Empirical Software Engineering (2025) 30:87
https://doi.org/10.1007/s10664-024-10603-z

SCodeSearcher: soft contrastive learning for code search

Jia Li[1] · Zheng Fang[1] · Xianjie Shi[1] · Zhi Jin[1] · Fang Liu[2] · Jia Li[1] · Yunfei Zhao[1] · Ge Li[1]

Accepted: 8 December 2024

Abstract
Code search has been a critical software development activity in facilitating the efficiency of developers. It retrieves programs to satisfy the user intent from a codebase. Recently, many researchers have applied contrastive learning to learn the semantic relationships between queries and code snippets, resulting in impressive performance in code search. Though achieving improvements, these models ignore the following challenging scenarios in code search. First, a good code search tool should be able to retrieve all code snippets from a candidate pool that meet the given query and are implemented in diverse manners, thus the retrieved programs can satisfy different programming styles of developers. Second, in the open-source community, some programs have similar implementations but provide different functions. Code search engines need to distinguish desired programs from these confusing code snippets that have similar implementations but can not meet the query. To address these limitations, we propose a soft contrastive learning method SCodeSearcher for code search, which highlights challenging examples by arranging high weights to them based on their challenging degrees in the contrastive learning objective. We conduct extensive experiments on five representative code search datasets including code retrieval and code question answering tasks. The experimental results show that SCodeSearcher only trained on a much smaller (less than one-tenth) corpus can achieve comparable performances to existing methods optimized on the large-scale dataset, significantly saving computing resources and training time.

Keywords Code search · Contrastive learning · Deep neural network

1 Introduction

Code search has become an important software development activity. Developers typically spend 19% of their working time searching existing code to reuse them based on their intent (Shuai et al. 2022). To facilitate the development efficiency, researchers turned to investigate automatic code search. It aims to retrieve desired programs from a codebase to meet a given query. With the number of programs increasing in the open-source community, code search becomes more challenging in retrieving desired code snippets.

Communicated by: Xing Hu, Tom Zimmerman, Xin Xia

This article belongs to the Topical Collection: *Special Issue on Internetware 2023*.

Extended author information available on the last page of the article

Researchers began to perform the code search based on sparse retrieval approaches (Lu et al. 2015; Lv et al. 2015; Linstead et al. 2009). Generally, they consider lexical information and retrieve programs according to the lexical matching degree between the natural language query and the candidate code snippet. However, these approaches can not understand the semantics of queries and code snippets and are easily biased by the textual entities of the source code, especially for function names and identities. Lately, with the development of deep learning technology, many researchers proposed dense retrievers (Cambronero et al. 2019; Du et al. 2021) for code search. Specifically, they use deep neural networks to encode queries and source codes and acquire high-dimensional vectors. They then use semantical similarities to estimate their semantical relationships, and finally select the code snippets with high similarities. Among these approaches, pre-trained language models (Feng et al. 2020; Guo et al. 2020, 2022) have shown impressive performance in code search, which are trained on a large amount of code corpus with specific tasks such as masked language modeling. Although achieving initial success in code search, they mainly focus on learning the representation of a single code snippet. Recently, contrastive learning has shown great potential in many understanding tasks (Jian et al. 2024; Wu et al. 2021; Wang et al. 2021). Inspired by this, many researchers (Li et al. 2022; Shi et al. 2022; Li et al. 2023) turned to using the contrastive learning technique (Chen et al. 2020) for code search. It pulls the representations of semantically equivalent quires and source codes together, meanwhile, pushes semantically different queries and programs apart in the high-dimensional representational space. Most existing works apply heuristic methods to augment examples, including changing method names and variable names. Nevertheless, these augmented examples usually keep the same implementations as their original programs, leading to ignoring modeling the challenging scenarios in code search.

In practical software development, different programmers usually complete an intent with diverse implementations. In addition, the open-source community has a lot of programs that are implemented similarly but have different semantics. A good code search tool should possess the capability to retrieve desired programs with varied implementations that can meet the programming styles of different developers. Additionally, it should be able to distinguish desired code snippets from numerous confusing programs that share similar implementations with the ground-truth source code. To address these limitations, we propose a novel soft contrastive learning method SCodeSearcher for code search. The key idea of soft contrastive learning is sufficiently exploring the challenging relationships between queries and code snippets:

❶ **Semantic relationship between the query and false positive code snippet**: with the number of open-source programs increasing, there are a lot of code snippets that have similar implementations but are equipped with non-equivalent semantics. Therefore, for a specific natural language query, the software community usually has a mass of false positive code snippets. These false positive programs are confusing to code search tools, where these tools have difficulty in identifying the correct source code from them. Learning the semantic relationship between the query and false positive code snippets is crucial for code search tools, while it is generally neglected in existing methods. To achieve this, we construct false positive programs for each query and highlight these source codes in the contrastive learning objective.

❷ **Semantic relevance among the query and positive code snippets with diverse implementations**: developers usually complete a specific query with different implementations, that is, these programs have equivalent semantics but are implemented in diverse ways. Given a user query, it is necessary to identify diverse code snippets that are functionally equivalent to the query. Thus, the retrieved code snippets can be able to satisfy the programming styles

of different developers, leading to seamless integration of these programs into developers' projects. To achieve this, except for the original positive code, we also require the code search model to identify other functionally equivalent code snippets with diverse implementations.

To sufficiently explore the above challenging scenarios in code search, we propose a novel soft contrastive learning method named SCodeSearcher, which highlights challenging examples by arranging different weights to them based on their challenging degrees in the contrastive learning objective. Currently, large language models (LLMs) have demonstrated great intelligence in many code-related tasks, especially in code generation (GPT-3.5 2022; GPT-4 2023; THUDM: Codegeex 2022; Chen et al. 2021; Li et al. 2022). They have strong abilities in understanding user intents and generating satisfactory source codes. In this paper, we use powerful LLMs to automatically generate positive code snippets with diverse implementations and false positive programs for soft contrastive learning.

We conduct extensive experiments on five representative datasets, including CSN-Python (Husain et al. 2019), CSN-Python (Husain et al. 2019), CoSQA (Huang et al. 2021), StaQC (Yao et al. 2018), and WebQuery (Guo et al. 2022). We apply the widely used metrics (i.e., MRR and accuracy) to verify the effectiveness of SCodeSearcher. Experimental results demonstrate that SCodeSearcher trained only on a much smaller corpus can achieve comparable performances to the state-of-the-art methods (Shi et al. 2022; Li et al. 2022, 2023) that are optimized on a large amount of multi-modal dataset. Specifically, SCodeSearcher achieves 2.08% and 3.03% relative improvements on the CoSQA and StaQC datasets in terms of MRR; on the WebQuery dataset, our method achieves 4.50% relative improvements in the accuracy metric. This demonstrates that SCodeSearcher is effective and efficient in exploring the semantic relationships between queries and code snippets.

This paper extends our preliminary study, which appears as a research paper in Internetware (Li et al. 2023). In particular, we extend our preliminary work in the following direction:

- We propose a soft contrastive learning method named SCodeSearcher for code search that is an extended version of the model proposed in our preliminary work (Li et al. 2023). In our previous work, the model MCodeSearcher is proposed to learn the semantic relationships between the query and code snippet, and the semantic relevance between code snippets. In this paper, besides modeling these relationships, we also explore the challenging scenarios in code search, including identifying desired programs from confusing code snippets and retrieving programs with diverse implementations to satisfy different programming styles.
- We highlight challenging examples by incorporating different weights according to their challenging degree in the contrastive learning objective, to sufficiently grasp the semantic relationships between queries and programs.
- We further discuss how the weight ranges of both positive examples and negative instances in soft contrastive learning influence the performance of code search.

