Enrich Code Search Query Semantics with Raw Descriptions

Xiangzheng Liu[1,2], Jianxun Liu[1,2(✉)], Haize Hu[1,2], and Yi Liu[1,2]

[1] School of Computer Science and Engineering, Hunan University of Science and Technology, Xiangtan, Hunan, China
ljx0934@mail.hnust.edu.cn

[2] Hunan Provincial Key Laboratory for Services Computing and Novel Software Technology, Hunan University of Science and Technology, Xiangtan, Hunan, China

Abstract. Code search can recommend relevant source code according to the development intention (query statement) of the demander, thereby improving the efficiency of software development. In the research of deep code search model, code description is used to replace query sentences for training. However, the heterogeneity existing between the query statement and the code description will seriously affect the accuracy of the code search model. In order to make up for the shortcomings of code search, this paper proposes a sentence-integrated query expansion method—SIQE. Unlike previous query expansion methods that focus on word-level expansion, SIQE uses the entire code description fragment as the source of query expansion. And by learning the mapping relationship between the query statement and the code description, the heterogeneity problem between them is compensated. In order to verify the effect of the proposed model in code search tasks, the article conducts code search experiments and analyzes on two languages: python and java. Experimental results show that, compared with the baseline model, SIQE has higher code search results. Therefore, the SIQE model can effectively improve the search effect of query statements, improve the accuracy of code search, and further improve the development of software engineering.

Keywords: Code search · Query expansion · Software engineering · Deep learning

1 Introduction

In the software development process, a significant part of the work involves writing functional code fragments that are repeatedly written and used in various development projects. Code search technology helps search for functional code fragments based on their code function descriptions. In the early days of code search, information retrieval methods (IR) based on keyword rule matching were used to search for relevant code fragments. Such as [13,16,20,22,25]. However, these methods heavily rely on the repetition of words between the code and the

H. Gao et al. (Eds.): CollaborateCom 2023, LNICST 561, pp. 284–302, 2024.
https://doi.org/10.1007/978-3-031-54521-4_16

query, leading to search bias when different words are used to represent the same query. In recent years, deep learning (DL) has been applied in code search, such as [2,3,6,15,33]. These DL-based methods jointly embeds code fragments and code descriptions into a high-dimensional vector space. Since obtaining natural language queries is not always feasible, code descriptions are used instead. The similarity between vectors is used to represent the matching degree of code fragments and code descriptions. These methods can learn the semantic relationships between code fragments and code descriptions, enabling the identification of semantically similar words.

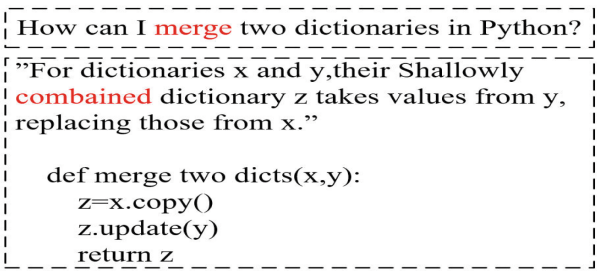

Fig. 1. A example of qurey-description-code

Although deep learning-based code search takes into account the semantic relationship between code fragments and code descriptions, there are still gaps between the query statement and the code description in practical applications. As shown in Fig. 1, there are two main issues: First, the query statement, such as "How can I merge two dictionaries in Python" is generally shorter than the code description, which may not fully express the queryer's intention. Second, a large number of code descriptions use domain-specific terms that may differ from the queryer's expression. For example, "merge" and "combine" may have similar meanings in the domain but are expressed differently in the query statement. These gaps lead to heterogeneity between the query and description, which can significantly reduce the performance of the original model. To address this issue, researchers have explored various query expansion methods [10,17,19,24,28,30]. For instance: Work [19] used WordNet (a synonym database) to enrich the query statement with keywords from the query. Work [28] used WordSim, a synonym prediction library trained on code search datasets, to reduce the impact of noise in WordNet and expand queries. Work [24] crawled the question-and-answer prediction library on Stack Overflow, extracted meaningful word pairs from it, and automatically expanded the query. Work [30] used reinforcement learning to reconstruct the query statement, where the performance of code search was used as the reward of the reconstruction model. The reconstructed query was then used to perform the search task. By expanding the query statement, these methods aim to bridge the gap between the query and description and improve the accuracy of query results.

These query expansion methods mentioned above can to some extent enrich the semantic information of the query and improve the accuracy of the search. However, there are still some issues that need to be addressed. Firstly, irrelevant word noise may be added in the process of enriching the query. Secondly, the current query expansion methods mainly focus on the expansion of words, without considering the differences in sentence expression between the code describer and the queryer. To tackle these issues, we propose a new approach called Sentence-Integrated Query Expansion (SIQE), which aims to enrich the query sentence and improve its accuracy in code search.

