definition of 'class' without any accompanying code snippets, making it unsuitable for training the model.

(2) Inclusion of questions with code snippets: To enhance the quality of data between questions and code snippets, we specifically chose questions that contained code snippets within the comments. Furthermore, we selected code snippets accepted by the question asker as the representative code for the dataset. Query included in the dataset satisfied both the first and second principles mentioned above.

(3) Accepted comments as description: Each question on Stack Overflow receives various comments, but not all of them are valuable for our purposes. Additionally, some questions may lack comments altogether. To address this, we filtered the comments and selected those that were accepted by the question asker as the description for the dataset. Fig. 10 provides an example of the data extraction process.



How do I break out of nested loops in Java?

Asked 14 years, 2 months ago Modified 3 months ago Viewed 1.4m times

1 2 Next

2720

Like other answerers, I'd definitely *prefer* to put the loops in a different method, at which point you can just return to stop iterating completely. This answer just shows how the requirements in the question can be met.

You can use `break` with a label for the outer loop. For example:

```
public class Test {
    public static void main(String[] args) {
        outerloop:
        for (int i=0; i < 5; i++) {
            for (int j=0; j < 5; j++) {
                if (i * j > 6) {
                    System.out.println("Breaking");
                    break outerloop;
                }
                System.out.println(i + " " + j);
            }
        }
        System.out.println("Done");
    }
}
```

Fig. 10. An example of data construction.

Table 2
CodesearchNet dataset.

Language	Go	Java	JavaScript
Number	346,365	496,688	138,625
Language	PHP	Python	Ruby
Number	578,118	457,461	53,279

The CodesearchNet dataset is widely acknowledged as a prominent dataset for code search research. It consists of a comprehensive collection of 2 million data pairs, comprising query and corresponding code snippets (Query-Code). The dataset encompasses six commonly used programming languages, including Go, Java, JavaScript, PHP, Python, and Ruby. The distribution of data across these languages is presented in Table 2, where the numbers denote the respective count of data pairs.

To further assess the accuracy of the query expansion model QEIEF, we have created the CodesearchQE dataset using data from Stack Overflow. This dataset specifically focuses on the Stack Overflow platform. In the CodesearchQE dataset, the Query field represents the question posed by a searcher on Stack Overflow, which serves as the description of the desired code functionality. The Description field, on the other hand, corresponds to the response provided by code developers to the searcher's question. The Description field offers a detailed explanation of the code snippet and its relevance to the query. Consequently, the Description field in this CodesearchQE dataset directly characterizes code fragments. By utilizing the CodesearchQE dataset (as depicted in Fig. 11), we can evaluate whether the proposed model can effectively expand the Query field to match the Description field. This evaluation provides a more direct and straightforward assessment of the accuracy of the query expansion, showcasing the model's ability to accurately capture the intent and functionality of the code.

4.2. *Experimental evaluation environment*

The experimental section of our study was conducted on a Linux server system equipped with two 11 GB GPUs, specifically the Nvidia GTX 2080Ti model. For the experimental simulations, we utilized the PyTorch framework as our platform of choice, implementing the training model using the Python programming language. The experimental setup required Python version 3.6 or later, along with PyTorch version 0.4.1.

4.2.1. *Base code search model*

To compare and analyze the effectiveness of the proposed methods for code search tasks, we conducted a query expansion study on two base code search models: DeepCS and UNIF. The objective of this study was to evaluate the impact of query expansion techniques on the performance of these base models in code search.

DeepCS [5]: introduced by Gu et al. was one of the first studies to incorporate deep learning into code search. The aim of DeepCS was to bridge the semantic gap between natural language and code language by leveraging deep learning networks to embed both languages into the same vector space. In the DeepCS model, the code fragment is initially parsed into four distinct parts: Methodname sequence, API sequence, Token sequence, and Description sequence. Each of these parts captures different aspects of the code fragment. Next, DeepCS employs Long Short-Term Memory (LSTM) networks to extract the feature information from each of the four sequences, thereby obtaining sequence vectors that represent the code fragments. To obtain the vector representation of the code fragment, the Methodname sequence, API sequence, and Token sequence are concatenated or merged using a splicing technique. This merged vector captures the combined information from these three sequences. Finally, DeepCS utilizes cosine similarity to train the model to calculate the similarity between the vector representation of the code fragment and the vector representation of the corresponding Description.

UNIF [11]: Based on the NCS (Natural Code Search) word embedding model, Cambronero et al. constructed vector matrices for the source code language and the natural language description, which were formulated as query statements during the search process. The vector matrices were weighted to extract the association between the code language and natural language.

4.2.2. *Pre-training extended models*

To conduct a comprehensive evaluation of the effectiveness of our proposed query expansions, which is based on pre-training extended models, we performed a comparative analysis. Specifically, we compared our model against the most widely used existing models in the field, namely WordNet, FP, QECK, and NQE extension models.

WordNet [12]: Building upon the traditional code search model, Azad H. K. et al. aimed to enhance the accuracy of code search by mitigating the disparity between the query language and the description. To achieve this, they employed WordNet to extend the query through keyword matching and utilized the TF-IDF model to weigh the WordNet words, thereby reducing the divergence between the description and the query. The words obtained through WordNet expansion were then added to the initial query as an expanded query.

QECK [13]: The QECK extension model uses keywords for expansion, extracting semantic keywords from the StackOverflow Q&A database and using them to expand the initial query.

NQE [14]: The NQE extension model extracts method names from the source code and uses the keywords in the method names to expand the initial query.

FP [14]: The FP extension model uses frequent itemsets to analyze word frequency of code description and uses the frequency results as the source of query expansion.