The main contributions of this paper, which form a super-set of those in our preliminary study, are summarized as follows:

- We consider the following challenging scenarios in code search: a lot of programs have similar implementations but display non-equivalent semantics; in addition, many code snippets have equivalent semantics but are implemented in diverse ways. For a good code search tool, it is not negligible to explore the semantic relationship between the query and source code in these challenging scenarios.
- We propose a soft contrastive learning method, SCodeSearcher, for code search. It highlights the challenging examples in the contrastive learning objective, aiming to sufficiently

grasp the semantic relationship between the query and false positive code snippet, and the semantic relevance among the query and positive code snippets with diverse implementations.
– We evaluate SCodeSearcher on five representative datasets. Compared to the state-of-the-art methods, our method only uses a much smaller corpus (less than one-tenth) and can achieve comparable performances to them, significantly saving training time and computing resources.
– We strengthen the experiments by adding more studies, exploring how the weights in the soft contrastive learning objective affect the performance. We also design different strategies to highlight the challenging examples.

2 Related work

2.1 Code Search

Code search is an important research topic in software engineering. The key idea of code search is to retrieve semantically equivalent source codes from a codebase given a natural language query. It can accelerate developers to reuse existing programs and improve software development efficiency.

In the early stage, researchers designed the sparse retriever to select proper code snippets. The sparse retriever pays close attention to the lexical matching between queries and source codes (Linstead et al. 2009; Lv et al. 2015). Concretely, it calculates the lexical matching degree among them and then selects the candidate programs with high matching scores. The study (Linstead et al. 2009) applied a code search tool Sourcerer to an information retrieval model. It incorporates the textual features and structural information of source codes, so as to better match programs and queries. Researchers (Lv et al. 2015) proposed a sparse phrase-based method, which retrieves the natural language query and code snippet according to their phrase-based matching degree. However, these sparse retrievers are easily affected by the user-defined code entities such as the method name and the variable names. Moreover, they lack the ability to understand the semantics of queries and source codes, thus generally achieving performance poor in retrieving desired programs.

Recently, the open-source community has appeared with a lot of high-quality source codes. Many researchers turned to dense retrieval models (Cambronero et al. 2019; Du et al. 2021) with deep learning technology that has performed impressive results in many code-related tasks. The dense retriever uses deep neural networks to encode natural language queries and code snippets into high-dimensional vectors, respectively. Based on the dense vectors, they estimate the semantic correlation between the source code and the query, then select the most matching code snippet. Researchers (Cambronero et al. 2019) compared neural code search systems running on the same platform and designed a minimal supervision extension to an existing unsupervised technique. To better model the semantic relationship, these methods (Gu et al. 2018) and Wan et al. (2019) considered the syntax information of code snippets and acquired syntax-augmented representations with multi-modal attention neural networks. Lately, pre-trained language models (Feng et al. 2020; Guo et al. 2020; Wang et al. 2021) have shown great progress in code search since they are trained on the bimodal corpus (e.g., <query, code snippet> pairs) and map them into a unified vector space. CodeBERT (Feng et al. 2020) attempted to obtain high-quality contextual embeddings of source codes by pre-training on a large amount of bimodal data. GraphCodeBERT (Guo et al. 2020) and SPT-Code (Niu et al. 2022) proposed to explore the syntactic structure of code snippets

by introducing their abstract syntax tree (AST) and dataflow information. In recent years, contrastive learning has shown impressive performance in code search (Li et al. 2023; Shi et al. 2022; Li et al. 2022). Although making progress in code search, most of them ignore the challenging scenarios between code snippets and queries. To mitigate these issues, we propose a soft contrastive learning method SCodeSearcher that focuses on sufficiently grasping the challenging semantic correlation between queries and code snippets for code search. Note that this paper does not cover the LLMs for the code search task since these models are mainly designed for generation-related tasks.

2.2 Self-supervised contrastive learning

Contrastive learning (Chen et al. 2020) is a machine learning paradigm for unsupervised representation learning, which has demonstrated state-of-the-art performance in various domains. It aims to learn program representations by minimizing the similarity between semantically similar code snippets while maximizing the similarity between semantically dissimilar source codes, bringing similar instances closer together while pushing dissimilar instances further apart in the representation space.

In software engineering, the study (Bui et al. 2021) was the first contrastive learning study for programming understanding. It employs programming heuristic transformation methods, such as renaming variables and rearranging code statements, to create code variations. Subsequently, a neural network is trained on these variations to recognize pairs of code snippets that exhibit semantic equivalence within a vast pool. Researchers (Ding et al. 2021) introduced another heuristic augmentation to create positive programs. They involve manipulating variables, altering function calls, and renaming functions. The neural network undergoes optimization to reduce the disparity between the original code snippets and the generated counterparts. Lately, CoCoSoDa (Shi et al. 2022) proposed a dynamic method to augment positive examples, which replaces tokens with types in programs and dynamically masks words in queries. Mcodesearcher (Li et al. 2023) proposed a neural-based augmentation method for generating diverse programs with the equivalent semantic. Based on the augmented code snippets, Mcodesearcher designed a multi-view contrastive learning model to sufficiently exploit the semantic correlation between queries and programs, and the relationship among code snippets. While achieving improvements in code search, these studies ignore the challenging relationships between queries and programs.

When it comes to the size of training data for contrastive learning-based code search, existing approaches generally depend on a large amount of corpus. The most commonly used dataset for contrastive learning is the CodeSearchNet (Husain et al. 2019) dataset. It contains 2,326,976 bimodal pairs and 6,452,446 unimodal examples, covering six programming languages such as Python, Java, and Ruby. However, training on such large datasets is time-consuming and resource-intensive. On the contrary, SCodeSearcher only is trained in around 40,000 bimodal examples with soft contrastive learning and achieves comparable performance to other methods.

2.3 Large language models for retrieval

Large language models (LLMs) are sophisticated AI systems trained on vast amounts of data, which have performed impressive capabilities in many domains. Some studies (Qin et al. 2023) have applied them for retrieval tasks, where a query and all candidates are connected as input, and then LLMs rank candidates. Although with good performances, the

direct application to retrieval tasks such as code search is not the best choice since the limited sequence length of LLMs impedes their applicability to large-scale retrieval tasks.

Even so, many studies introduced Generation-Augmented Retrieval (GAR) method for retrieval tasks. They first use LLMs to generate additional materials including references or other useful information. These generated materials and the query are then fed into models to retrieve the best candidate. It was the first work (Mao et al. 2020) to propose GAR and evaluate its effectiveness in the question answering task. Researchers (Gao et al. 2022; Wang et al. 2023) extended GAR to the passage retrieval task under zero-shot setting and fine-tuning scenarios. Inspired by this, researchers began to use GAR for code search. The work (Li et al. 2022) utilizes the powerful code generation model for the code retrieval task and finds that it is helpful to augment the documentation query with its generated code snippets from the code generation model. This study (Li et al. 2024) extended the foundational GAR framework and proposed a simple yet effective method that additionally rewrites the code within the codebase and normalizes the styles of generated programs. Different from the GAR framework, our approach uses LLMs to generate diverse code snippets and applies them to augment contrastive learning for resolving existing challenging scenarios in code search.

3 SCodeSearcher

In this paper, we propose a soft contrastive learning method, dubbed SCodeSearcher, for code search. The entire pipeline of our approach is shown in Fig. 1. Overall, SCodeSearcher contains three submodules: source code augmentation with LLMs, soft contrastive learning, and fine-tuning on code search.