SIQE comprises two main components: the Description Search Model (DS) and the Ensemble Search (ES). The DS model is trained to find the k most semantically relevant code description fragments given an input query statement. The ES component integrates the sentence information from these k description fragments with the original query, and sends the expanded query to the Code Search Model (CS) for improved search accuracy. To reduce the impact of noise, we weight the extended sentence based on its similarity score with the original sentence. Since there is currently no publicly available dataset containing query, code description, and code snippets, we crawled a dataset from Stack Overflow to demonstrate the effectiveness of our approach in experiments. The dataset is extracted from the 22,546 most active Java tag questions and 19,276 most active Python tag questions on Stack Overflow, and it includes query, a code description, and code snippet triplet dataset.

This article makes the following contributions:

- We have prepared a dataset of triples, including query, code description, and code snippet. We used this dataset to verify the gap between searching using query statements and code descriptions.
- We propose the Sentence Integrated Query Expansion method (SIQE). We are the first to recognize the importance of sentence patterns in query expansion, and we add sentence pattern information to the query to improve its accuracy. To reduce the noise impact of expanding useless words, we propose a method of integrating the expanded sentences with weights according to the similarity score between the expanded sentences and the original sentence, thereby improving the accuracy of the search.
- We have conducted extensive experiments on different languages and different code search models, demonstrating the advantages of our method (SIQE). Furthermore, we have made our code and dataset publicly available.

The article is structured as follows: Sect. 2 introduces our method. Section 3 describes our dataset, experimental evaluation indicators, and experimental construction. Section 4 presents the comparative results of the experiments and the conclusions drawn. Section 5 introduces related work. Section 6 summarizes the paper, discusses the limitations of our method, and outlines future prospects.

2 Methodology

The overall structure of Sentence Integrated Query Expansion (SIQE) is illustrated in Fig. 3, and it consists of three main components: 1. Code Search Model (CS) 2. Description Search Model (DS) 3. Ensemble Search (ES) First, the CS and DS are trained separately using the code-description data and query-description data, respectively. In the ES stage, when a new query statement is presented, the DS provides k code description fragments with the most similar semantics. These k fragments are used to extend the original query sentence structure and are embedded into a vector through CS along with the original query. To reduce the noise impact of expanding useless words, each of the k vectors is assigned a weight, which is the similarity score between the vector and the original query vector. Finally, the k + 1 vectors are integrated using the parallelogram law of vector spaces to obtain the final query vector, which is used to search for the most similar code fragment in the code space.

2.1 Code Search (CS)

The purpose of the Code Search Model (CS) is to develop a suitable method for embedding natural language and code into vectors. Since this work focuses on query expansion, we utilize the existing code search models DeepCS and UNIF [1,6]. These CS models typically consist of two embedding network parts, which are typically composed of Long Short-Term Memory (LSTM) or Convolutional Neural Networks (CNN). The first network embeds the code description (d), and the second network embeds the code (c). The matching degree of the (d, c) pair is determined by calculating the cosine similarity of the two vectors. After calculating the cosine similarity with all codes in the database, the code with the highest similarity score is selected as the search result.

The goal of the CS is to maximize the cosine similarity between the correct code and the original query during training, while minimizing the cosine similarity between the incorrect code and the original query. Specifically, for each sample $(d, c+, c-)$ where d is the code description, $c+$ is the correct code fragment, and c− is the incorrect code fragment randomly selected from the code database, the loss function can be defined as formula 1.

$$L(\theta)_{CS} = \sum_{(D,C+,C-)} max(0, \varepsilon - sim(d, c+) + sim(d, c-)) \tag{1}$$

where θ represents the parameters of the code search model, d is the code description vector, $c+$ is the vector of the correct code fragment after embedding, $c-$ is the vector of the incorrect code fragment after embedding, ε is a hyperparameter that adjusts the minimum tolerance range of the model for the difference between the cosine similarity scores of d and c+ and c−. This parameter determines whether a sample needs to be added to the overall loss function based on whether the difference between the similarity score calculated by the model for c+ and the similarity score calculated by c− is greater than ε. The *sim* function

calculates the cosine similarity between two vectors. The goal of the CS is to maximize the cosine similarity between the correct code and the original query during training and minimize the cosine similarity between the wrong code and the original query.

2.2 Desc Search (DS)

The Description Search Model (DS) aims to find the k code description fragments that are semantically similar to a given natural language query in the code description database. These k code descriptions are then used to expand the sentences and semantics of the original query. This approach enriches the semantics of the original query and also considers the differences in expression between the code description writing method and the natural language query writing method.

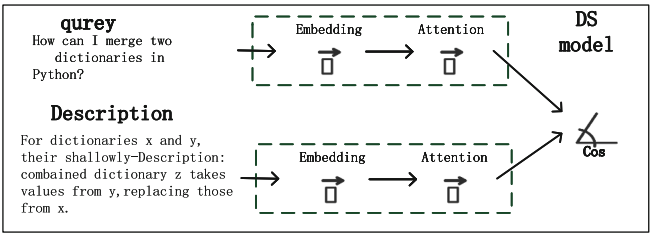

Fig. 2. DS model structure

DS Model. The Description Search Model (DS) is similar to the Code Search Model (CS) in that it consists of two neural network models that encode natural language inputs and calculate similarity scores. In the case of DS, the goal is to identify the k code description fragments that are most semantically similar to a given natural language query. The DS model is shown in Fig. 2. Since both the query and code description belong to natural language, the attention mechanism is used to build the embedding network. The attention mechanism [29] is a popular structure in the field of natural language processing that has been shown to be effective for various tasks, such as machine translation and text classification.