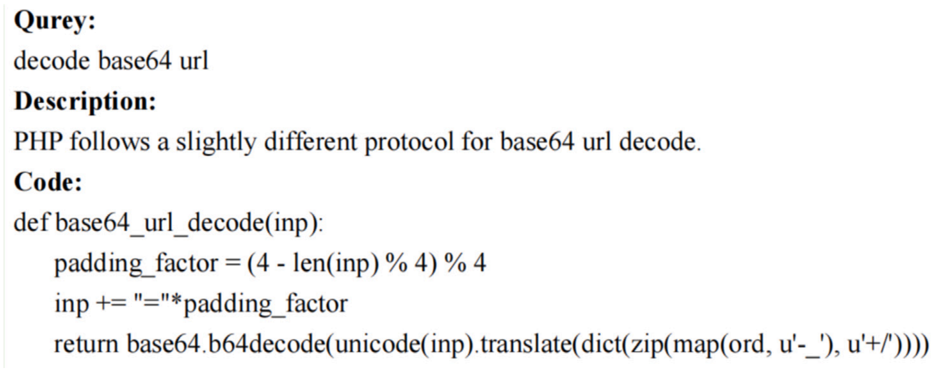
4.2.3. *Evaluation indicators*

The QEIEF study primarily builds upon the DeepCS and UNIF code search models. Additionally, existing research on query expansion models commonly employs R@k (k = 1, 5, and 10) and MRR as evaluation metrics to assess the performance of these models. In order to analyze the effectiveness of the models in code search tasks, we also utilize R@k (k = 1, 5, and 10) and MRR as evaluation metrics to measure the effectiveness of the code search model.

$$R@k = \frac{1}{|Q|}\sum_{q=1}^{Q}\delta\left(F\,\mathrm{Rank}_q \le k\right) \tag{9}$$

where, Q is a set of query, $\delta()$ is a function which returns 1 if the input is true and o otherwise.

$$MRR = \frac{1}{|Q|}\sum_{q=1}^{|Q|} F R\,\mathrm{an}\,k_q \tag{10}$$



Qurey:

decode base64 url

Description:

PHP follows a slightly different protocol for base64 url decode.

Code:

```
def base64_url_decode(inp):
    padding_factor = (4 - len(inp) % 4) % 4
    inp += "="*padding_factor
    return base64.b64decode(unicode(inp).translate(dict(zip(map(ord, u'-_'), u'+/'))))
```

Fig. 11. A example of CodesearchQE dataset.

Table 3
Results of variance analysis.

Model	Input	Python			
		R@1	R@5	R@10	MRR
DeepCS	Description	0.204	0.424	0.533	0.310
	Query	0.058	0.160	0.251	0.122
UNIF	Description	0.340	0.538	0.632	0.439
	Query	0.151	0.353	0.451	0.250
Model	**Input**	**Java**			
		R@1	R@5	R@10	MRR
DeepCS	Description	0.244	0.535	0.721	0.375
	Query	0.056	0.163	0.262	0.124
UNIF	Description	0.365	0.595	0.684	0.470
	Query	0.176	0.360	0.477	0.269

During the experiment, we assigned a value of 1 to the positive samples, while the negative samples were assigned a value of 0. The final calculation is similar to that of an RNN. Therefore, the article only utilizes an RNN for evaluating the model in the experimental analysis.

5. Result analysis

The experiment will be developed from 4 questions:

Question 1: How does the difference between Query and Description affect code search?

Question 2 : How effective is the query expansion of the QEIEF model?

Question 3 : How does QEIEF perform compared to other expansion methods?

Question 4 : What is the impact of model parameters on QEIEF?

Question 5 : How well doesQEIEF fit in pre-trained models.

5.1. How does the difference between Query and Description affect code search?

In Section 3, we provided an overview of the distinctions between the Query and Description in terms of developer logic and word length. In this section, we will further explore the divergences between the two from an experimental perspective. We will individually employ the Query and Description as inputs to compare and analyze their impact on the code search task. To ensure a fair evaluation, we will utilize the DeepCS and UNIF base code search models for our investigation. For the experiments conducted in this section, we utilized the CodesearchQE dataset that we constructed. The experimental results are presented in Table 3.

As shown in Table 3, the experimental analysis was conducted using the DeepCS and UNIF code search models on two language datasets, namely Python and Java. The results indicate that utilizing Description as input yields superior search outcomes compared to using Query for both the DeepCS and UNIF models. In the DeepCS model, when Python language was considered, employing Description as input resulted in improvements of 251.72%, 165.0%, 112.35%, and 154.10% in R@1, R@5, R@10, and MRR, respectively. Similarly, for the Java language, Description as input led to enhancements of 335.71%, 228.22%, 175.19%, and 202.42% in R@1, R@5, R@10, and MRR, respectively. Regarding the UNIF model, for the Python language, the utilization of Description as input led to improvements of 126.0%, 52.41%, 40.13%, and 75.6% in R@1, R@5, R@10, and MRR, respectively. Similarly, for the Java language, employing Description as input resulted in improvements of 107.39%, 65.28%, 43.40%, and 74.72% in R@1, R@5, R@10, and MRR, respectively. Consequently, in the context of the code search task, utilizing Description search is more effective than using Query.

Traditional code search methods typically employ Description as the training data for the model, while Query is used for online search matching instead of Description. The experimental results demonstrate a significant disparity between Query and Description in traditional code search research, and this disparity has a notable impact on the code search outcomes. This disparity can be attributed to two primary reasons. Firstly, Description is provided by the code developer, whereas Query is generated by the searcher, resulting in differences in the understanding and perspective of the same code. These contrasting viewpoints can lead to variations in the choice of terms and the overall expression of the search query. Secondly, Query tends to be shorter in length compared to Description, and it may not capture the full semantics conveyed in the Description. The brevity of Query limits its ability to effectively match the underlying meaning and intent expressed in the Description. Consequently, utilizing Description as search query can effectively enhance the accuracy of code search in comparison to employing Query.

5.2. How effective is the query expansion of the QEIEF model?

In order to evaluate the feasibility of the model's internal composition, we examine the influence of intent-enhanced feedback on the code search results. To achieve this, we conduct an experimental analysis by comparing the performance of the base model using query statements and the model with intent-enhanced feedback. The experimental analysis is conducted on two primary datasets, namely the CodeSearchNet dataset and the constructed CodesearchQE dataset. The result of the analysis for the CodeSearchNet dataset is presented in Table 4.

In the case of the DeepCS model, the QEIEF model exhibited improvements of 35.90%, 30.30%, 27.78%, and 17.46% in R@1, R@5, R@10, and MRR, respectively, compared to regular code search. Similarly, for the UNIF model, the QEIEF model demonstrated enhancements of 29.73%, 15.63%, 12.33%, and 23.53% in R@1, R@5, R@10, and MRR, respectively, in comparison to regular code search. Therefore, in the context of the CodeSearchNet dataset, the QEIEF model

Table 4
Results of structure analysis 1.

Model	Input	R@1	R@5	R@10	MRR
DeepCS	Query	0.39	0.66	0.72	0.63
	QEIEF	0.53	0.86	0.92	0.74
UNIF	Query	0.37	0.64	0.73	0.51
	QEIEF	0.48	0.74	0.82	0.63

Table 5
Results of structure analysis 2.

