CodeRetriever: Large-scale Contrastive Pre-training for Code Search

Xiaonan Li[1]*, Yeyun Gong[2], Yelong Shen[2], Xipeng Qiu[1]†,
Hang Zhang[2], Bolun Yao[2], Weizhen Qi[2], Daxin Jiang[2], Weizhu Chen[2], Nan Duan[2]
[1] Shanghai Key Laboratory of Intelligent Information Processing, Fudan University
[1] School of Computer Science, Fudan University [2]Microsoft
[1]{lixn20, xpqiu}@fudan.edu.cn,
[2]{yegong, yeshe, v-zhhang, yaobolun, weizhen djiang, wzchen, nanduan}@microsoft.com

Abstract

In this paper, we propose the CodeRetriever model, which learns the function-level code semantic representations through large-scale code-text contrastive pre-training. We adopt two contrastive learning schemes in CodeRetriever: unimodal contrastive learning and bimodal contrastive learning. For unimodal contrastive learning, we design an unsupervised learning approach to build semantic-related code pairs based on the documentation and function name. For bimodal contrastive learning, we leverage the documentation and in-line comments of code to build code-text pairs. Both contrastive objectives can fully leverage large-scale code corpus for pre-training. Extensive experimental results show that CodeRetriever achieves new state-of-the-art with significant improvement over existing code pre-trained models, on eleven domain/language-specific code search tasks with six programming languages in different code granularity (function-level, snippet-level and statement-level). These results demonstrate the effectiveness and robustness of CodeRetriever. The codes and resources are available at https://github.com/microsoft/AR2/tree/main/CodeRetriever.

```
Doc:
Return the Fibonacci number.
Code:
def Fibonacci(n):
  if n == 0:
    return 0

  elif n in [1,2]:
    return 1

  return \
    Fibonacci(n-1)+Fibonacci(n-2)
```

```
Doc:
Get the Fibonacci number.
Code:
def Fibonacci_Number(index):
   cache = [0]*(index+1)
   cache[0] = 0
   cache[1] = 1
   cache[2] = 1
   for i in range(3,index+1):
      cache[i] = \
         cache[i-1] + cache[i-2]
   return cache[index]
```

(a) Fibonacci

```
Doc:
Sort the input array into ascending order.
Code:
def bubbleSort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
         # if adjacent elements appear in
         # descending order, swap them.
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

(b) BubbleSort

Figure 1: Code examples. (a) Two different implementations of Fibonacci number algorithm; (b) Documentation, in-line comment, and code in BubbleSort implementation.

1 Introduction

Code search aims to retrieve functionally relevant code given a natural language query to boost developers' productivity (Parvez et al., 2021; Husain et al., 2019). Recently, it has been shown that code pre-training techniques, such as CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al., 2021), could significantly improve code search performance via self-supervised pre-training using large-scale code corpus (Husain et al., 2019).

However, existing code pre-training approaches usually adopt (masked) language modeling as the training objective which targets on learning to predict (masked) tokens in a given code context (Feng et al., 2020; Guo et al., 2021; Ahmad et al., 2021; Wang et al., 2021b). However, this token-based approach generally results in poor code semantic representations due to two reasons. The first one is the anisotropy representation issue. As discussed in (Li et al., 2020), the token-level self-training approach causes the embeddings of high-frequency tokens clustered and dominate the representation space, which greatly limits the expressiveness of long-tailed low-frequency tokens in pre-trained models. Thus, the anisotropic representation space induces poor function-level code semantic representation (Li et al., 2020). In programming language, the problem of token imbalance is even more severe than that of natural language. For example, common keywords and operators such as "=", "{", and "}" appear almost everywhere in

*Work is done during internship at Microsoft Research Asia.

†Corresponding author.

Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, pages 2898 - 2910
December 7-11, 2022 ©2022 Association for Computational Linguistics

Java code. The second one is the cross-language representation issue. The widely used CodeSearchNet corpus (Husain et al., 2019) contains codes from six different programming languages such as Python, Java, etc. Since the code with mixed programming languages can hardly appear within the same context, it is challenging for the pre-trained model to learn a unified semantic representation of the code with the same functionality but using different programming languages.

To address these limitations, we propose the CodeRetriever model, focusing on learning the function-level code representations, specifically for code search scenarios. The CodeRetriever model consists of a text encoder and a code encoder, which encodes text/code into separate dense vectors. The semantic relevance between code and text (or code and code) is measured by the similarity between dense vectors (Karpukhin et al., 2020b; Huang et al., 2013; Shen et al., 2014).