Train DS Model. The purpose of DS training is to maximize the cosine similarity between the query and the correct code description, while minimizing the cosine similarity between the query and a wrong code description randomly selected from the code description database. The DS model uses the gradient descent method to obtain the optimal network parameters. Specifically, for each sample (q, d^+, d^-), where q is a natural language query, d^+ is a correct code

description fragment, and d^- is a wrong code description fragment randomly selected from the code database, the loss function can be defined as formula 2.

$$L(\theta)_{DS} = \sum_{(Q,D+,D-)} max(0, \varepsilon - sim(q, d+) + sim(q, d-)) \tag{2}$$

where θ represents the parameters of the DS model, $\text{sim}(q, d, \theta)$ is the cosine similarity between the query q and the code description d after being embedded by the DS model, and ε is a margin hyperparameter that controls the minimum difference between the similarity score of the correct and wrong code description. The loss function aims to make the similarity score between the query and the correct code description higher than that of the wrong code description by at least ε.

2.3 Ensemble Search

The purpose of Ensemble Search (ES) is to further reduce the impact of noise and enhance the accuracy of queries based on the extended k-code description fragments of the DS model. Specifically, given a natural language query q and k code description fragments $(d_1, d_2, ..., d_k)$, ES generates the final query expansion vector q^* using the formula 3.

$$q^* = q + \sum_{i=1}^{k} d_i \cdot score_{d_i}, \tag{3}$$

where q, d_1, and d_k are the vectors embedded by the code search model (CS), and $score_{d_k}$ is the cosine similarity score between d_k and q. ES reduces the noise impact of useless information in descriptions by adding cosine similarity weights. The final query expansion vector q^* is obtained by fusing the vectors using the parallelogram rule, and it is used to search for code fragments in the code database using the formula 4.

$$score(c, q) = sim(\underbrace{ES(\overbrace{d_1, \cdots, d_k}^{DS(q)}, q)}_{q^*}, c) \tag{4}$$

where sim represents the similarity between the query and code fragment, and $ES(DS(q), q)$ represents the final query vector q^* obtained using the k vectors recommended by DS.

3 Experimental Setup

3.1 Dataset

CodeSearchNetDataset (CSND). CSND [11] is a large code search corpus collected from the GitHub website and is the most commonly used database in

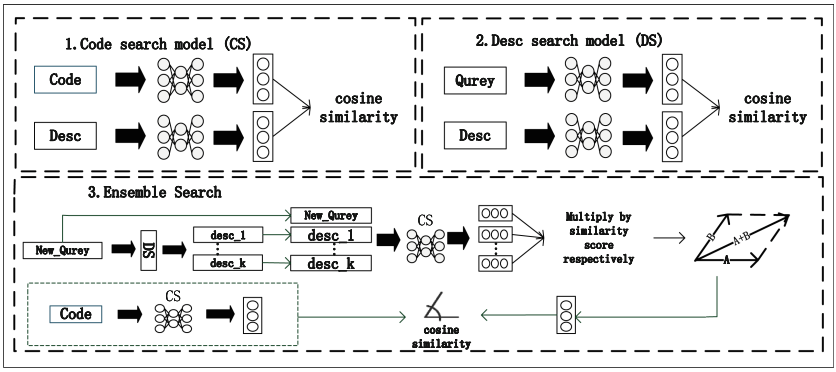

Fig. 3. SIQE framework

the field of code search, covering six languages: Go, Java, JavaScript, Python, Php, and Ruby. Each sample in the CSND corpus consists of source code and code description. To preprocess the Python and Java datasets in the corpus, we segment camel case text or text composed of underscores into words and convert them to lowercase. We then count the word frequencies and retain only the 10,000 words with the highest frequency of occurrence as the vocabulary. The corpus information in the preprocessed dataset is shown in Table 1. We train a Code Search model using the CSND dataset.

Although CSND is a large and widely used code search standard dataset, natural language queries are also essential for query expansion. Unfortunately, CSND does not include natural language queries, and currently, there is no open-source dataset that includes queries, code descriptions, and code snippets. To address this gap, we have collected such a dataset by crawling StackOverflow. StackOverflow is a community where software developers from around the world discuss problems and solutions. For each problem description raised by a software developer, other developers provide answers, often including code snippets, and explanations that combine the code snippets with the question. We use these explanations as the code description corpus and the code snippets as the code corpus.

For example, if a software developer raises a question "How do I merge two dictionaries in a single expression?", we use it as a natural language query corpus and then extract the comment "For dictionaries x and y, their shallowly-merged dictionary z takes values from y, replacing those from x." as a description of the code snippet. We also extract the code from the comment: "def merge_two_dicts(x, y): z = x.copy() z.update(y)". Finally, we obtain a data sample containing natural language queries, code descriptions, and code snippets, as shown in Fig. 1.

Since not every question contains code snippets, and comments often contain a lot of useless information, we took the following steps to improve the quality of the collected data:

– We filter out questions starting with keywords such as "what," which mainly include some discussion questions, such as "What are metaclasses in Python?"
– We only collect answers that the questioner has accepted and extract code descriptions and code snippets from them. If the question has no accepted answers, we discard it. If there is no code snippet in the answer or there is no commenting text, we also discard the question.