Model	Input	Python			
		R@1	R@5	R@10	MRR
DeepCS	Query	0.045	0.168	0.259	0.116
	QEIEF	0.321	0.499	0.657	0.432
UNIF	Query	0.073	0.206	0.292	0.148
	QEIEF	0.194	0.453	0.592	0.322
Model	**Input**	**Java**			
		R@1	R@5	R@10	MRR
DeepCS	0.058	0.160	0.251	0.203	
	QEIEF	0.321	0.499	0.657	0.432
UNIF	Query	0.176	0.360	0.477	0.310
	QEIEF	0.373	0.706	0.814	0.519

Table 6
Comparative analysis results of Python.

Model	Method	Python			
		R@1	R@5	R@10	MRR
DeepCS	WordNet	0.079	0.178	0.276	0.125
	FP	0.181	0.218	0.441	0.234
	QECK	0.282	0.383	0.511	0.357
	NQE	0.301	0.450	0.584	0.393
	QEIEF	0.321	0.499	0.657	0.432
UNIF	WordNet	0.114	0.230	0.310	0.158
	FP	0.155	0.243	0.342	0.161
	QECK	0.181	0.395	0.491	0.291
	NQE	0.201	0.421	0.511	0.301
	QEIEF	0.194	0.453	0.592	0.322

Table 7
Comparative analysis results of Java.

Model	Method	Java			
		R@1	R@5	R@10	MRR
DeepCS	WordNet	0.116	0.214	0.392	0.223
	FP	0.275	0.431	0.597	0.384
	QECK	0.291	0.451	0.621	0.405
	NQE	0.301	0.471	0.648	0.411
	QEIEF	0.321	0.499	0.657	0.432
UNIF	WordNet	0.206	0.404	0.509	0.324
	FP	0.288	0.436	0.580	0.376
	QECK	0.223	0.411	0.528	0.301
	NQE	0.241	0.431	0.552	0.312
	QEIEF	0.373	0.706	0.814	0.519

exhibits superior search performance. The results of the experimental analysis of the CodesearchQE data set is shown in Table 5.

In the DeepCS model, for the Python language, QEIEF led to improvements of 613.33%, 197.02%, 153.67%, and 272.41% in R@1, R@5, R@10, and MRR, respectively. Similarly, for the Java language, QEIEF demonstrated enhancements of 453.45%, 211.88%, 161.75%, and 112.81% in R@1, R@5, R@10, and MRR, respectively. Within the UNIF model, for the Python language, QEIEF resulted in improvements of 165.75%, 119.90%, 102.74%, and 117.57% in R@1, R@5, R@10, and MRR, respectively. Likewise, for the Java language, QEIEF exhibited improvements of 111.93%, 96.11%, 70.65%, and 67.42% in R@1, R@5, R@10, and MRR, respectively. Consequently, the utilization of intent-enhanced feedback expansion proves to be effective in enhancing the accuracy of code search in code search tasks.

5.3. How does QEIEF perform compared to other expansion methods?

To further assess the effectiveness of the proposed intention-enhanced feedback expansion model, we conduct a comparative analysis with two baseline models. While there exist several query expansion methods, it is impractical to compare them individually due to the unavailability of the source code. Hence, we employ commonly used models such as WordNet, FP, QECK, and NQE to perform the comparative analysis on the constructed CodesearchQE dataset. The results of this comparative analysis are presented in Tables 6–7.

For the Python language in the DeepCS model, QEIEF demonstrated improvements of 55.14%, 43.61%, 12.15%, and 6.23% in R@1 compared to the WordNet, FP, QECK, and NQE extension methods, respectively. Additionally, QEIEF improved R@5 by 32.46%, 56.31%, 23.25%, and 9.82%, respectively, and R@10 by 36.38%, 32.88%, 22.22%, and 11.11%, respectively. In terms of MRR, QEIEF improved the metric by 36.34%, 45.83%, 17.36%, and 9.03%, respectively.

For the Java language in the DeepCS model, QEIEF demonstrated improvements of 34.83%, 27.44%, 23.22%, and 20.58% in R@1 compared to the WordNet, FP, QECK, and NQE extension methods, respectively. Additionally, QEIEF improved R@5 by 29.00%, 24.25%, 20.74%, and 17.22%, respectively, and R@10 by 17.82%, 15.56%, 12.16%, and 8.35%, respectively. In terms of MRR, QEIEF improved the metric by 30.86%, 25.00%, 20.90%, and 19.73%, respectively.

For the UNIF model in the Python language,QEIEF demonstrated improvements of 41.23%, 20.10%, 6.70%, and (−3.61)% in R@1 compared to the WordNet, FP, QECK, and NQE extension methods, respectively. Additionally, QEIEF improved R@5 by 49.23%, 46.36%, 12.80%, and 7.06%, respectively, and R@10 by 47.64%, 42.23%, 17.06%, and 13.68%, respectively. In terms of MRR, QEIEF improved the metric by 50.93%, 50.00%, 9.63%, and 6.52%, respectively.

For the Java language in the UNIF model, QEIEF demonstrated improvements of 44.77%, 38.37%, 40.21%, and 35.39% in R@1 compared to the WordNet, FP, QECK, and NQE extension methods, respectively. Additionally, QEIEF improved R@5 by 42.78%, 38.24%, 41.78%, and 39.95%, respectively, and R@10 by 37.47%, 28.75%, 35.14%, and 32.19%, respectively. In terms of MRR, QEIEF improved the metric by 37.57%, 27.55%, 42.00%, and 39.88%, respectively.

For the Python language in the DeepCS model, QEIEF demonstrated improvements of 55.14%, 43.61%, 12.15%, and 6.23% in R@1 compared to the WordNet, FP, QECK, and NQE extension methods, respectively. Additionally, QEIEF improved R@5 by 32.46%, 56.31%, 23.25%, and 9.82%, respectively, and R@10 by 36.38%, 32.88%, 22.22%, and 11.11%, respectively. In terms of MRR, QEIEF improved the metric by 36.34%, 45.83%, 17.36%, and 9.03%, respectively.

For the Java language in the DeepCS model, QEIEF demonstrated improvements of 34.83%, 27.44%, 23.22%, and 20.58% in R@1 compared to the WordNet, FP, QECK, and NQE extension methods, respectively. Additionally, QEIEF improved R@5 by 29.00%, 24.25%, 20.74%, and 17.22%, respectively, and R@10 by 17.82%, 15.56%, 12.16%, and 8.35%, respectively. In terms of MRR, QEIEF improved the metric by 30.86%, 25.00%, 20.90%, and 19.73%, respectively.