In the training of CodeRetriever, the code/text encoders are optimized by minimizing two types of contrastive losses: **1.Unimodal contrastive loss**, encourages the model to push codes with similar functionality closer in representation space. To estimate whether two codes are semantically close, the model needs to reason based on the given code and understand its semantics. **2.Bimodal contrastive loss**, helps model the relevance between code and text. Since the document or comment contains rich semantic information of the code, it can encourage the model to learn better code representation from natural language.

In this work, we adopt the commonly used CodeSearchNet corpus (Husain et al., 2019) for training the CodeRetreiver. CodeSearchNet mainly contains paired dataset (a function paired with a document) and unpaired dataset (only a function). The paired dataset could be directly used for bimodal contrastive learning. For unimodal contrastive learning in CodeRetriver, we build positive code-code pairs by an unsupervised semantic-guided approach. Figure 1(a) shows a code-code example: two implementations of the Fibonacci number algorithm. Moreover, the generated code-code pairs can be with different programming languages, which can mitigate the cross-language representation issues. To further take advantage of the large-scale code in unpaired data and paired data, we extract the code and in-line comment pairs to enhance the bimodal contrastive learning in CodeRetriever. Figure 1(b) shows an example to indicate that the in-line comment (comment shortly) can also reflect the code's semantics and internal logic. Specifically, the underlying logic of "if adjacent elements appear in descending order, swap them" corresponds to sorting the input array into ascending order and such fine-grained semantic information can also help learn better code representation.

Through contrasting these unimodal and bimodal pairs, CodeRetriever can 1. learn better the function-level code semantic representation, which could alleviate the anisotropy representation issue (Gao et al., 2021b; Yan et al., 2021); 2. explicitly model the relevance of codes with different programming languages and treat unified natural language as a fulcrum to mitigate cross-language representation issue. We evaluate CodeRetriever on eleven code search datasets covering six programming languages, real-world scenarios and codes with different granularity (function-level, snippet-level and statement-level), and the results show that CodeRetriever achieves a new state-of-the-art performance.

2 Preliminary: Code Search

CodeSearchNet corpus (Husain et al., 2019) is the largest publicly available code dataset. The corpus is collected from open-source non-fork GitHub repositories, which contains 2.1M paired data (a function paired with a document) and 6.4M unpaired data (only functions).

In the literature, code-search approaches (Husain et al., 2019; Jain et al., 2020; Feng et al., 2020; Guo et al., 2021) make use of the paired code-document dataset in CodeSearchNet corpus to train a siamese encoder model for language to code retrieval. However, rich unlabeled code corpus is either simply abandoned or severed as code pre-training corpus (Feng et al., 2020; Guo et al., 2021). We argue that token-level code pre-training objectives do not explicitly learn the function-level code representation. Thus existing code pre-training models (Jain et al., 2020; Feng et al., 2020; Guo et al., 2021) are sub-optimal for code search.

In this work, we propose the CodeRetriever to learn the function-level code semantic representation. CodeRetriever is initialized with the code pre-trained model (i.e., GraphCodeBERT). It takes code-doc and code-comment paired data for bimodal contrastive learning, and code-code paired data for unimodal contrastive learning. After CodeRetriever's pre-training, it can serve for downstream domain/language specified datasets.

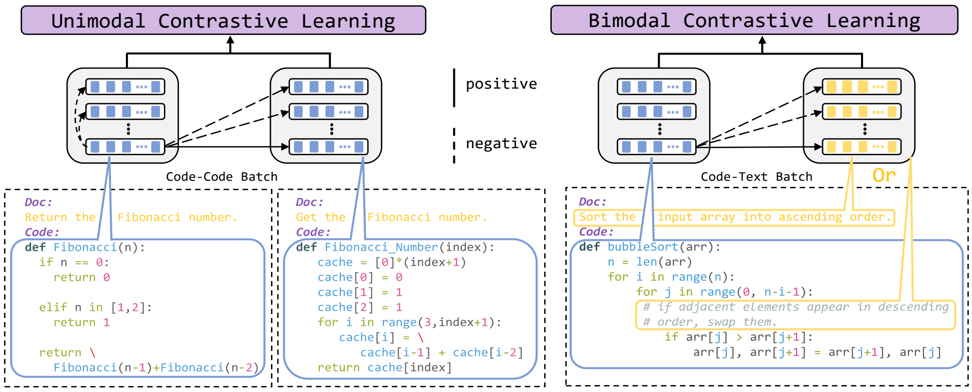


Figure 2: Unimodal and bimodal contrastive learning in CodeRetriever.

3 Approach

In this section, we present the model architecture and training objective of CodeRetriever.