We extract data from the Python and Java directories in StackOverflow according to the above rules, then undergo the same preprocessing as CSND, and finally obtain our dataset. The dataset information is shown in Table 1.At the same time, we counted the token length of the query, description, and code after processing in the dataset, as shown in Fig. 4. It can be seen that the token length of the query is much less than that of the description. We trained Description Search (DS) model using the collected dataset. The results of the DS training are shown in the Table 2.

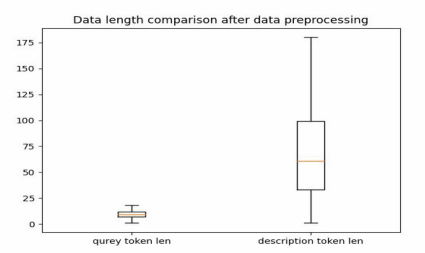

Fig. 4. Data length comparison after data preprocessing

Table 1. CodeSearchNet dataset compared to collected datasets

Dataset		train_set	test_set	avg_code_token_len	avg_desc_token_len	avg_qurey_token_len
CodeSearchNet	**java**	450941	26717	162.41	38.83	–
	python	409230	22104	167.51	54.21	–
Ours	**java**	21546	1000	88.9	72.56	9.67
	python	18276	1000	98.69	63.88	9.34

3.2 Evaluation Metric

We use two evaluation metrics to assess the performance of our code search model: *Recall@k* and Mean Reciprocal Rank (MRR). These metrics are defined as follows:

– *Recall@k* measures the proportion of correct code fragments among the top k code fragments retrieved by the model. It is calculated as shown in formula 5, where Q represents a sample in the test set, and I is the indicator function that returns 1 when the correct code fragment is among the top k retrieved fragments, and 0 otherwise. We report *Recall@k* values for $k = 1$, 5, and 10.

$$Recall@k = \frac{1}{|Q|}\sum_{i=1}^{|Q|} I(Q_i <= k) \tag{5}$$

– MRR calculates the mean reciprocal rank of the correct code fragment across the entire test set. It is computed as shown in formula 6, where Q represents a sample in the test set, and $Rank(Q_i)$ represents the rank of the correct code fragment returned by the model in the entire test set.

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|} Rank(Q_i) \tag{6}$$

Table 2. DS Search Model Results

	Python				Java			
	R@1	R@5	R@10	MRR	R@1	R@5	R@10	MRR
Valid	0.414	0.646	0.727	0.542	0.510	0.734	0.877	0.525
Test	0.331	0.597	0.698	0.50	0.331	0.597	0.698	0.438

3.3 Baselines

As a query expansion component for code search, SIQE can be used with any code search model. We primarily evaluated our method using two popular code search models: 1. DeepCS [6] is an LSTM-based (Long Short-Term Memory Neural Network) model that embeds text data and uses max-pooling to produce the final output. In addition to considering the text of the code, DeepCS also incorporates method names and API sequence information. Since the API information in the data cannot be extracted, the article modifies the DeepCS model to only incorporate method names in the code vector representation. 2. UNIF [1] is a lightweight code search model. For the query statement, the average pooling is directly used as the embedding vector after the Embedding operation, and the code is embedded through the attention mechanism. UNIF has very high performance and accuracy in the field of code search.

To demonstrate the effectiveness of our query expansion method, we compared it against two popular query expansion methods:

1. The WordNet [19] extension method, which uses the WordNet thesaurus to expand the query by adding synonyms of the query keywords. 2. The frequent itemsets (FP) expansion method, which first identifies frequent itemsets

in the code descriptions by computing their support, and then expands the query keywords using words corresponding to the frequent itemsets. We used the FP-growth algorithm [21] to identify frequent itemsets.

3.4 Implementation Details

To begin with, we tokenize the corpus in the dataset based on camel case and underscores. We then select the top 10,000 words from the corpus based on their frequency of occurrence, and encode the corpus according to this vocabulary. Any words that are not in the vocabulary are discarded. Finally, we divide the encoded data into training and test sets. For training the Description Search (DS) model, we use the code description and natural language query parts of the training data. For training the Code Search (CS) model, we use the code descriptions and code fragments of the training data. For each training sample, we add a corresponding negative sample for description search. After training the DS and CS models, we use the DS model to search for relevant code descriptions in the entire test set (excluding correct code description fragments) for a new query, and use this information as query expansion data. We integrate this with the query using the ES method and pass it to the CS model to return the ranking of code fragments. We evaluate the performance of the model using two metrics: Recall@k and MRR. Recall@k measures the proportion of correct code fragments among the first k code fragments returned by the code search model. MRR calculates the reciprocal of the rank of the fragments returned by the code search model in the entire test set. We compared our query expansion method with two popular query expansion methods: WordNet extension and frequent itemset-based expansion. We use the FPgrowth algorithm to construct frequent itemsets. We trained the DS model for 200 epochs with a learning rate of 1e−3 and a learning rate decay ratio of 0.5. The settings of the CS model are consistent with the original text. All experiments were run on a server with 4 Nvidia 3080TI GPUs and a total memory of 64G. The total training time was 40 h.