For the UNIF model in the Python language,QEIEF demonstrated improvements of 41.23%, 20.10%, 6.70%, and (−3.61)% in R@1 compared to the WordNet, FP, QECK, and NQE extension methods, respectively. Additionally, QEIEF improved R@5 by 49.23%, 46.36%, 12.80%, and 7.06%, respectively, and R@10 by 47.64%, 42.23%, 17.06%, and 13.68%, respectively. In terms of MRR, QEIEF improved the metric by 50.93%, 50.00%, 9.63%, and 6.52%, respectively.

For the Java language in the UNIF model, QEIEF demonstrated improvements of 44.77%, 38.37%, 40.21%, and 35.39% in R@1 compared to the WordNet, FP, QECK, and NQE extension methods, respectively. Additionally, QEIEF improved R@5 by 42.78%, 38.24%,

Table 8
N value adjustment of Python.

Model	N	Python			
		R@1	R@5	R@10	MRR
	1	0.298	0.413	0.617	0.403
	2	0.304	0.477	0.647	0.429
DeepCS	3	**0.321**	**0.499**	0.657	**0.432**
	4	0.318	0.477	**0.673**	0.429
	5	0.301	0.409	0.601	0.407
	1	0.172	0.411	0.578	0.301
	2	0.184	0.436	0.588	0.318
UNIF	3	**0.194**	0.453	**0.592**	**0.322**
	4	0.182	**0.464**	0.576	0.312
	5	0.179	0.412	0.554	0.309

Table 9
N value adjustment of Java.

Model	N	Java			
		R@1	R@5	R@10	MRR
	1	0.297	0.411	0.612	0.413
	2	0.314	**0.506**	0.637	0.422
DeepCS	3	**0.321**	0.499	**0.657**	**0.432**
	4	0.279	0.401	0.612	0.412
	5	0.267	0.398	0.605	0.400
	1	0.359	0.684	0.796	0.498
	2	**0.373**	0.701	**0.817**	
UNIF	3	**0.373**	**0.706**	0.814	**0.519**
	4	0.367	0.699	0.809	0.509
	5	0.357	0.675	0.789	0.479

41.78%, and 39.95%, respectively, and R@10 by 37.47%, 28.75%, 35.14%, and 32.19%, respectively. In terms of MRR, QEIEF improved the metric by 37.57%, 27.55%, 42.00%, and 39.88%, respectively.

Based on the comparative experimental results, we can draw the following three conclusions:

(1) Among the five extension models, the DeepCS baseline model yields better results than UNIF. This can be attributed to DeepCS's utilization of sequence embedding to parse source code into method names, APIs, tokens, and description, which effectively captures the features of the source code. Additionally, employing a deep learning model for feature extraction enhances the accuracy of code search.

(2) The results of the five extension models, based on both the DeepCS and UNIF baseline models, are better on the Python dataset compared to the Java dataset. This suggests that the features present in the Python dataset are more suitable for feature extraction by the DeepCS and UNIF baseline models. Furthermore, the disparity between query and code description is more pronounced in the Python dataset. Consequently, query expansion proves effective in reducing the disparity between query and code description, thus improving the accuracy of code search.

(3) Among the four query expansion models of the baseline model, with the exception of the NEQ model, theQEIEF model outperforms the others. This is because theQEIEF model employs code comments from the source code developer as the source for expansion, effectively bridging the gap between query and code description. By selecting the code developer's comments as the extension source,QEIEF better captures the developer's intentions and aligns with the semantics of the source code. Moreover,QEIEF maintains the semantic and syntactic structure of the code comments during the expansion process, ensuring the accuracy of the extended data's feature information.

5.4. What is the impact of model parameters on QEIEF?

The query expansion process in QEIEF involves two key parameters: the number of intention-enhanced feedback (N) and the number of intentions (T). Through experiments, we obtained parameter tuning results corresponding to Tables 8–11.

From Tables 8–9, we observe that as the number of intention-enhanced feedback (N) gradually increases from 1 to 5, specific patterns emerge. In the DeepCS model, the optimal results for $R@10$ in the Python language and $R@5$ in the Java language are achieved when N is 4 and 2, respectively. For all other cases, the optimal results occur when N is 3. In the UNIF model, the optimal results for $R@5$ in Python and $R@10$ in Java are obtained when N is 4 and 2, respectively, while the remaining cases have the best results when N is 3. Based on these findings, we select the top 3 ($N = 3$) ranked query results as input for intention-enhanced feedback in our model.

From Tables 10–11, we can observe that as the number of intention-enhanced feedback (T) gradually increases from 1 to 5, certain patterns emerge. In the DeepCS model, the optimal results for $R@5$ in the Python language and $R@1$ in the Java language are achieved when

Table 10
T value adjustment of Python.

Model	T	Python			
		R@1	R@5	R@10	MRR
	1	0.281	0.478	0.634	0.421
	2	**0.321**	0.499	**0.657**	**0.432**
DeepCS	3	0.316	0.487	0.614	0.428
	4	0.311	**0.501**	0.602	0.419
	5	0.269	0.401	0.600	0.403
	1	0.189	0.446	0.584	0.319
	2	**0.194**	**0.453**	**0.592**	**0.322**
UNIF	3	0.190	0.451	0.591	**0.322**
	4	0.174	0.398	0.578	0.314
	5	0.171	0.399	0.580	0.311

Table 11
T value adjustment of Java.

Model	T	Java			
		R@1	R@5	R@10	MRR
	1	0.319	0.485	0.612	0.412
	2	0.321	**0.499**	**0.657**	**0.432**
DeepCS	3	**0.325**	0.491	0.635	**0.432**
	4	0.314	0.476	0.611	0.421
	5	0.301	0.402	0.617	0.411
	1	0.354	0.698	0.798	0.502
	2	**0.373**	0.706	**0.814**	**0.519**
UNIF	3	0.369	0.701	0.810	0.513
	4	0.354	**0.711**	0.801	0.496
	5	0.359	0.698	0.798	0.503

T is 4 and 3, respectively. For all other cases, the optimal results occur when T is 2. In the UNIF model, the optimal results for MRR in Python and $R@5$ in Java are obtained when T is 3 and 4, respectively, while the remaining cases have the best results when T is 2. Based on these findings, we select a number of feedback cycles of 2 ($T = 2$) for intention-enhanced feedback in our model.