3.1 Source code augmentation with LLMs

In software engineering, there are an increasing number of code snippets that have similar implementations but possess different semantics, leading to a mass of false positive programs

Query: Given a list of integers, return a new list containing only the even numbers.

```python
def get_even_numbers(numbers):
    even_numbers = []
    for num in numbers:
        if num % 2 == 0:
            even_numbers.append(num)
    return even_numbers
```

(a)

```python
def even_num(numbers):
    return list(filter(lambda x: x % 2 == 0, numbers))
```

(b)

Fig. 1 The pipeline of SCodeSearcher. Our approach first augments challenging examples by LLMs and then trains the model with soft contrastive learning

for a query. In addition, considering the diversity of the code implementations, it is necessary to identify the functionally equivalent code snippets with different implementations, thus meeting the needs of different developers' programming styles. The above two scenarios are challenging for a code search tool and are usually overlooked in previous studies. Therefore, we need to obtain and highlight these examples in the training process. Recently, LLMs have demonstrated amazing cod generation capabilities, which are pre-trained on a large number of bimodal examples with effective training objectives. In this paper, we use LLMs to automatically augment programs, instead of collecting them from the open-source community with human efforts.

3.1.1 Augment false positive programs

For each code snippet c_i in the training set $T = \{c_k\}_{k=1}^{n}$, we first use LLMs to generate a number of false positive programs $f_i = \left\{c_{i,f}^{m}\right\}_{m=1}^{r}$ that have similar implementations with c_i but is different from c_i in semantics. To achieve this transformation, we design specific instructions to prompt LLMs as shown in Fig. 2, where we require LLMs to make minimal changes to alter the functionality of the source code. Finally, we traverse the entire training set and acquire the false positive set F as follows:

$$F = \{f_k | c_k\}_{k=1}^{n} \tag{1}$$

where n is the number of examples in the training corpus. Based on this, the generated false positive programs are confusing to a code search tool since they have similar implementations with the ground truth.

3.1.2 Augment programs with diverse implementations

In practice, programmers generally write code snippets with different manners that have diverse implementations. Therefore, for a query, there are a lot of diverse programs that can satisfy the query. A good code search tool should be able to retrieve these source codes, resulting in meeting different programming styles. To mimic the challenging scenario, we

Query: Get the reciprocal of each number in the list.

```
def reciprocal_numbers(numbers):
    return [1 / num for num in numbers if num != 0]
```

(a)

Query: Get the reciprocal of each odd number in the list.

```
def filter_even(numbers):
    return [1 / num for num in numbers if num % 2 != 0]
```

(b)

Fig. 2 The instruction for augmenting false positive programs

use LLMs to generate programs with diverse implementations given a query, where LLMs need to prioritize changing the structure and implementation logic of codes. The instruction is shown in Fig. 3. For a code snippet c_i, LLMs generate positive programs $p_i = \left\{c_{i,p}^z\right\}_{z=1}^{w}$ that is semantically equivalent to c_i but is different in implementations. As the same with generating false positive examples, we finally obtain the positive program set P:

$$P = \{p_k | c_k\}_{k=1}^{n} \tag{2}$$

After acquiring positive programs with diverse implementations, we demand LLMs generate their functional descriptions $d_i = \left\{q_{i,p}^z\right\}_{z=1}^{w}$. That can reflect the functionality similarities between the augmented programs and their original code snippets in a certain, which are then used in Section 3.2.

3.2 Optimizing with soft contrastive learning

As described in Section 3.1, for a user requirement, there are many false positive programs and a lot of positive source codes in open-source communities, such as GitHub [1] and Stack Overflow [2]. The false positive programs are confusing for code search tools since these programs have similar implementations with user-desired code snippets. The positive programs have diverse implementations that can satisfy users with different programming styles. Thus, an impressive code search tool can not only identify the ground-truth program from many confusing programs but also retrieve diverse user-needed programs to meet different programming styles. In order to make the code search model solve the above two challenging scenarios, we propose SCodeSearcher, a soft contrastive learning approach for code search. SCodeSearcher incorporates these challenging examples into the contrastive learning objective and then highlights them in the training process. In this section, we first describe contrastive learning in detail and then elaborate our soft contrastive learning method.

3.2.1 Contrastive learning

For the code search task, contrastive learning aims to pull the semantically equivalent natural language queries and code snippets together, meanwhile, push the functionally non-equivalent user queries and programs apart. Specifically, in the mini-batch examples, N code snippets and their corresponding natural language queries are treated as N positive examples contained in the positive set $\mathbb{P}$. For a user query in the mini-batch, existing studies (Shi et al. 2022) usually treat other functionally non-equivalent code snippets as its negative items. Therefore, we acquire $(N-1)$ negative instances $\mathbb{N}$ for the query q_i. The contrastive learning loss $\mathcal{L}$ is formulated as:

$$\mathcal{L} = -\mathbb{E}_{Q_i \sim \mathbb{P}}\left[log \frac{e^{s(Q_i, C_i)/\tau}}{e^{s(Q_i, C_i)/\tau} + \sum_{C_j \in \mathbb{N}} e^{s(Q_i, C_j)/\tau}}\right] \tag{3}$$

where Q_i and C_i are the representations of the user query q_i and the code snippet c_i, respectively. In this paper, we use GraphCodeBERT (Guo et al. 2020), a multi-layer bidirectional Transformer (Vaswani et al. 2017) as the backbone, to encode code snippets and queries. Given a code snippet c_i, we use the tokenizer to split c_i into a sequence

[1] https://github.com/

[2] https://stackoverflow.com/

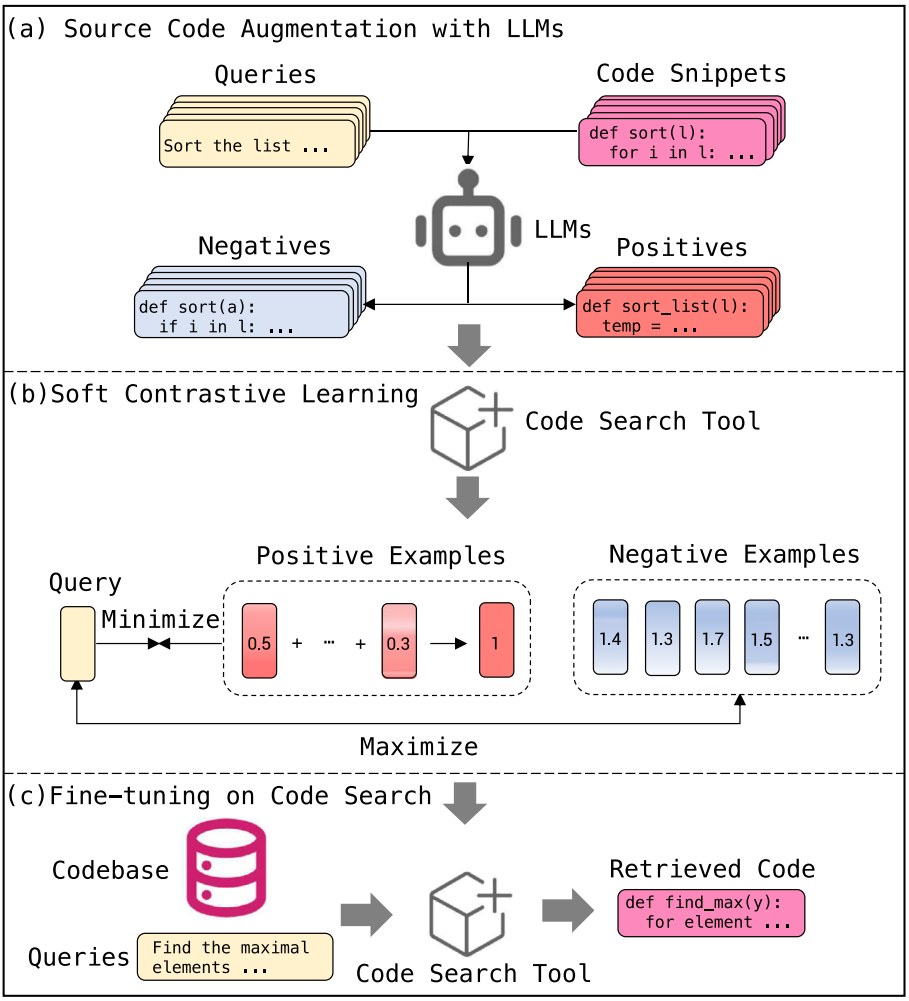

Fig. 3 The instruction for augmenting programs with diverse implementations

of tokens $\{c_{i,1}, c_{i,2}, ..., c_{i,n_c}\}$, and concatenate the sequence with a classification symbol [CLS] forming the input as $\{[\texttt{CLS}], c_{i,1}, c_{i,2}, ..., c_{i,n_c}, [\texttt{SEP}]\}$. We perform the same pre-processing procedure for the query q_i as well, and acquire the final query input as $\{[\texttt{CLS}], q_{i,1}, q_{i,2}, ..., q_{i,n_q}, [\texttt{SEP}]\}$. Then, the encoder maps them to high-dimensional representation vectors Q_i and C_i.