4 Results

In our previous experimental setup, we compared the impact of using natural language queries and code descriptions as inputs to the code search (CS) model. Next, we evaluated the effectiveness of our Deep Semantics (DS) model by comparing the results obtained using the code descriptions recommended by the DS model and those obtained using randomly selected code descriptions as query expansion. To demonstrate the superiority of our Sentence Integrated Query Expansion (SIQE) method, we compared it with the WordNet query expansion method and the FP query expansion method. Finally, we analyzed the effect of the parameters on the performance of the SIQE model.

4.1 Preliminary Experiments

We conducted an experiment to compare the performance of the code search (CS) model using query and code description as inputs. Specifically, we trained

two CS models, DeepCS and UNIF, and then tested the models using query and code description. The results, presented in Table 3, show that for both Java and Python, the performance of the CS models is significantly better when using code description as input compared to using query as input for both DeepCS and UNIF models.

Table 3. The impact of Description and Query on the model

Model	Evaluation Set	Python				Java			
		R@1	R@5	R@10	MRR	R@1	R@5	R@10	MRR
DeepCS	**Description**	0.204	0.424	0.533	**0.31**	0.244	0.535	0.721	**0.375**
	Qurey	0.058	0.16	0.251	**0.122**	0.056	0.163	0.262	**0.124**
UNIF	**Description**	0.340	0.538	0.632	**0.439**	0.365	0.595	0.684	**0.470**
	Qurey	0.151	0.353	0.451	**0.250**	0.176	0.360	0.477	**0.269**

4.2 Main Comparison Results

We conducted several experiments to evaluate the effectiveness of our Sentence Integrated Query Expansion (SIQE) method for code search. First, we compared the performance of the DeepCS and UNIF models using the original query and the query after applying the SIQE method on the Java and Python datasets. Second, we analyzed the impact of using the DS model in the SIQE method by comparing the results obtained using k code description fragments recommended by the DS model and k code description fragments selected randomly. To demonstrate the impact of the similarity score in the SIQE method on the accuracy of code search, we compared the performance of the CS before and after adding the similarity score. Finally, we compared our SIQE method with the current popular query expansion methods, WordNet and FP, and demonstrated the superiority of the SIQE method. Table 4 presents the results of our experiments. In this table, "Query" refers to using a raw query to search for code, "WordNet" refers to searching for code using WordNet-extended queries, "FP" indicates searching for code using the query after FP expansion, "SIQE" stands for searching for code using the SIQE extended query, "SIQE_n" represents searching for code using a query after SIQE expansion that does not use similarity score weighting, and "SIQE_r" represents searching for code using a query after SIQE expansion of descriptions that are not recommended by the DS module but picked randomly. Our experiments led to four main conclusions, which can be inferred from the results presented in Table 4.

Conclusion 1: Results Show that Using the SIQE Extension Method Significantly Improves the Accuracy of Code Search Compared to Using the Original Query. It can be seen from the Table 4 that compared with the original query, the SIQE extension method greatly improves the accuracy of

the CS. Specifically, the MRR value on the Python dataset increased by 82.7% and 34.3% for the DeepCS and UNIF models, respectively. The R@1, R@5, and R@10 metrics also improved by 70.7%, 85%, 61.8%, 55.6%, 18.6%, and 29.9%. Similarly, on the Java dataset, the MRR value increased by 86.2% and 33.4% for the DeepCS and UNIF models, respectively, and the R@1, R@5, and R@10 metrics all improved by over 40%, with the maximum increase for R@1 being over 70.9%. These results demonstrate that the SIQE extension method can enhance query semantics and significantly improve the accuracy of code search by utilizing recommended code descriptions.

Table 4. Performance comparison of different extension models to code search models

Method	Python				Java			
DeepCS								
	R@1	**R@5**	**R@10**	**MRR**	**R@1**	**R@5**	**R@10**	**MRR**
Qurey	0.058	0.16	0.251	0.122	0.056	0.163	0.262	0.124
WordNet	0.076	0.214	0.392	0.144 18.0%↑	0.096	0.234	0.412	0.154 24.1%↑
FP	0.056	0.204	0.290	0.142 16.3%↑	0.066	**0.304**	0.310	0.162 30.6%↑
SIQE	**0.141**	**0.297**	**0.400**	**0.223 82.7%↑**	**0.161**	0.296	**0.421**	**0.231 86.2%↑**
SIQE_n	0.131	0.265	0.387	0.210 72.1%↑	0.105	0.287	0.354	0.198 59.6%↑
SIQE_r	0.025	0.081	0.141	0.066 45.9%↓	0.031	0.121	0.165	0.084 35.4%↓
UNIF								
	R@1	**R@5**	**R@10**	**MRR**	**R@1**	**R@5**	**R@10**	**MRR**
Qurey	0.151	0.353	0.451	0.250	0.176	0.360	0.477	0.269
WordNet	0.171	0.374	0.459	0.268 7%↑	0.206	0.404	0.509	0.275 3%↑
FP	0.191	0.386	0.480	0.285 13.9%↑	0.216	0.392	0.520	0.291 8.9%↑
SIQE	**0.236**	**0.419**	**0.587**	**0.336 34.3%↑**	**0.298**	**0.501**	**0.611**	**0.357 33.4%↑**
SIQE_n	0.171	0.357	0.494	0.298 18.8%↑	0.179	0.402	0.472	0.274 1.8%↑
SIQE_r	0.091	0.257	0.394	0.206 17.6%↓	0.140	0.332	0.419	0.205 23.7%↓