5.5. How well doesQEIEF fit in pre-trained models

In this section, we delve into the performance ofQEIEF with respect to pre-trained models. The original pre-trained model code, known as CodeBERT, was introduced by Feng et al. [15]. CodeBERT is based on the BERT model and presents a pre-training approach for both natural language and programming languages (NL-PL). It was pre-trained on six languages, including Ruby, JavaScript, Go, Python, Java, and PHP, using the CodeSearchNet dataset. CodeBERT opened up a new avenue in code research and has gained significant popularity, accumulating over 1100 citations to date. Based on CodeBERT, Guo et al. [16] proposed GraphCodeBERT, which further incorporates semantic information of code by employing data flows instead of the AST syntax structure information utilized by CodeBERT. GraphCodeBERT outperformed CodeBERT in various tasks such as code search,

Table 12
Pre-trained model extension results.

Method	Python			
	R@1	R@5	R@10	MRR
Query	0.197	231	0.284	0.261
WordNet	0.113	0.184	0.213	0.170
FP	0.159	0.174	0.195	0.218
QECK	0.135	0.162	0.189	0.196
NQE	0.162	0.223	0.267	0.213
QEIEF	0.174	0.229	0.281	0.239
Method	**Java**			
	R@1	R@5	R@10	MRR
Query	0.224	0.277	0.311	0.301
WordNet	0.174	0.195	0.224	0.214
FP	0.186	0.201	0.234	0.268
QECK	0.179	0.199	0.214	0.224
NQE	0.201	0.224	0.265	0.274
QEIEF	0.210	0.254	0.289	0.289

clone detection, code translation, and code refinement. As a result, GraphCodeBERT has gained wide adoption and has been cited over 400 times. Considering the quality and relevance of the model, we selected GraphCodeBERT as the foundational model for our experiments and evaluated the performance ofQEIEF on pre-trained models.

To gain deeper insights into the performance of our proposedQEIEF utilizing the GraphCodeBERT pre-trained model, we conducted code search experiments. These experiments involved employing theQEIEF extension method on the CodesearchQE dataset we constructed, with GraphCodeBERT serving as the underlying model. The results of these experiments are presented in Table 12.

The "Query" column in the table represents the results of the GraphCodeBERT base model without theQEIEF extension. Based on the experimental results, GraphCodeBERT exhibits the best performance on the CodesearchQE dataset. However, when applying theQEIEF proposed in the paper and the four comparative extension methods to the GraphCodeBERT base model, all of them resulted in a decrease in accuracy compared to the base model. These experimental findings suggest that query expansion is not suitable for the pre-trained model GraphCodeBERT. We have identified two potential reasons for this observation:

(1) Input Pair Mismatch: The base models, CodeBERT and GraphCodeBERT, utilize input pairs of [Description-Code] for embedding. During model training, the Description and Code are considered as a unified entity and trained to obtain a combined score through feature extraction. In the context of the CodesearchQE dataset and the pre-trained model, QEIEF expands the Description into the Query. The expanded Query needs to form an input pair of [Query-Code] with the Code, necessitating the re-learning of the mapping relationship between the Query and Code. As the expanded Query significantly differs from the original Query, it induces substantial changes in the model parameters. Consequently, each expansion introduces parameter variations, leading to unstable model parameters.

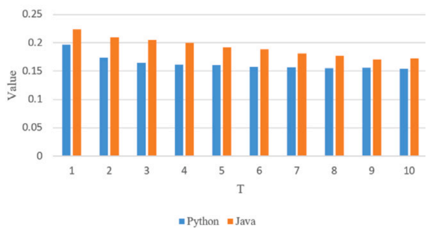
(2) Limited Dataset Size: The CodesearchQE dataset constructed for the experiments consists of a limited amount of data, comprising a total of 68,531 data pairs (Query-Code-Description). Pre-trained models like CodeBERT and GraphCodeBERT typically require large datasets to effectively train their model parameters. Compared to deep learning networks, pre-trained models rely on a more substantial amount of data for training. Given the limited size of the CodesearchQE dataset, it may not meet the data requirements necessary for training pre-trained models such as CodeBERT and GraphCodeBERT. Consequently, our constructed CodesearchQE dataset currently cannot adequately adapt to pre-trained models like CodeBERT and GraphCodeBERT.

To provide additional evidence supporting the notion that query expansion is not well-suited for pre-trained models like CodeBERT and GraphCodeBERT, we conducted experiments on the intent enhancement feedback times (T) using the proposedQEIEF. The experimental results are visually depicted in Figs. 12–13.

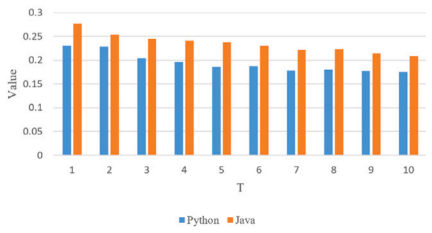
From the experimental results, it is evident that for the pre-trained model GraphCodeBERT, an increase in the number of feedback enhancement iterations leads to a deterioration in the performance of theQEIEF model. Consequently, based on the CodesearchQE dataset, pre-trained models such as CodeBERT and GraphCodeBERT are deemed unsuitable for the proposed expansion model, QEIEF, as outlined in this paper.

6. Related work

Initial research on code search was mainly focused on information retrieval techniques [17], which simply treated code language as natural language, ignoring the semantic divide between code language and natural language [18]. With Gu et al. [5] introducing deep learning into the study of code search for the first time, deep learning networks were used to embed code language and natural language into the same vector space, largely reducing the semantic divide between them [19]. Based on the research of Gu et al. deep learning has been extensively studied for code search applications. On the one hand, researchers develop datasets adapted to codes, and the validity of the data directly affects the accuracy of code search models [20]. For example, Yan S et al. [21] built CosBench, a dataset consisting of 1000 items, 52 code independent datasets. sun Z et al. [22] analyzed the impact of high quality datasets on code search studies and contributed a high quality dataset adapted to code search. On the other hand, researchers have mainly based their research on code search models. For example, Rahman M M et al. [23] used keywords from Stack Overflow's Q&A to associate with APIs and then reconstructed query for query statements. Liu C et al. [24] proposed a proposed deep learning model CodeMatcher based on the complexity of DeepCS model, incorporating IR techniques. Hu F et al. [25] studied based on long code properties and proposed an adapted model MLCS (Modeling Long Code for Code Search) for long code search. Yu H et al. [26] based on structure and quality of deep code search and introduced a code representation of program slicing to represent structural information as well as code fragments. Reen W et al. [27] and Haldar R et al. [28] characterize source code from both global and local search perspectives, and thus improve the effectiveness of code search. Hu G et al. [29] and Rao N et al. [30] introduce supervised learning mechanisms based on deep learning to improve the accuracy of code feature information characterization. Huang J et al. [31] and Li H et al. [32] used data augmentation techniques for enhanced characterization of source code data in order to improve the accuracy of data characterization. With the continuous research of code search, various models were born by application, such as graph neural networks [33], which were introduced to mainly characterize source code as graphs to improve the contextual semantic information of source code [34,35]. In the continuous research of models, researchers found that a single model cannot accurately characterize code information [36], and multi-model characterization is better to characterize feature information [37]. Du L et al. [38] and Shi E et al. [39] used multimodal characterization of codes to improve the accuracy between codes and query. In the course of the study, researchers found that different feature information has different weights in the process of characterization [10,40,41], therefore, researchers introduced attention models for weighting analysis. For example, Fang S et al. [42] used a self-attention model to characterize the weight relationship between data and data based on the study of Gu et al. [5], which largely improved the accuracy of code characterization. Based on more than the above studies, there are more research methods for code search. For example, annotation approaches [43], Code2Vec [44], equivalence inference [45], path characterization [46,47], fast-slow models [48], binary characterization [49], encoders [50], syntax tree