3.2.2 Soft contrastive learning

To sufficiently explore the semantic relationships between queries and code snippets, soft contrastive learning highlights challenging examples by arranging different weights to them based on their challenging degrees.

As described above, developers usually complete a query with different implementations. It is necessary for a code search tool to retrieve functionally equivalent code snippets with

different implementations, thus satisfying the needs of different developers' programming styles. To enhance the model to effectively grasp the semantic relevance among the query and positive code snippets with diverse implementations, we aggregate the representations of semantically equivalent code snippets into a vector, and then treat the vector as the entire semantic representation of these programs. Specifically, for a code snippet c_i, there are some positive programs $p_i = \left\{c_{i,p}^z\right\}_{z=1}^{w}$ that have equivalent semantics with c_i but are implemented in different ways, where each of them has the functional description $q_{i,p}^z$. Ideally, these programs should be equipped with the same weights in the aggregation process since they have the same semantics. However, in practice, LLMs can not guarantee the functions of the generated programs remain the same with their counterpart c_i. To minish the gap, we use TF-IDF (Aizawa 2003) to calculate the similarity score $s_{i,p}^z$ between q_i and $q_{i,p}^z$, and treat $s_{i,p}^z$ as the semantic similarity between the original program c_i and the positive code snippet $c_{i,p}^z$. We then use the similarity score $s_{i,p}^z$ as the aggregation weight of program $c_{i,p}^z$. Finally, we acquire the aggregation representation C_i for semantically equivalent programs, which can be formulated as (5) and (6). After aggregation, we use $C_{i,+}$ as the representation of programs that have the same semantics. In our soft contrastive learning, we push $C_{i,+}$ and Q_i together.

Since the false positive programs have similar implementations to user-desired programs, they are confusing for code search models. In this paper, we use the editing distance, between the original code snippet and its false positive program, as their similarity in implementations, which can reflect the challenging degrees of false positive programs in a certain. Concretely, for a code snippet c_i, we first calculate their editing distance $e_{i,j}^m$ between c_i and its false positive example $c_{i,f}^m$. Then, we treat the distance $e_{i,j}^m$ as the weight of false positive example $c_{i,f}^m$ in soft contrastive learning, where the more confusing negative programs are equipped with higher weights. As the same with the original contrastive learning, there are $(N-1)$ negative examples $\mathbb{N}_{SCL}$ for the query q_i. The soft contrastive learning objective $\mathcal{L}_{SCL}$ is formulated as:

$$\mathcal{L}_{SCL} = -\mathbb{E}_{Q_i \sim \mathbb{P}_{SCL}}\left[log \frac{e^{s(Q_i, C_{i,+})/\tau}}{e^{s(Q_i, C_{i,+})/\tau} + \sum_{C_j \in \mathbb{N}_{SCL}} e^{s(Q_i, b_j C_j)/\tau}}\right] \tag{4}$$

$$C_{i,+} = a_i C_i + \sum_{z=1}^{w} a_{i,p}^z C_{i,p}^z \tag{5}$$

$$= softmax(\alpha \cdot [1, s_{i,p}^1, \cdots, s_{i,p}^w]) \tag{6}$$

$$b_j = \begin{cases} 1 & \text{if } 1 + \dfrac{k}{1+\text{dis}} < 1 \\ 2 & \text{if } 1 + \dfrac{k}{1+\text{dis}} > 2 \\ 1 + \dfrac{k}{1+\text{dis}} & \text{otherwise} \end{cases} \tag{7}$$

$$k = (\beta - 1)(1 + \text{avg}) \tag{8}$$

In soft contrastive learning loss $\mathcal{L}_{FFS}$, b_j represents the challenging degree of the j-th negative example in $\mathbb{N}_{FFS}$. $\alpha \in [0, 1]$ is the upper bound of weights in the positive examples. $\beta \in [1, 2]$ means the upper bound of weights in the negative examples. "avg" is the average editing distance of all examples in the training corpus. "dis" means the editing distance $e_{i,j}$

between the source code c_i and its the j-th negative code snippets. For c_i, we set the weights of its other negative examples (i.e., except for false positive examples) to 1.

3.3 Fine-tuning on code search

After the model is pre-trained with soft contrastive objective, we fine-tune it on the code search task. Code search targets to retrieve the source code from a codebase that can satisfy a given query. In this section, we formalize this task with some basic notations and terminologies. Concretely, we have a training set $\mathcal{P} = \{\hat{q}_i, \hat{c}_i\}_{i=1}^{k}$ where k means the number of paired queries and code snippets in $\mathcal{P}$. $\hat{q}_i$ and $\hat{c}_i$ represent a query and a code snippet. In the training process, given a mini-batch with u paired examples, we use the pre-trained model to encode $\hat{q}_i$ and $\hat{c}_i$, respectively, and then acquire the semantic representation vectors $\hat{Q}_i$ and $\hat{C}_i$. For each query, we treat other semantically non-equivalent in the mini-batch as its negative instances. Finally, we train the model with the following loss $\mathcal{L}_{CS}$:

$$s(\hat{q}_i, \hat{c}_i) = cos(\hat{Q}_i, \hat{C}_i) = \frac{\hat{Q}_i^T \hat{C}_i}{\|\hat{Q}_i\| \|\hat{C}_i\|} \tag{9}$$

$$\mathcal{L}_{CS} = -\sum_{i=0}^{k} log \frac{e^{s(Q_i, C_i)}}{\sum_{j=1}^{u-1} e^{s(Q_i, C_j)}} \tag{10}$$

where $s(\cdot)$ denotes the cosine similarity of two items. u is the batch size in the fine-tune process.

After being optimized with the code search task, the model is applied to encode all candidate programs in the codebase. Given a new query, our model first encodes it and then calculates the semantic similarity between the query and candidate programs based on their representations. The model ranks all candidates and retrieves code snippets with higher similarities in the codebase.

4 Experimental design

We conduct extensive experiments to evaluate our approach. In this section, we will describe the details of experimental designs, including research questions, datasets, baselines, and evaluation metrics.

4.1 Research questions

We mainly answer the following three research questions (RQ).

RQ1: How does SCodeSearcher perform compared to the state-of-the-art baselines? This RQ focuses on evaluating the performance of our approach compared with existing methods. We verify their effectiveness in five code search datasets.

RQ2: What is the impact of weight ranges in soft contrastive learning? This RQ aims to evaluate the influence of the weight ranges of both positive examples and negative instances in our soft contrastive learning method.

RQ3: What is the better design for highlighting the challenging examples? As described in Section 3, we first aggregate the semantic vectors of positive examples and treat them as the entire representation for the code snippets that have the same functional-

ity. In this RQ, we explore other design choices to highlight the challenging examples and compare them to our design.