Conclusion 2: The DS Model in SIQE Plays a Key Role in Query Expansion. According to Qurey, SIQE, SIQE in the Table 4, We can observe that the SIQE extension method enhances the mrr and Recall values compared to 'Qurey'. Nonetheless, the SIQE extension method without the DS model leads to a decrease in the performance of CS. For instance, the mrr value on the python dataset in the DeepCS model drops from 0.122 to 0.066. This implies that only code descriptions that have semantic similarity with the original query will positively impact the performance, whereas randomly selected non-similar code descriptions will negatively affect the performance. Therefore, this confirms that the DS model is crucial in the SIQE method.

Conclusion 3: Adding Similarity Score to SIQE as an Extended Weight Enhances the Performance of Code Search Models. It can be observed

that both SIQE and SIQE_N significantly improve the CS compared to the original query. However, the improvement of SIQE is more significant compared to SIQE_N. For instance, on the Java dataset, the UNIF model increases from 0.274 to 0.357. This indicates that adding similarity weight reduces the impact of noise in the query expansion process and plays a role in feature filtering to some extent, thereby further enhancing the performance of the code search model.

Conclusion 4: The SIQE Extension Method is Superior to the Popular WordNet and FP Methods. In Table 4, the results for the SIQE, WordNet, and FP data indicate that the SIQE extension method performs better than the WordNet and FP methods. Compared to the current popular WordNet extension method, which is based on thesaurus extension, the SIQE method shows more promising results. For example, the WordNet method improved the DeepCS model by 18.0% on the Python dataset, while the FP method improved the model by 16.3%. In contrast, the SIQE method improved the performance of the code search model from 0.122 to 0.223, an increase of 82.7%. This suggests that the SIQE method, which is based on similar sentence meaning, can more effectively enrich the query semantics and improve the performance of code search.

4.3 Parameter Analysis

Since the k code description fragments recommended by the DS model in the SIQE extension method directly affect the representation of the final query vector, the different sources of the code description fragments searched by the DS model will cause the recommended k code description fragments to vary, thus affecting the representation of the final query vector. Therefore, we analyzed the performance variation of the SIQE extension method for different values of k and different sources of DS model searched code description fragments.

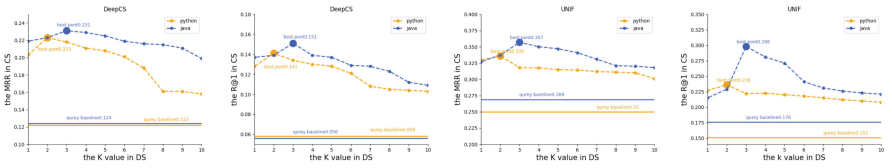

Fig. 5. Influence of different k values on code search model

To compare the impact of different k values on the code search model, we compared the MRR and Recall@1 values of the DeepCS and UNIF models in the Java and Python datasets when k varied from 1 to 10. The results are presented in Fig. 5. The graph shows an inverted U shape. For both the DeepCS and UNIF models, the best results were obtained when k was 2 or 3. This is reasonable since, when k is too small, the effect of the DeepCS model cannot be fully leveraged, while when k is too large, more noise data will be generated due

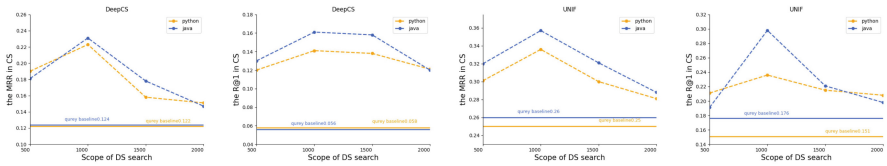

Fig. 6. Influence of DS search scope on code search model

to the limitations of the DeepCS model and the data source, which will reduce the ability of the code search model.

To compare the impact of different sources of DS model search code description fragments on the code search model, we selected four search ranges of 500, 1000, 1500, and 2000, and compared the performance of the CS model (with k set to the optimal value). The results are shown in Fig. 6. It can be seen that the best performance was achieved when the search range was about 1000 samples. This is reasonable since, when the search range is too large or too small, the performance of the DS model cannot be fully utilized. Only when the search range is around 1000 samples, which is close to the size of the DS test set, can the DS model search for more appropriate description fragments.

4.4 Example Study

Table 5 presents the results of applying different extension models to the original query "How to remove repeated elements from ArrayList?". The FP extension methods tend to find words that frequently co-occur with the keywords in the original query as the extension results, such as "repeated" and "difference", "ArrayList" and "interface", but their extension effect is very limited. The WordNet extension method extends the keywords according to the WordNet thesaurus, such as "remove" and "get rid of", "repeat" and "duplicate", etc. This greatly enriches the semantics, but also introduces a lot of noise. On the other hand, the SIQE method, by following the recommended Desc1, Desc2, Desc3, and Desc4, greatly enriches the semantics while reducing the noise, and extends the original query with sentence meaning information from the code description, thereby improving the performance of the code search model.