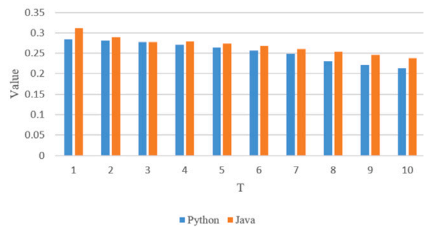


(a) Effect of T-value on R@1

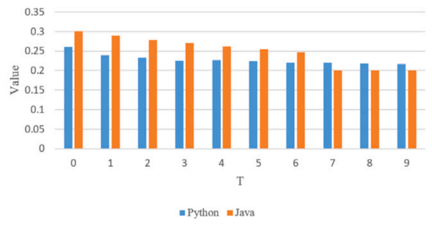


(b) Effect of T-value on R@5

Fig. 12. Effect of T-value 1.



(a) Effect of T-value on R@10



(b) Effect of T-value on MRR

Fig. 13. Effect of T-value 2.

matching [51], bag-of-words models [52], two-reinforcement learning [53], context-aware machines [54], hashing algorithms [55], information fusion techniques [56], function clarification [57], relevance matching and semantic binding [58] contextual structural information [59], meta-learning [60], function inference [61], and extended compression methods [62].

With the rapid development of code search models, the accuracy of the models has been greatly improved [63]. However, code search models all have certain adaptation limitations and the effectiveness of the models is mainly based on fixed data sets. Therefore, Ling C et al. [64] proposed an AdaCS model that adapts the model to new code bases. There are also researchers studying cross-model [65] studies and cross-language [66] studies of code search. With Feng Z et al. [15] introduced BERT into the preprocessing study of code search for the first time and proposed the CodeBERT pretraining model. The pre-training model has been developed rapidly and also improved the accuracy of code search to a great extent [67]. On the research of BERT pre-training model, the core of BERT model, migration learning [68–70] has been widely used in code search.

7. Threats to validity

To ensure a comprehensive discussion of the validity threats in research papers, it is essential to address both internal validity threats and external validity threats. Let us examine each of these threats and explore potential measures to mitigate them.

7.1. Internal validity threats

Selection Validity: In the process of experimental design and participant selection, the potential threat of selection bias can arise, leading to a lack of representativeness in the sample and potentially affecting the generalizability of study results. To mitigate the threat of chance in participant selection, the paper employed a strategy of averaging results from five independent experiments during the evaluation process. This approach helps reduce the influence of random variations and provides a more reliable estimate of the overall performance.

Data Validity: The lack of a standardized research dataset for query expansion studies can introduce data validity threats, as each researcher typically creates their own dataset. Differences in datasets can impact the model's training effectiveness and subsequently affect the accuracy of code search results. To address this threat, the paper utilized existing code search research datasets and crawled data from Q&A repositories in both Python and Java languages. Furthermore, to ensure the dataset's suitability for query expansion research, the paper incorporated descriptors from program developers, search requesters, and discussants as the expansion source for query statement expansion during the dataset construction. This expansion source is more aligned with real-world code search tasks, thereby enhancing the accuracy of code search results.

7.2. External validity threats

Generalization Validity: The paper acknowledges a potential limitation in generalizability due to the study's reliance on data from only two programming languages, Python and Java. The inability to verify the effects in other programming languages restricts the paper's ability to generalize the findings of the query expansion for code search. To mitigate this limitation, future work will focus on expanding the dataset in two ways. Firstly, by increasing the dataset's capacity, and secondly, by extending the dataset to include multiple programming languages, thus enabling the evaluation of the proposed approach across a broader range of languages. This expansion will enhance the generalizability of the study and facilitate the assessment of cross-linguistic effects.

Environmental Validity: The experimental environment employed in the paper's research may differ from real-world development environments, which could potentially limit the practical applicability of the research results. The suboptimal performance and limited capacity of the server used in the experimental process have impacted the computational power and speed. To address this limitation, future work aims to identify a more suitable experimental platform for subsequent research, ensuring improved computational capabilities and efficiency. Additionally, to align the research more closely with actual code search tasks, efforts will be made to engage with real enterprises or organizations to conduct the research, thereby enhancing the environmental validity of the study.

8. Conclusion

The paper presents a study focusing on code search accuracy, introduces a code search model centered on intent-enhanced query expansion, and validates the efficacy of this model in code search tasks through experimental analysis. This research marks the pioneering exploration of the querier's intention, offering a novel perspective and theoretical foundation for code search query expansion research. The findings of this study hold significant importance for advancing future research in the field of code search. The study draws the following conclusions:

(1) Traditional code search methods commonly treat code descriptions as natural language during training and queries during searches, often overlooking the distinctions between queries and code descriptions.

(2) The developed ternary dataset can be leveraged for studies on code search query expansion.

(3) The proposed QEIEF query expansion model demonstrates enhanced expansion accuracy, effectively boosting the precision of code search tasks.

The paper acknowledges certain limitations and outlines future directions for improvement and expansion in the following areas:

(1) In terms of the CodesearchQE dataset construction, the current limitations in working time have restricted the number of datasets that could be constructed. To address this limitation, future work will focus on expanding the CodesearchQE datasets, likely by creating additional datasets to enhance the scope and diversity of the training and evaluation data.