4.2 Datasets

Small datasets for soft contrastive learning The existing state-of-the-art code search methods (Li et al. 2022; Shi et al. 2022) use a large number of examples for contrastive learning. They usually train the model on CodeSearchNet (Husain et al. 2019) dataset that contains 6,452,446 code snippets and covers six programming languages, including Python, Java, and so on. Specifically, there are 1,156,085 Python programs and 1,569,889 Java programs in the dataset. Different from existing methods, we randomly select 20,000 Java and Python examples from CodeSearchNet for soft contrastive learning, respectively, where the size of the training corpus is less than one-tenth compared to the counterpart of other contrastive learning methods (Shi et al. 2022; Li et al. 2023).

Datasets for fine-tuning We conduct experiments on five representative datasets including CodeSearchNet (Husain et al. 2019), CoSQA (Huang et al. 2021), StaQC (Yao et al. 2018), and WebQuery (Lu et al. 2021). The statistics of all datasets are illustrated in Table 1. In this paper, we only evaluate SCodeSearcher on two popular programming languages (e.g., Java and Python) of CodeSearchNet. CoSQA dataset (Huang et al. 2021) consists of 19,604 labeled examples in Python language, where the queries are collected from the Microsoft Bing search engine[3] and the code snippets come from Github [4]. StaQC dataset (Yao et al. 2018) is collected from Stack Overflow[5]. In the dataset, some queries are associated with one or more relevant snippets. For code question answering, we use WebQuery dataset (Lu et al. 2021) to evaluate our SCodeSearcher. The dataset only contains the test set in Python language. Following previous studies (Guo et al. 2022; Li et al. 2023), we use the training set of the CoSQA dataset to train models and test it on WebQuery because the two datasets are collected from the same open-source platform.

4.3 Baselines

We compare SCodeSearcher with ten state-of-the-art code search models, including encoder-only methods, encoder-decoder approaches, and encoder-based contrastive learning methods. Next, we describe them in detail.

♦ The first type of baselines is encoder-only methods that are based on multi-layer Transformers, which has the same architecture as SCodeSearcher:

- **RoBERTa (code)** (Liu et al. 2019) is a multi-layer Transformer encoder (Vaswani et al. 2017) that is pre-trained on the natural language and source code (Husain et al. 2019) corpus with masked language modeling (Devlin et al. 2018) task.
- **CodeBERT** (Feng et al. 2020) learns unified representations for both code snippets and natural language queries with masked language modeling and replaced token detection tasks.
- **GraphCodeBERT** (Guo et al. 2020) introduces the data flow of code snippets into the training process and designs the data flow edge prediction task and node alignment task for learning representations of source code.

[3] https://www.bing.com/

[4] https://github.com/

[5] https://stackoverflow.com/

Table 1 The statistics of code search datasets, including the size of the training set, valid set, test set, and codebase

	Train	Valid	Test	Codebase
CSN-Java (Husain et al. 2019)	164,923	5,183	10,955	40,347
CSN-Python (Husain et al. 2019)	251,820	13,914	14,918	43,827
CoSQA (Huang et al. 2021)	19,604	500	4,500	6,267
StaQC (Yao et al. 2018)	203,700	2,600	2,749	14,579
WebQuery (Lu et al. 2021)	–	–	1,046	–

– **SynCoBERT** (Wang et al. 2021) designs two pre-training objectives originating from the symbolic and syntactic properties of source code, and introduces a multi-modal contrastive learning strategy to maximize the mutual information among different modalities.

♦ The second type of baselines is encoder-decoder pre-trained methods that have more parameters than our approach:

– **CodeT5** (Wang et al. 2021) is optimized with the denoising autoencoding task on the programming corpus, which is to reconstruct the corrupted input code sequence.
– **SPT-Code** (Niu et al. 2022) takes source code, corresponding AST, and paired function descriptions as input and designs three code-oriented sequence-to-sequence tasks, including the masked language task, code-AST prediction task, and method name generation task.
– **UniXcoder** (Guo et al. 2022) is a unified contrastive pre-trained method that leverages multi-modal contents, i.e., natural language queries and ASTs, to support code-related tasks such as code search and code generation.

♦ Besides pre-training, the third type of baselines is continually optimized with contrastive learning, which also is based on the encoder-only architecture. The size of their contrastive corpus is much larger than the dataset used in SCodeSearcher which only uses 40,000 bimodal examples:

– **CoCoSoDa** (Shi et al. 2022) optimizes the model with multimodal contrastive learning on CodeSearchNet (Husain et al. 2019) that uses 2,137,293 training examples.
– **CodeRetriever** (Li et al. 2022) trains the model on contrastive learning with 6,452,446 unimodal examples and 2,137,293 bimodal instances to learn function-level semantic representations of code snippets.
– **MCodeSearcher** (Li et al. 2023) designs a multi-view contrastive learning method to exploit the semantic relationship between queries and code snippets, which is pre-trained on 1,046,493 bimodal examples of CodeSearchNet (Husain et al. 2019).

4.4 Evaluation metrics

As shown is existing studies (Guo et al. 2022; Li et al. 2023), code search contains the code retrieval and code question answering tasks. In this paper, we measure the performance of our approach with Mean Reciprocal Rank (MRR) for the code retrieval task, and use the accuracy metric for code question answering, which are widely used (Li et al. 2022; Shi et al. 2022; Li et al. 2023). The accuracy metric measures the percentage of examples where a code snippet answers a given natural language query. MRR is the average of the reciprocal

ranks of results for a set of queries. The reciprocal rank of a query is the inverse of the rank of the first matched result. The metric MRR is formulated as follows:

$$\text{MRR} = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{\text{Rank}_i} \tag{11}$$

where $|Q|$ is the size of the query set. Rank_i denotes the place of the first correct code snippet for the query i. The higher the MRR and accuracy values are, the better the code retrieval performance is.

4.5 Experimental settings

Following existing studies (Li et al. 2023; Guo et al. 2020), we use the RoBERTa tokenizer to tokenize the code snippets and queries, which is based on Byte-Pair Encoding (BPE) algorithm (Sennrich et al. 2015). The maximal length of the query sequence and code snippet is set to 256 and 318, respectively. The hyper-parameter α is set to 0.2 and β is 1.3. The dropout probability is set to 0.1. We use the AdamW (Loshchilov and Hutter 2017) for optimizing. The initial learning rate is 1e-4 for soft contrastive learning. The warm-up step is set to 2000. We optimize SCodeSearcher for around 25,000 steps on 4×NIVDIA A6000 with batch size 64.

5 Resutls and analysis

RQ1: How does SCodeSearcher perform compared to the state-of-the-art baselines?

Setup We compare SCodeSearcher with ten advanced baselines (Section 4.3) on five code search datasets covering code retrieval and code question answering (Section 4.2). We measure the performance of models with MRR and accuracy metrics (Section 4.4). For the MRR and accuracy metrics, higher scores indicate the better performance of approaches.

Results Table 2 reports the results of different models on the five datasets, respectively. "En-Only" means the encoder-only model, "En-De" is the encoder-decoder model, and "CL" is the abbreviation of contrastive learning for convenience.