CodeRetriever adopts a siamese code/text encoder architecture to represent code/text as dense vectors. Let $E_{\text{code}}(\cdot;\theta)$ and $E_{\text{text}}(\cdot;\phi)$ denote code and text encoders, respectively. The semantic similarities between code-code pair (c, c^+), and text-code pair (t, c^+) are calculated as:

$$s(c, c^+) = \left\langle E_{\text{code}}(c;\theta), E_{\text{code}}(c^+;\theta)\right\rangle \quad (1)$$

$$s(t, c^+) = \left\langle E_{\text{text}}(t;\phi), E_{\text{code}}(c^+;\theta)\right\rangle, \quad (2)$$

where $\langle,\rangle$ indicates cosine similarity operation.

3.1 Unimodal Contrastive Learning

Given a paired code-code training sample (c, c^+), the unimodal contrastive loss is given by:

$$\mathcal{L}_{\text{uni}} = -\ln \frac{\exp\left(\tau s(c, c^+)\right)}{\sum_{c' \in \mathbb{C}} \exp\left(\tau s(c, c')\right)}, \quad (3)$$

where τ is the temperature, for simplicity, we let $\tau = 1$; set $\mathbb{C}$ consists of the paired code c^+ and $N-1$ unpaired code samples obtained by in-batch negative sampling (Karpukhin et al., 2020b). In particular, one batch can consist of hybrid programming languages, which can help the pre-trained model to learn a unified semantic space of codes with different programming languages.

3.2 Bimodal Contrastive Learning

Given a paired text-code training instance (t, c^+), the bimodal contrastive loss is defined as the same manner:

$$\mathcal{L}_{\text{bi}} = -\ln \frac{\exp\left(\tau s(t, c^+)\right)}{\sum_{c' \in \mathbb{C}} \exp\left(\tau s(t, c')\right)}, \quad (4)$$

where the definitions of τ and $\mathbb{C}$ are the same as in eqn. 3. The codes of the text-code batch also consist of hybrid programming languages, which can help align the semantic space of different programming languages and natural language. Since the document or comment reflects the functionality and crucial semantic information of source code, such positive pairs can help model better understand the semantics of code.

3.3 Overall Pre-training Objective

As illustrated in Figure 2, CodeRetreiver takes two types of text-to-code for bimodal contrastive training, which are code-document and code-comment. Therefore, we use $\mathcal{L}^1_{\text{bi}}$ and $\mathcal{L}^2_{\text{bi}}$ to denote code-document and code-comment contrastive loss, respectively. The overall pre-training objective for CodeRetreiver is:

$$\mathcal{L}(\theta, \phi) = \mathcal{L}_{\text{uni}} + \mathcal{L}^1_{\text{bi}} + \mathcal{L}^2_{\text{bi}} \quad (5)$$

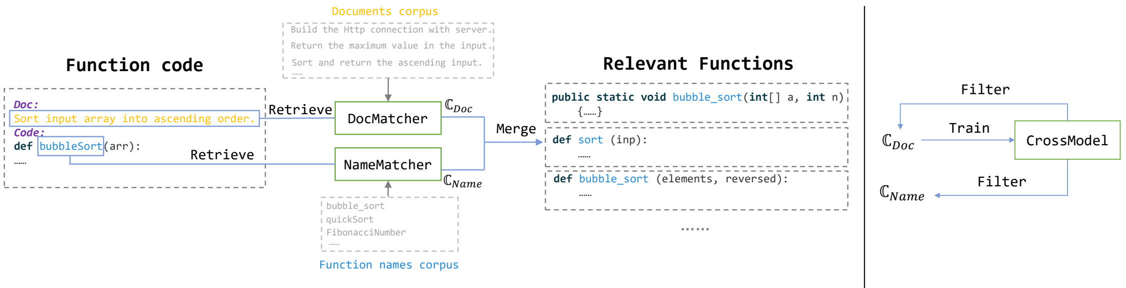
4 Building Positive Pairs

4.1 Code-Document

Documents of source codes usually can provide rich semantic information and highly describe the functionality of codes. For example, in Figure 1(b), the document "Sort the input array into ascending order." clearly summarizes the goal of the code, which can help the model to better understand the code. So we take code c and its corresponding document t as positive pairs. Thus we can not only help model better understand code but also align different programming languages' representation through the unified natural language description as a pivot.

4.2 Code-Comment

Unlike documents, the in-line comments widely exist in unpaired code. As shown in Figure 1(b), it can reflect the code's internal logic and contains fine-grained semantic information, despite certain noisy



(a) (b)

Figure 3: The illustration of building code-code pairs. (a) Step 1. Collect noisy code-code pairs through function name match and documentation match; (b) Step 2. Denoise code-code pairs with CrossModel.