(2) The lack of open-source code for many extended models prevents a comprehensive comparison and analysis with such models. To address this, the next research efforts will involve the development of these models and conducting comparative evaluations with the QEIEF model proposed in the paper. This will facilitate a more thorough understanding of the strengths and weaknesses of different approaches.

Ethical and informed consent for data used

The data used in the paper was partly collected by myself, while the other part was sourced from open source libraries, and there were no violations of ethical standards.

CRediT authorship contribution statement

Haize Hu: Writing – original draft, Software, Data curation. **Mengge Fang:** Writing – review & editing, Resources. **Jianxun Liu:** Methodology.

Declaration of competing interest

No conflict of interest exists in the submission of this manuscript, and manuscript is approved by all authors for publication. I would like to declare on behalf of my co-authors that the work described was original research that has not been published previously, and not under consideration for publication elsewhere, in whole or in part. All the authors listed have approved the manuscript that is enclosed.

Data availability

Data will be made available on request.

Acknowledgments

This work was supported by the Natural Science Foundation of China, under grants 61866013.

References

[1] H. Niu, I. Keivanloo, Y. Zou, Learning to rank code examples for code search engines, Empir. Softw. Eng. 22 (1) (2017) 259–291.
[2] C. Liu, X. Xia, D. Lo, et al., Opportunities and challenges in code search tools, ACM Comput. Surv. 54 (9) (2021) 1–40.
[3] J. Lin, X. Ma, S.C. Lin, et al., Pyserini: A python toolkit for reproducible information retrieval research with sparse and dense representations, in: Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, 2021, pp. 2356–2362.
[4] J.A. Khan, Comparative study of information retrieval models used in search engine, in: 2014 International Conference on Advances in Engineering & Technology Research, ICAETR-2014, IEEE, 2014, pp. 1–5.
[5] X. Gu, H. Zhang, S. Kim, Deep code search, in: 2018 IEEE/ACM 40th International Conference on Software Engineering, ICSE, IEEE, 2018, pp. 933–944.
[6] X. Ge, D.C. Shepherd, K. Damevski, et al., Design and evaluation of a multi-recommendation system for local code search, J. Vis. Lang. Comput. 39 (2017) 1–9.
[7] Y. Yang, Q. Huang, IECS: Intent-enforced code search via extended boolean model, J. Intell. Fuzzy Systems 33 (4) (2017) 2565–2576.
[8] Q. Huang, X. Wang, Y. Yang, et al., SnippetGen: Enhancing the code search via intent predicting, SEKE (2017) 307–312.
[9] F. Zhang, H. Niu, I. Keivanloo, et al., Expanding query for code search using semantically related api class-names, IEEE Trans. Softw. Eng. 44 (11) (2017) 1070–1082.
[10] L. Xu, H. Yang, C. Liu, et al., 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, 2021, pp. 342–353.
[11] J. Cambronero, H. Li, S. Kim, et al., 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, 2019, pp. 964–974.
[12] H.K. Azad, A. Deepak, A new approach for query expansion using wikipedia and WordNet, Inf. Sci. 492 (2019) 147–163.
[13] L. Nie, H. Jiang, Z. Ren, et al., Query expansion based on crowd knowledge for code search, IEEE Trans. Serv. Comput. 9 (5) (2016) 771–783.
[14] J. Liu, S. Kim, V. Murali, et al., Neural query expansion for code search, in: Proceedings of the 3rd Acm Sigplan in-Ternational Workshop on Machine Learning and Programming Languages, 2019, pp. 29–37.
[15] Z. Feng, D. Guo, D. Tang, et al., Codebert: A pre-trained model for programming and natural languages, 2020, arXiv preprint arXiv:2002.08155.
[16] D. Guo, S. Ren, S. Lu, et al., Graphcodebert: Pre-training code representations with data flow, 2020, arXiv preprint arXiv:2009.08366.
[17] Z. Deng, L. Xu, C. Liu, et al., Code semantic enrichment for deep code search, J. Syst. Softw. 207 (2024) 111856.
[18] M.M. Rahman, C.K. Roy, A systematic literature review of automated query reformulations in source code search, 2021, arXiv preprint arXiv:2108.09646.
[19] K. Kim, Steps Towards Semantic Code Search, University of Luxembourg, Luxembourg, Luxembourg, 2021.
[20] H. Husain, H.H. Wu, T. Gazit, et al., Codesearchnet challenge: Evaluating the state of semantic code search, 2019, arXiv preprint arXiv:1909.09436.
[21] S. Yan, H. Yu, Y. Chen, et al., Are the code snippets what we are searching for? A benchmark and an empirical study on code search with natural-language query, in: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering, SANER, IEEE, 2020, pp. 344–354.
[22] Z. Sun, L. Li, Y. Liu, et al., On the importance of building high-quality training datasets for neural code search, in: Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 1609–1620.
[23] P. Salza, C. Schwizer, J. Gu, et al., On the effectiveness of transfer learning for code search, IEEE Trans. Softw. Eng. (2022).
[24] C. Liu, X. Xia, D. Lo, et al., Simplifying deep-learning-based model for code search, 2020, arXiv preprint arXiv:2005.14373.
[25] F. Hu, Y. Wang, L. Du, et al., Long code for code search, 2022, arXiv preprint arXiv:2208.11271.
[26] H. Yu, Y. Zhang, Y. Zhao, et al., Incorporating code structure and quality in deep code search, Appl. Sci. 12 (4) (2022) 2051.
[27] W. Ren, H. Zhang, Q. Liu, et al., Greedy code search based memetic algorithm for the design of orthogonal polyphase code sets, IEEE Access 7 (2019) 13561–13576.
[28] R. Haldar, L. Wu, J. Xiong, et al., A multi-perspective architecture for semantic code search, 2020, arXiv preprint arXiv:2005.06980.
[29] G. Hu, M. Peng, Y. Zhang, et al., Unsupervised software repositories mining and its application to code search, Softw. - Pract. Exp. 50 (3) (2020) 299–322.
[30] N. Rao, C. Bansal, J. Guan, Search4Code: Code search intent classification using weak supervision, in: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories, MSR, IEEE, 2021, pp. 575–579.
[31] J. Huang, D. Tang, L. Shou, et al., Cosqa: 20 000+ web query for code search and question answering, 2021, arXiv preprint arXiv:2105.13239.

[32] H. Li, C. Miao, C. Leung, et al., Exploring representation-level augmentation for code search, 2022, arXiv preprint arXiv:2210.12285.