Table 6 shows a negative effect of applying SIQE to the query "go to next page". The correct code ranking in the code search model using the original query is 5, while the ranking after SIQE extension is more than 10. It can be seen that for some queries that are too short, the SIQE method cannot always guarantee a positive effect on the code search model. This may be due to the simplicity of our DS model, which cannot always search for the description statements that are beneficial to the code search model, or due to the insufficient size of the query-description database we collected. We plan to optimize our DS model and expand the query-description database in future work.

Table 5. Example1

Qurey	remove repeated elements from ArrayList?
FP	so remove difference repeated elements from ArrayList interface list
WordNet	remove take get rid of repeat duplicate recur elements component constituent array raiment range list tilt name
Desc1	remove the elements in descending order first index 5 then 3 then 1 remove the elements from the list without undesirable side effects
Desc2	use an iterator then it next ( ) give the next item in the array and remove ( ) will remove the last value of next ( ) for without giving an exception if keep looping
Desc3	prints the only thing is that re missing from the last matches iterating over all possible substrings and call matches (regex) on each substring
Desc4	create a linked hash set from the list contain each element only once and in the same order as the list then create a new list from this
Description	remove repeated elements to add the contents to a Set (which will not allow duplicates) and then add the Set back to the ArrayList
Code	public void remove_repeated(ArrayList yourList){Set<String> set = new HashSet<>(yourList); yourList.clear(); yourList.addAll(set); }

Table 6. Example2

Qurey	go to next page
Desc1	use a thread pool to download files in parallel
Desc2	retrieve the next item from the iterator by calling its next ( ) method if default is given
Desc3	use the urllib module to download individual ls
Desc4	Return a column of the given page number.
Description	Callback to go to the next tab . Called by the accel key .
Code	def accel_next (self , * args) : if self.get_notebook().get_current_page()+1==self.get_notebook().get_n_pages () : self.get_notebook().set_current_page (0) else : self . get_notebook ( ). next_page ( ) return True

5 Related Word

5.1 Code Search

Today, code search is becoming increasingly important in helping developers find suitable code snippets based on their queries. Code search can significantly improve the efficiency of development and reduce the workload of developers. Early code search methods relied heavily on information retrieval techniques, which primarily used keyword matching. For example, Work [22] combined code keyword matching and PageRank for code search, while Work [9] improved the performance of code search by querying the attributes of keywords. In recent years, researchers have focused on code search methods based on deep learning. These methods use neural networks to jointly embed the query and the code into a high-dimensional vector space and use the cosine similarity between vectors to determine the semantic similarity between the query and the code. For instance, Work [6] proposed the DeepCS model, which uses a bidirectional LSTM network for code search. Work [1] proposed a lightweight attention model called UNIF. Work [27] proposed the CARLCS-CNN code search model based on joint attention representation learning. Work [32] proposed TabCS, an attention-based two-stage code search model. Work [18] proposed GraphSearchNet, which is based on the graph neural network.

5.2 Code Representation

Code characterization is an upstream task whose purpose is to learn the semantics of a program by obtaining feature information from the code and using it to represent the code for downstream tasks such as clone detection, defect detection, and code summarization [8,12,14,23,31,34]. For instance, the work by [11] separates the code into tokens and inputs it into a neural network. Meanwhile, [34] builds a code representation model based on the abstract syntax tree while taking into account the upper and lower structure information of the code. With the development of pre-training models such as BERT [4] and language models [26], code search models based on pre-training models have also been proposed. For example, [5] proposed CodeBERT, which is based on the BERT pre-training model. Additionally, [7] proposed GraphCodeBERT, which is based on the graph structure.

6 Summary

In this paper, we propose a query extension model called SIQE that is based on the structure of code description statements. SIQE is designed to enrich queries by recommending code descriptions with similar semantics, and to assign weight to each recommended code description to reduce the impact of noise. In the SIQE method, we first propose a query-based code description search model called DS. DS can effectively recommend multiple code descriptions that

benefit the code search model, based on the user's input query, to enhance the query semantics. We also crawled the query-description-code corpus from Stack Overflow and validated that the SIQE method outperforms the baseline method through experiments. In future work, we plan to optimize the structure of the DS model and query-description database to further enrich the query semantics and improve the accuracy of code search.

Acknowledgement. This work is supported by the National Natural Science Foundation of China [Grant No. 61872139] and the Research Project of Hunan Provincial Education Department [Grant No. 22C0600].