signals. So we consider code-comment as positive pairs to further help model to learn better code representation. In this section, we introduce how we build code-comment pairs. We first leverage the code parser (tree-sitter) to split the code-block into two parts: pure code and the corresponding in-line comments. Then we perform post-processing as follows to filter noisy paired samples to obtain the code-comment corpus:

- We merge comments with continuous lines into one comment. This is inspired by the phenomenon where developers usually write a complete comment into multiple-lines to make it easier to read, like in Figure 1(b).
- Comments with little information are removed, including: 1) shorter than four tokens; 2) comments beginning with "TODO"; 3) comments for automated code checking, like "Linter $\cdots$"[1]. 4) non-text comments, i.e., commented code.
- Functions with little semantic information are removed such as functions with names "__getter__", "__setter__" etc, are removed.

After cleaning, we collect about 1.9 million code-comment pairs. The detailed statistics of the overall code-text corpus can be seen in Appendix A.

4.3 Code-Code

Code-code paired datasets can provide explicit training signals for models to learn the semantic representation of code. However, it is challenging to build large-scale and high-quality semantically relevant code-to-code pairs from an unlabeled corpus. To a specific functionality, there are a lot of ways to implement it and the resulting code can be full of diversity. They can have totally different logic, libraries invoked, and identifier names. Even for experienced developers, it's challenging and time-consuming for them to assess the semantic similarity of two codes, which makes human annotation costly and not scalable. Although two codes of the same functionality can have different implementations, their documentations or function names can be very similar, as shown in Figure 1(a). Inspired by this phenomenon, we propose the unsupervised techniques as following to collect a large-scale code-to-code corpus.

Step 1. Collect noisy code-code pairs by maching function name and documentation. **1)** We adopt the recently proposed unsupervised method, SimCSE (Gao et al., 2021b), to train with the function name corpus, obtain "NameMatcher" model; and train with documentation corpus to obtain "DocMatcher" model; Both "NameMatcher" and "DocMatcher" are dense retrieval models. For example, given a function name, "NameMatcher" could be able to retrieve top-K relevant function names in the corpus. We refer readers to its original paper (Gao et al., 2021b) for more details. **2)** For any given function in the corpus, we retrieve its relevant functions through function name matching using the "NameMatcher". The similar manner is applied to "DocMatcher", which collects code-code pairs by matching their corresponding documentations. We denote the code-code pairs collected through "DocMatcher" as $\mathbb{C}_{\text{Doc}}$, and use $\mathbb{C}_{\text{Name}}$ to indicate the code-code pairs collected through "NameMatcher". We only keep code-code pairs if their retrieval scores (by "NameMatcher" and "DocMatcher") are greater than threshold (0.75).

Step 2. Denoise Code-code pairs with CrossModel. The code-code sets $\mathbb{C}_{\text{Name}}$ and $\mathbb{C}_{\text{Doc}}$ collected from Step 1 can be noisy, especially for $\mathbb{C}_{\text{Name}}$ as functions with the same function name can have different functionalities. In this step, we train a binary classifier model, CrossModel (M_c), for filtering noisy code-code pairs. **1)** We take the code-code pairs $\mathbb{C}_{\text{Doc}}$, which is less noisy, as the training set to train the CrossModel M_c. It takes

[1] Linter is a static analysis tool for checking code.

Lang	Ruby	Javascript	Go	Python	Java	PHP	Overall
ContraCode (Jain et al., 2020)	-	30.6	-	-	-	-	-
SyncoBERT (Wang et al., 2021a)	72.2	67.7	91.3	72.4	72.3	67.8	74.0
CodeBERT (Feng et al., 2020)	67.9	62.0	88.2	67.2	67.6	62.8	69.3
GraphCodeBERT (Guo et al., 2021)	70.3	64.4	89.7	69.2	69.1	64.9	71.3
UniXcoder (Guo et al., 2022)	74.0	68.4	91.5	72.0	72.6	67.6	74.4
CodeRetriever (In-Batch Negative)	75.3	69.5	91.6	73.3	74.0	68.2	75.3
CodeRetriever (Hard Negative)	75.1	69.8	92.3	74.0	74.9	69.1	75.9
CodeRetriever (AR2)	**77.1**	**71.9**	**92.4**	**75.8**	**76.5**	**70.8**	**77.4**

Table 1: The comparison on the CodeSearch dataset. We get the ContraCode's result by fine-tuning the released checkpoint (Jain et al., 2020). Other results of compared models are reported by previous papers.