Analysis (1) Compared with existing contrastive learning methods, even with much less data (less than one-tenth), SCodeSearcher achieves comparable performances with these approaches in most experiments, significantly saving computing resources and training time. Specifically, compared to the state-of-the-art method MCodeSearcher, our method brings 2.08% and 3.03% relative improvements on CoSQA and StaQC datasets in terms of MRR; on the WebQuery dataset, our method achieves 4.50% relative improvements in the accuracy metric. By highlighting the challenging examples in soft contrastive learning based on their challenging degrees, SCodeSearcher sufficiently explores semantical relationships between queries and code snippets. (2) SCodeSearcher is superior to the models that use the encoder-decoder architecture as the backbone. The encoder-decoder models, including CodeT5, SPT-Code, and UniXcoder, have more parameters than our method. SCodeSearcher can outperform them, which further suggests the effectiveness and efficiency of our method. (3) When compared with encoder-only models, SCodeSearcher achieves substantial improvements on all datasets. GraphCodeBERT is an encoder-only model and is used to initialize SCodeSearcher. SyncoBERT is pre-trained with multi-tasks which contains multi-modal con-

Table 2 The performance of SCodeSearcher and the state-of-the-art baselines on five code search datasets

		Retrieval				Classification
		CSN-Python	CSN-Java	CoSQA	StaQC	WebQuery
En-Only	RoBERTa(code)	58.70	59.90	60.30	20.85	45.41
	CodeBERT	67.20	67.60	64.70	23.40	51.27
	GraphCodeBERT	69.20	69.10	67.50	23.80	54.39
	SyncoBERT	72.40	72.30	–	–	–
En-De	CodeT5	69.80	68.60	–	–	–
	SPT-Code	69.90	70.00	–	–	–
	UniXcoder	72.17	72.67	70.10	25.74	55.74
CL(En)	CoCoSoDa	71.70	72.60	–	–	–
	CodeRetriever	75.80	76.50	75.40	24.20	–
	MCodeSearcher	83.54	79.57	77.23	25.93	57.07
	SCodeSearcher	71.23	73.08	78.84	26.52	59.64

"CSN" is the abbreviation of CodeSearchNet for convenience

trastive pre-training on a large number of query-code pairs. Thus, through soft contrastive learning, SCodeSearcher can effectively distinguish the semantically equivalent code snippets from a candidate pool given a user query.

Answer to RQ1: Even with the much smaller (less than one-tenth) training set, SCodeSearcher can achieve comparable results on both code retrieval and code classification tasks, resulting in significantly saving training time and computing resources. Experimental results demonstrate that highlighting challenging examples in soft contrastive learning according to the challenging degree of examples is effective in constructing semantic relationships between queries and code snippets.

RQ2: What is the impact of weight ranges in soft contrastive learning?

Setup The weight ranges of both positive examples and negative instances are important parameters in our method. In this RQ, we adjust the weight range of examples in soft contrastive learning to provide some insights for the code search community.

Results The results are shown in Table 3. α means that the weight range of the original code snippet is from 0 to $1-\alpha$. The weight range of positive source code is from 0 to α. β represents that the weight range of the false positive examples is from 1 to $1+\beta$.

Analysis (1) The performances of SCodeSearcher are always better than GraphCodeBERT even if the weight range changes. This demonstrates the robustness of our soft contrastive learning method. (2) The weight ranges of both positive examples and negative code snippets have a significant influence on the performance of SCodeSearcher. For instance, as the weight range of negative examples changes, the results vary from 70.24% to 78.84% in the CoSQA dataset. Selecting proper weight ranges is non-negligible for soft contrastive learning. (3) When β is set to 1.7, the performance of SCodeSearcher drops dramatically compared to the results of the setting β=1.3. The reason might be that the model pays more attention to distinguishing the augmented false positive examples, yet ignores other diverse negative examples in the training process. When α is 0.4 for positive examples, the results only achieve 68.09% on the CoSQA dataset. We argue that the programming styles of augmented

Table 3 The impact of weight ranges of positive examples and negative instances in soft contrastive learning

	CoSQA	StaQC
GraphCodeBERT	67.50	23.80
The weight range of positive examples		
α=0.2 (Ours)	78.84	26.52
α=0.4	68.09	24.73
The weight range of negative examples		
β=1.3 (Ours)	78.84	26.52
β=1.5	79.26	26.46
β=1.7	70.24	25.92

code snippets may be different from the counterparts of the candidate examples, and thus a larger weight for augmented positive examples affects the model to learn the representation of original positive samples that conform to the programming style of the dataset, leading that the trained model is insensitive to the candidate examples in the codebase. However, in practice, the programming styles of candidate examples are diverse, thus the model will effectively user-desired programs.

Answer to RQ2: The performance of our method is always superior to the results of GraphCodeBERT in the different weight ranges. The weight ranges of both positive code snippets and negative examples have a significant effect on the performance of SCodeSearcher. Selecting suitable weight ranges is non-negligible for soft contrastive learning.

RQ3: What is the better design for highlighting the challenging examples?

Setup To highlight the challenging positive examples that are implemented in diverse ways and have the same functionality, we aggregate the semantic vectors of positive examples and treat them as the entire representation for these code snippets as described in Section 3. In this RQ, we explore other design choices to highlight the challenging examples and compare them to our design. Concretely, to highlight these positive examples, we first calculate the cosine similarities between the query and positive code snippets, respectively. Then, we aggregate their similarity scores with the weights as described in Formulation (6). Thus, the variant soft contrastive learning objective is formalized as:

$$\mathcal{L}_{VSCL} = -\mathbb{E}_{Q_i \sim \mathbb{P}}\left[log \frac{e^{s(Q_i, C_{i,+})/\tau}}{e^{s(Q_i, C_{i,+})} + \sum_{C_j \in \mathbb{N}_{SCL}} e^{s(Q_i, b_j C_j)/\tau}} \right] \tag{12}$$

$$s(Q_i, C_{i,+}) = a_i s(Q_i, C_i) + \sum_{z=1}^{w} a_{i,p}^{z} s(Q_i, C_{i,p}^{z}) \tag{13}$$

Results The results are shown in Table 4. For direct comparison, we name this manner as the fusion-last strategy and treat the aggregating method in Section 3 as the fusion-first strategy.

Analysis (1) The fusion-last strategy is not a better choice for exploring the challenging scenarios in code search. In both datasets, the fusion-last strategy causes a sharp drop. For example, this manner only achieves 45.56% on the CoSQA dataset in terms of MMR, which is worse than both the fusion-first strategy and GraphCodeBERT. (2) Highlighting the challenging examples in a reasonable strategy is important for sufficiently exploring the semantic

Table 4 The performance of different strategies for highlighting the challenging examples

	CoSQA	StaQC
GraphCodeBERT	67.50	23.80
Fusion-First Strategy	78.84	26.52
Fusion-Last Strategy	45.56	21.47

relationships between queries and code snippets. As shown in the results, the two strategies lead to significant ability differences in retrieving code snippets.

Answer to RQ3: The fusion-first strategy is better than the fusion-last manner. The fusion-first strategy can sufficiently highlight the challenging examples and effectively align the semantic relationships between code snippets and queries.

6 Discussions

6.1 Quality analysis of the generated examples with LLMs

As we all know, we can not guarantee the correctness of LLMs' generated sequences since LLMs are probabilistic-based models and sometimes they are not able to understand user instructions. Thus, it is necessary to analyze the qualities of the generated examples with LLMs. In this section, we show some cases to demonstrate the qualities of augmented examples, including the generated positive code snippets, and their function descriptions, and false positive source codes.

Figures 4 and 5 show the query and its corresponding code snippet from the CSN-Python dataset used for soft contrastive learning. The query is "read a dictionary of strings from a file". It requires three operations: "open the file", "obtain each line", and "read a dictionary". The code snippet provided by this dataset successfully implements these operations and meets the requirement.

The augmented positive code snippets Figure 6 demonstrates the generated positive code snippets with the instruction as described in Section 3.1. We can find that the generated code snippet can meet the natural language query. Meanwhile, compared with the original code snippet, the augmented source code is implemented in a different manner and has a more rigorous logic. It considers the exception scenarios including not finding the file and other errors when reading the file.

```
Instructions:

You are an excellent Python developer.
Please modify the input Python code.
You should try to make minimal changes to the code, but you need to change the
functionality of the code. You need to ensure that the function names in the code remain
unchanged.
Please provide the code only.

Input Python Code:
```

Fig. 4 The illustration of a query randomly selected from the CSN-Python dataset

```
Instruction:

You are an excellent Python developer.
Please rewrite the input Python code.
Your rewrite needs to ensure that the functionality remains unchanged. You need to
prioritize changing the structure and implementation logic of the code.
Please only provide the code.

Input Python Code:
```

Fig. 5 The code snippet of the query "read a dictionary of strings from a file" in the CSN-Python dataset

The generated functional descriptions of positive code snippets Figure 7 illustrates the generated functional descriptions of the augmented positive code snippets. The generated sequence not only provides the same requirement as the original query, but also gives a more detailed description such as describing the format of each line in the file. It further proves the functional correctness of our generated positive examples and the reliability of the generated description.