References

1. Cambronero, J., Li, H., Kim, S., Sen, K., Chandra, S.: When deep learning met code search. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 964–974 (2019)
2. Chen, Q., Zhou, M.: A neural framework for retrieval and summarization of source code. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 826–831 (2018)
3. Cheng, Y., Kuang, L.: CSRS: code search with relevance matching and semantic matching. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, pp. 533–542 (2022)
4. Devlin, J., Chang, M.-W., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)
5. Feng, Z., et al.: CodeBERT: a pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155 (2020)
6. Gu, X., Zhang, H., Kim, S.: Deep code search. In: Proceedings of the 40th International Conference on Software Engineering, pp. 933–944 (2018)
7. Guo, D., et al.: GraphCodeBERT: pre-training code representations with data flow. arXiv preprint arXiv:2009.08366 (2020)
8. Haiduc, S., Aponte, J., Marcus, A.: Supporting program comprehension with source code summarization. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, vol. 2, pp. 223–226 (2010)
9. Haiduc, S., Bavota, G., Marcus, A., Oliveto, R., De Lucia, A., Menzies, T.: Automatic query reformulations for text retrieval in software engineering. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 842–851. IEEE (2013)
10. Huang, Q., Yang, Y., Cheng, M.: Deep learning the semantics of change sequences for query expansion, vol. 49, pp. 1600–1617. Wiley Online Library (2019)
11. Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., Brockschmidt, M.: CodeSearchNet challenge: evaluating the state of semantic code search. arXiv preprint arXiv:1909.09436 (2019)
12. Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Softw. Eng. **28**(7), 654–670 (2002)

13. Le, T.-D.B., Wang, S., Lo, D.: Multi-abstraction concern localization. In: 2013 IEEE International Conference on Software Maintenance, pp. 364–367. IEEE (2013)
14. LeClair, A., Haque, S., Wu, L., McMillan, C.: Improved code summarization via a graph neural network. In: Proceedings of the 28th International Conference on Program Comprehension, pp. 184–195 (2020)
15. Ling, C., Lin, Z., Zou, Y., Xie, B.: Adaptive deep code search. In: Proceedings of the 28th International Conference on Program Comprehension, pp. 48–59 (2020)
16. Linstead, E., Bajracharya, S., Ngo, T., Rigor, P., Lopes, C., Baldi, P.: Sourcerer: mining and searching internet-scale software repositories. Data Min. Knowl. Disc. **18**(2), 300–336 (2009)
17. Liu, J., Kim, S., Murali, V., Chaudhuri, S., Chandra, S.: Neural query expansion for code search. In: Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, pp. 29–37 (2019)
18. Liu, S., Xie, X., Siow, J., Ma, L., Meng, G., Liu, Y.: GraphSearchNet: enhancing GNNs via capturing global dependencies for semantic code search. IEEE Trans. Software Eng. **49**, 2839–2855 (2023)
19. Lu, M., Sun, X., Wang, S., Lo, D., Duan, Y.: Query expansion via wordnet for effective code search. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 545–549. IEEE (2015)
20. Lv, F., Zhang, H., Lou, J., Wang, S., Zhang, D., Zhao, J.: CodeHow: effective code search based on API understanding and extended boolean model (E). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 260–270. IEEE (2015)
21. McCardle, P., Cooper, J.A., Houle, G.R., Karp, N., Paul-Brown, D.: Emergent and early literacy: current status and research directions-introduction. Learn. Disabil. Res. Pract. **16**(4), 183–185 (2001)
22. McMillan, C., Grechanik, M., Poshyvanyk, D., Xie, Q., Fu, C.: Portfolio: finding relevant functions and their usage. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 111–120 (2011)
23. Nguyen, A.T., Nguyen, T.T., Al-Kofahi, J., Nguyen, H.V., Nguyen, T.N.: A topic-based approach for narrowing the search space of buggy files from a bug report. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), pp. 263–272. IEEE (2011)
24. Nie, L., Jiang, H., Ren, Z., Sun, Z., Li, X.: Query expansion based on crowd knowledge for code search. IEEE Trans. Serv. Comput. **9**(5), 771–783 (2016)
25. Poshyvanyk, D., Petrenko, M., Marcus, A., Xie, X., Liu, D.: Source code exploration with google. In: 2006 22nd IEEE International Conference on Software Maintenance, pp. 334–338. IEEE (2006)
26. Radford, A., et al.: Language models are unsupervised multitask learners. OpenAI Blog **1**(8), 9 (2019)
27. Shuai, J., Xu, L., Liu, C., Yan, M., Xia, X., Lei, Y.: Improving code search with co-attentive representation learning. In: Proceedings of the 28th International Conference on Program Comprehension, pp. 196–207 (2020)
28. Tian, Y., Lo, D., Lawall, J.: Automated construction of a software-specific word similarity database. In: 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), pp. 44–53. IEEE (2014)
29. Vaswani, A., et al.: Attention is all you need. In: Advances in Neural Information Processing Systems, vol. 30 (2017)

30. Wang, C., et al.: Enriching query semantics for code search with reinforcement learning. Neural Netw. **145**, 22–32 (2022)
31. Wang, W., Li, G., Shen, S., Xia, X., Jin, Z.: Modular tree network for source code representation learning. ACM Trans. Softw. Eng. Methodol. (TOSEM) **29**(4), 1–23 (2020)
32. Xu, L., 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), pp. 342–353. IEEE (2021)
33. Yao, Z., Peddamail, J.R., Sun, H.: CoaCor: code annotation for code retrieval with reinforcement learning. In: The World Wide Web Conference, pp. 2203–2214 (2019)
34. Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X.: A novel neural source code representation based on abstract syntax tree. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 783–794. IEEE (2019)