[33] S. Liu, X. Xie, L. Ma, et al., Graphsearchnet: Enhancing gnns via capturing global dependency for semantic code search, 2021, arXiv preprint arXiv:2111.02671.

[34] C. Shi, Y. Xiang, J. Yu, et al., Semantic code search for smart contracts, 2021, arXiv preprint arXiv:2111.14139.

[35] X. Zhang, J. Xin, A. Yates, et al., Bag-of-words baselines for semantic code search, in: 1st Workshop on Natural Language Processing for Programming, ACL, 2021, pp. 88–94.

[36] J. Gu, Z. Chen, M. Monperrus, Multimodal representation for neural code search, in: 2021 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2021, pp. 483–494.

[37] C. Wu, M. Yan, Learning deep semantic model for code search using CodeSearchNet corpus, 2022, arXiv preprint arXiv:2201.11313.

[38] L. Du, X. Shi, Y. Wang, et al., Is a single model enough? mucos: A multi-model ensemble learning for semantic code search, 2021, arXiv preprint arXiv:2107.04773.

[39] E. Shi, W. Gub, Y. Wang, et al., Enhancing semantic code search with multimodal contrastive learning and soft data augmentation, 2022, arXiv preprint arXiv:2204.03293.

[40] J. Shuai, L. Xu, C. Liu, et al., Improving code search with co-attentive representation learning, in: Proceedings of the 28th International Conference on Program Comprehension, 2020, pp. 196–207.

[41] G. Hu, M. Peng, Y. Zhang, et al., Neural joint attention code search over structure embeddings for software Q&A sites, J. Syst. Softw. 170 (2020) 110773.

[42] S. Fang, Y.S. Tan, T. Zhang, et al., Self-attention networks for code search, Inf. Softw. Technol. 134 (2021) 106542.

[43] G. Heyman, T.Van. Cutsem, Neural code search revisited: Enhancing code snippet retrieval through natural language intent, 2020, arXiv preprint arXiv:2008.12193.

[44] L. Arumugam, Semantic Code Search using Code2Vec: A Bag-of-Paths Model, University of Waterloo, 2020.

[45] V. Premtoon, J. Koppel, A. Solar-Lezama, Semantic code search via equational reasoning, in: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, 2020, pp. 1066–1082.

[46] Z. Sun, Y. Liu, C. Yang, et al., PSCS: A path-based neural model for semantic code search, 2020, arXiv preprint arXiv:2008.03042.

[47] L. Arumugam, Semantic Code Search using Code2Vec: A Bag-of-Paths Model, University of Waterloo, 2020.

[48] A.D. Gotmare, J. Li, S. Joty, et al., Cascaded fast and slow models for efficient semantic code search, 2021, arXiv preprint arXiv:2110.07811.

[49] J. Yang, C. Fu, X.Y. Liu, et al., Codee: a tensor embedding scheme for binary code search, IEEE Trans. Softw. Eng. (2021).

[50] Y. Meng, An intelligent code search approach using hybrid encoders, Wirel. Commun. Mob. Comput. 2021 (2021).

[51] B. Taskaya, Reiz: Structural source code search, J. Open Sour. Softw. 6 (62) (2021) 3296.

[52] X. Zhang, J. Xin, A. Yates, et al., Bag-of-words baselines for semantic code search, in: 1st Workshop on Natural Language Processing for Programming, ACL, 2021, pp. 88–94.

[53] C. Wang, Z. Nong, C. Gao, et al., Enriching query semantics for code search with reinforcement learning, Neural Netw. 145 (2022) 22–32.

[54] W. Sun, C. Fang, Y. Chen, et al., Code search based on context-aware code translation, 2022, arXiv preprint arXiv:2202.08029.

[55] W. Gu, Y. Wang, L. Du, et al., Accelerating code search with deep hashing and code classification, 2022, arXiv preprint arXiv:2203.15287.

[56] F. Hu, Y. Wang, L. Du, et al., Revisiting code search in a two-stage paradigm, 2022, arXiv preprint arXiv:2208.11274.

[57] Z. Eberhart, C. McMillan, Generating clarifying questions for query refinement in source code search, 2022, arXiv preprint arXiv:2201.09974.

[58] Y. Cheng, L. Kuang, CSRS: Code search with relevance matching and semantic matching, 2022, arXiv preprint arXiv:2203.07736.

[59] S. Dahal, A. Maharana, M. Bansal, Scotch: A semantic code search engine for IDEs, in: Deep Learning for Code Workshop, 2022.

[60] Y. Chai, H. Zhang, B. Shen, et al., Cross-domain deep code search with few-shot meta learning, 2022, arXiv preprint arXiv:2201.00150.

[61] S. Arakelyan, A. Hakhverdyan, M. Allamanis, et al., NS3: Neuro-symbolic semantic code search, 2022, arXiv preprint arXiv:2205.10674.

[62] L. Gu, Z. Wang, J. Liu, et al., Expansion-compression learning: A deep learning code search method that simulates reading habits, in: 2022 26th International Conference on Engineering of Complex Computer Systems, ICECCS, IEEE, 2022, pp. 195–200.

[63] S.K. Shivakumar, A survey and taxonomy of intent-based code search, Int. J. Softw. Innov. (IJSI) 9 (1) (2021) 69–110.

[64] C. Ling, Z. Lin, Y. Zou, et al., Adaptive deep code search, in: Proceedings of the 28th International Conference on Program Comprehension, 2020, pp. 48–59.

[65] Z. Shi, Y. Xiong, X. Zhang, et al., Cross-modal contrastive learning for code search, in: 2022 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2022, pp. 94–105.

[66] G.V. Mathew, Cross-Language Code Similarity and Applications in Clone Detection and Code Search, North Carolina State University, 2022.

[67] N. Rao, C. Bansal, J. Guan, Search4Code: Code search intent classification using weak supervision, in: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories, MSR, IEEE, 2021, pp. 575–579.

[68] P. Salza, C. Schwizer, J. Gu, et al., On the effectiveness of transfer learning for code search, IEEE Trans. Softw. Eng. (2022).

[69] Y. Peng, J. Xie, G. Hu, et al., Transformer-based code search for software Q&A sites, J. Softw.: Evol. Process (2022) e2517.

[70] E. Papathomas, T. Diamantopoulos, A. Symeonidis, Semantic code search in software repositories using neural machine translation, in: International Conference on Fundamental Approaches To Software Engineering, Springer, Cham, 2022, pp. 225–244.