The augmented false positive source codes As shown in Fig. 8, the generated negative code snippet can not satisfy the query, meanwhile has a similar implementation to the original program. We can find that the negative code snippet converts the value of the dictionary into the integer format. However, the original query describes that the dictionary in each line is the string format. Thus, LLMs is able to generate our desired challenging negative code

```
Instructions:

You are an excellent Python developer. Please describe the functionality of the input Python code.

Here are three examples.

Input Python Code:
def create_object(cls, members):
    obj = cls.__new__(cls)
    obj.__dict__ = members
    return obj
Functionality:
Promise an object of class `cls` with content `members`.

Input Python Code:
def _xxrange(self, start, end, step_count):
        _step = (end - start) / float(step_count)
        return (start + (i * _step) for i in xrange(int(step_count)))
Functionality:
Generate n values between start and end.

Input Python Code:
def csv_to_dicts(file, header=None):
    with open(file) as csvfile:
        return [row for row in csv.DictReader(csvfile, fieldnames=header)]
Functionality:
Reads a csv and returns a List of Dicts with keys given by header row.

Input Python Code:
```

Fig. 6 The positive code snippet of the original source code generated by GPT-3.5

Query:
Read a dictionary of strings from a file.

Fig. 7 The functional description of the positive source code generated by GPT-3.5

snippets, which have similar implementations to the original program and can not meet the original query.

We find that the generated examples, both code snippets and function descriptions, can satisfy our needs, which is further evaluated by the impressive performance of SCodeSearcher. Thus, using LLMs to generate examples can be treated as a relatively reliable method. We think there are two possible reasons. The first reason is that GPT-3.5 used to generate examples has impressive generation abilities for source code and natural languages as shown in many studies (GPT-3.5 2022; ChatGPT 2022). The second reason is the instructions we have given are clear and it is easy for the LLM to understand our demands.

6.2 Case study of retrieved code snippets

To further varify the effectiveness of SCodeSearcher, we show some retrieved code snippets based on GraphCodeBERT and SCodeSearcher. Figure 9 demonstrates the retrieved results for the query "check which items exist in all sublists" in the StaQC dataset. We can find that our method SCodeSearcher can accurately understand the semantics of the query and code snippet, then retrieve the correct results from the candidate pool. On the contrary, GraphCodeBERT returns an irrelevant code snippet. This demonstrates the effectiveness of our method in understanding and distinguishing the semantics between queries and code snippets.

6.3 Threats to validity

Threats to external validity To mitigate this threat, we carefully design our experimental datasets, baselines, and evaluation metrics. For the datasets, we follow previous studies (Li et al. 2022, 2023; Shi et al. 2022) and apply five representative code search datasets. They can evaluate models' abilities from two aspects, including code retrieval and code classification. Besides, the five datasets cover two popular programming languages (i.e., Java and Python). To verify the effectiveness of SCodeSearcher, we select a series of advanced code search

```
def read_dict_from_file(file_path):
    with open(file_path) as file:
        lines = file.read().splitlines()
        obj = {}
        for line in lines:
            key, value = line.split(':', maxsplit=1)
            obj[key] = eval(value)
        return obj
```

Fig. 8 The generated false positive code snippets generated by the GPT-3.5 model

```python
def read_dict_from_file(file_path):
    obj = {}
    try:
        with open(file_path, 'r') as file:
            for line in file:
                parts = line.strip().split(':', maxsplit=1)
                if len(parts) == 2:
                    key, value = parts
                    obj[key] = value
        return obj
    except FileNotFoundError:
        print(f"{file_path} not be found")
        return {}
    except Exception as e:
        print(f"read {file_path} error: {e}")
        return {}
```

Fig. 9 An illustration of retrieved source code by SCodeSearcher and baselines

models. They are pre-trained with code-related tasks on a large amount of multi-modal corpus. For metrics, following existing works (Feng et al. 2020; Guo et al. 2022; Li et al. 2023), we select a widely used MRR metric to evaluate different methods on the code retrieval task, and use the accuracy metric to verify models on the code classification task. In addition, considering that LLMs can not guarantee to generate the correct examples that we need, to eliminate this risk, we designed clear instructions and incorporated these uncertainties into the weights in soft contrastive learning.

Threats to internal validity As we all know, deep neural models are easily influenced by the hyper-parameters. For the baselines, we apply the source code and parameters published by their original papers (Feng et al. 2020; Guo et al. 2020; Wang et al. 2021; Li et al. 2023). For our method, we use GraphCodeBERT to initialize SCodeSearcher. To select suitable hyper-parameters, we implement a small-range grid search on several hyper-parameters (i.e., weight range of positive examples α, and weight range of negative examples β, etc), and keep the training hyper-parameters (i.e., batch size, learning rate, and maximal sequence length) the same in all methods. Thus, there might be room to tune network architectures and more hyper-parameters of our approach for more improvements.

7 Conclusion

In this paper, we propose a soft contrastive learning method, SCodeSearcher, for code search. SCodeSearcher considers the challenging scenarios in code search. It highlights challenging examples with different weights according to their challenging degrees. We conduct extensive experiments on five code search datasets. Experimental results show that SCodeSearcher trained on a much smaller (less than one-tenth) corpus achieves comparable performance compared with existing contrastive learning methods. It demonstrates the effectiveness and

efficiency of our soft contrastive learning method in exploring the semantic relationships between queries and code snippets. We public our replication package[6], including the datasets and the source code, to facilitate other researchers and practitioners to repeat our work and verify their studies.

Acknowledgements This research is supported by National Natural Science Foundation of China under projects (Nos. 62192731, 62192730, 62192733), the Major Program (JD) of Hubei Province (No.2023BAA024).

Funding This research is supported by the National Key R&D Program under Grant No. 2023YFB4503801, the Natural Science Foundation of China under Grant No. 62192731, 62072007, 62192733, 61832009, and the Key Program of Hubei under Grant JD2023008.

Declarations

Conflicts of Interest All authors declared that they have no conflict of interest.

References

Aizawa A (2003) An information-theoretic perspective of tf-idf measures. Inf Process Manag 39(1):45–65

Bui ND, Yu Y, Jiang L (2021) Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In: Proceedings of the 44th international ACM SIGIR conference on research and development in information retrieval, pp 511–521

Cambronero J, Li H, Kim S, Sen K, Chandra S (2019) 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

ChatGPT (2022) https://platform.openai.com/docs/models/gpt-3-5

Chen M, Tworek J, Jun H, Yuan Q, de Oliveira Pinto HP, Kaplan J, Edwards H, Burda Y, Joseph N, Brockman G, Ray A, Puri R, Krueger G, Petrov M, Khlaaf H, Sastry G, Mishkin P, Chan B, Gray S, Ryder N, Pavlov M, Power A, Kaiser L, Bavarian M, Winter C, Tillet P, Such FP, Cummings D, Plappert M, Chantzis F, Barnes E, Herbert-Voss A, Guss WH, Nichol A, Paino A, Tezak N, Tang J, Babuschkin I, Balaji S, Jain S, Saunders W, Hesse C, Carr AN, Leike J, Achiam J, Misra V, Morikawa E, Radford A, Knight M, Brundage M, Murati M, Mayer K, Welinder P, McGrew B, Amodei D, McCandlish S, Sutskever I, Zaremba W (2021) Evaluating large language models trained on code. arXiv:2107.03374

Chen T, Kornblith S, Norouzi M, Hinton G (2020) A simple framework for contrastive learning of visual representations. In: International conference on machine learning, PMLR, pp 1597–1607

Devlin J, Chang MW, Lee K, Toutanova K (2018) Bert: pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805

Ding Y, Buratti L, Pujar S, Morari A, Ray B, Chakraborty S (2021) Contrastive learning for source code with structural and functional properties. arXiv:2110.03868

Du L, Shi X, Wang Y, Shi E, Han S, Zhang D (2021) Is a single model enough? mucos: A multi-model ensemble learning approach for semantic code search. In: Proceedings of the 30th ACM international conference on information & knowledge management, pp 2994–2998

Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, et al (2020) Codebert: a pre-trained model for programming and natural languages. arXiv:2002.08155

Gao L, Ma X, Lin J, Callan J (2022) Precise zero-shot dense retrieval without relevance labels. arXiv:2212.10496

GPT-3.5 (2022). https://platform.openai.com/docs/deprecations

GPT-4: (2023)

Gu X, Zhang H, Kim S (2018) Deep code search. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, pp 933–944

Guo D, Lu S, Duan N, Wang Y, Zhou M, Yin J (2022) Unixcoder: unified cross-modal pre-training for code representation. arXiv:2203.03850

Guo D, Ren S, Lu S, Feng Z, Tang D, Liu S, Zhou L, Duan N, Svyatkovskiy A, Fu S, et al (2020) Graphcodebert: pre-training code representations with data flow. arXiv:2009.08366

[6] https://github.com/libowen09/SCodeSearcher

Huang J, Tang D, Shou L, Gong M, Xu K, Jiang D, Zhou M, Duan N (2021) Cosqa: 20,000+ web queries for code search and question answering. arXiv:2105.13239
Husain H, Wu HH, Gazit T, Allamanis M, Brockschmidt M (2019) Codesearchnet challenge: evaluating the state of semantic code search. arXiv:1909.09436
Jian Z, Li J, Wu Q, Yao J (2024) Retrieval contrastive learning for aspect-level sentiment classification. Inf Process Manag 61(1):103539
Li D, Shen Y, Jin R, Mao Y, Wang K, Chen W (2022) Generation-augmented query expansion for code retrieval. arXiv:2212.10692
Li H, Zhou X, Shen Z (2024) Rewriting the code: A simple method for large language model augmented code search. arXiv:2401.04514
Li J, Liu F, Li J, Zhao Y, Li G, Jin Z (2023) Mcodesearcher: Multi-view contrastive learning for code search. In: Proceedings of the 14th Asia-pacific symposium on internetware, pp 270–280
Li X, Gong Y, Shen Y, Qiu X, Zhang H, Yao B, Qi W, Jiang D, Chen W, Duan N (2022) Coderetriever: unimodal and bimodal contrastive learning. arXiv:2201.10866
Li Y, Choi D, Chung J, Kushman N, Schrittwieser J, Leblond R, Eccles T, Keeling J, Gimeno F, Dal Lago A et al (2022) Competition-level code generation with alphacode. Science 378(6624):1092–1097
Linstead E, Bajracharya S, Ngo T, Rigor P, Lopes C, Baldi P (2009) Sourcerer: mining and searching internet-scale software repositories. Data Min Knowl Disc 18(2):300–336
Liu Y, Ott M, Goyal N, Du J, Joshi M, Chen D, Levy O, Lewis M, Zettlemoyer L, Stoyanov V (2019) Roberta: a robustly optimized BERT pretraining approach. arXiv:1907.11692.
Loshchilov I, Hutter F (2017) Decoupled weight decay regularization. arXiv:1711.05101
Lu M, Sun X, Wang S, Lo D, Duan Y (2015) 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
Lu S, Guo D, Ren S, Huang J, Svyatkovskiy A, Blanco A, Clement C, Drain D, Jiang D, Tang D, et al (2021) Codexglue: a machine learning benchmark dataset for code understanding and generation. arXiv:2102.04664
Lv F, Zhang H, Lou Jg, Wang S, Zhang D, Zhao J (2015) Codehow: effective code search based on api understanding and extended boolean model (e). In: 2015 30th IEEE/ACM international conference on automated software engineering (ASE), pp 260–270. IEEE
Mao Y, He P, Liu X, Shen Y, Gao J, Han J, Chen W (2020) Generation-augmented retrieval for open-domain question answering. arXiv:2009.08553
Niu C, Li C, Ng V, Ge J, Huang L, Luo B (2022) Spt-code: sequence-to-sequence pre-training for learning the representation of source code. arXiv:2201.01549
Qin Z, Jagerman R, Hui K, Zhuang H, Wu J, Shen J, Liu T, Liu J, Metzler D, Wang X, et al (2023) Large language models are effective text rankers with pairwise ranking prompting. arXiv:2306.17563
Sennrich R, Haddow B, Birch A (2015) Neural machine translation of rare words with subword units. arXiv:1508.07909
Shi E, Gub W, Wang Y, Du L, Zhang H, Han S, Zhang D, Sun H (2022) Enhancing semantic code search with multimodal contrastive learning and soft data augmentation. arXiv:2204.03293
Shuai J, Xu L, Liu C, Yan M, Xia X, Lei Y (2020) Improving code search with co-attentive representation learning. In: Proceedings of the 28th international conference on program comprehension, pp 196–207
THUDM: Codegeex (2022) https://github.com/THUDM/CodeGeeX
Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser L, Polosukhin I (2017) Attention is all you need. In: I. Guyon, U. von Luxburg, S. Bengio, H.M. Wallach, R. Fergus, S.V.N. Vishwanathan, R. Garnett (Eds.), Advances in neural information processing systems 30: annual conference on neural information processing systems 2017, December 4-9, 2017, Long Beach, CA, USA, pp 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html
Wan Y, Shu J, Sui Y, Xu G, Zhao Z, Wu J, Yu P (2019) Multi-modal attention network learning for semantic source code retrieval. In: 2019 34th IEEE/ACM international conference on automated software engineering (ASE), IEEE, pp 13–25.
Wang L, Yang N, Wei F (2023) Query2doc: query expansion with large language models. arXiv:2303.07678
Wang X, Wang Y Mi F, Zhou P, Wan Y, Liu X, Li L, Wu H, Liu J, Jiang X (2021) Syncobert: syntax-guided multi-modal contrastive pre-training for code representation. arXiv:2108.04556
Wang Y, Wang W, Joty S, Hoi SC (2021) Codet5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv:2109.00859
Wu B, Zhang Z, Wang J, Zhao H (2021) Sentence-aware contrastive learning for open-domain passage retrieval. arXiv:2110.07524
Yao Z, Weld DS, Chen WP, Sun H (2018) Staqc: a systematically mined question-code dataset from stack overflow. In: Proceedings of the 2018 World Wide Web Conference, pp 1693–1703

Authors and Affiliations

Jia Li[1] · Zheng Fang[1] · Xianjie Shi[1] · Zhi Jin[1] · Fang Liu[2] · Jia Li[1] · Yunfei Zhao[1] · Ge Li[1]

✉ Jia Li
lijiaa@pku.edu.cn

Zheng Fang
fangz@pku.edu.cn

Xianjie Shi
2100013180@stu.pku.edu.cn

Zhi Jin
zhijin@pku.edu.cn

Fang Liu
fangliu@buaa.edu.cn

Jia Li
lijia@stu.pku.edu.cn

Yunfei Zhao
zhaoyunfei@pku.edu.cn

Ge Li
lige@pku.edu.cn

1 Key Lab of High Confidence Software Technology (Peking University), MoE, Beijing, China

2 State Key Laboratory of Complex & Critical Software Environment, Beihang University, Beijing, China