Dataset	Adv	CoSQA	CoNaLa	SO-DS	StaQC	Overall
SyncoBERT (Wang et al., 2021a)	38.1	-	-	-	-	-
CodeBERT (Feng et al., 2020)	27.2	64.7	20.9	23.1	23.4	31.9
GraphCodeBERT (Guo et al., 2021)	35.2	67.5	23.5	25.3	23.8	35.1
UniXcoder (Guo et al., 2022)	41.3	70.1	-	-	-	-
CodeRetriever (In-Batch Negative)	43.0	70.6	29.6	27.1	**25.5**	39.0
CodeRetriever (Hard Negative)	45.1	74.1	**29.9**	31.8	24.6	41.1
CodeRetriever (AR2)	**46.9**	**75.4**	29.1	**33.9**	24.2	**41.9**

Table 2: The comparison on datasets that are closer to the real scenario. The results of Compared models on the Adv dataset and UniXcoder on CosQA are reported by previous papers, other results are from our implementation since they are not reported previously.

the concatenation of code-code pair as input and is more powerful for predicting their relevant score (range from 0 to 1) via deep token interaction. In the training of M_c, we use set $\mathbb{C}_{\mathrm{Doc}}$ as positive training instances while sampling random code-code pairs as negative instances. **2)** We remove the code-code pairs in $\mathbb{C}_{\mathrm{Doc}}$ and $\mathbb{C}_{\mathrm{Name}}$ if their prediction scores by M_c are smaller than certain threshold. Let $\mathbb{C}^*_{\mathrm{Name}}$ and $\mathbb{C}^*_{\mathrm{Doc}}$ be the denoised subsets of $\mathbb{C}_{\mathrm{Name}}$ and $\mathbb{C}_{\mathrm{Doc}}$. The final code-code corpus is the joint of set $\mathbb{C}^*_{\mathrm{Name}}$ and $\mathbb{C}^*_{\mathrm{Doc}}$. Since we take the natural language as the anchor to get $\mathbb{C}^*_{\mathrm{Name}}$ and $\mathbb{C}^*_{\mathrm{Doc}}$, the code pair can have different programming languages and mitigate the cross-language representation issue.

We show the process of Step 1 and Step 2 in Figure 3(a) and Figure 3(b), respectively. Overall, the collected code-code corpus contains 23.4 million pairs. We provide a more detailed description on building code-code corpus, involved hyper-parameters and detailed cross-language statistics of code-code pairs in Appendix B, C and D.

5 Experiment

For fair comparison, CodeRetriever adopts the same model architecture as previous works (Feng et al., 2020; Guo et al., 2021). CodeRetriever shares parameters of code encoder and text encoder. It contains 12 layers Transformer with hidden size of 768 and attention heads of 12. To accelerate the training process, we initilaize CodeRetriever with the released parameters of GraphCodeBERT (Guo et al., 2021). We show more details in Appendix E.

5.1 Benchmark Datasets

We evaluate CodeRetriever on several code search benchmarks, including **CodeSearch** (Husain et al., 2019; Guo et al., 2021), **Adv** (Lu et al., 2021), **CoSQA** (Huang et al., 2021), **CoNaLa** (Yin et al., 2018), **SO-DS** (Heyman and Cutsem, 2020), **StaQC** (Yao et al., 2018). The CodeSearch benchmark contains six datasets with different programming languages. The Adv dataset normalizes the method names and variable names in the dev/test set, which makes it more challenging. CoNaLa, SO-DS, and StaQC are collected from stackoverflow questions, and CoSQA are collected from web search engines. Therefore, the queries in CoSQA, CoNaLa, SO-DS, and StaQC are closer to the real code-search scenario compared with Adv and CodeSearch. Meanwhile, CoNALA, SO-DS and StaQC contain the code with different granularity, i.e., statement-level and snippet-level. The statistics of these benchmark datasets are listed in Appendix F. Following previous works (Feng et al., 2020; Guo et al., 2021), we use Mean Reciprocal Rank (MRR) (Hull, 1999) as the evaluation metric on all benchmark datasets.

5.2 Experiment: Fine-Tuning

In the fine-tuning experiments, CodeRetriever and other code pre-trained models are fine-tuned on the eleven language/domain-specific code search tasks, each task provides a set of labeled query-code pairs for model adaptation. [2]

5.2.1 Fine-tuning

Previous works on dense text retrieval (Karpukhin et al., 2020a; Xiong et al., 2021; Qu et al., 2021) show that the strategy of selecting negative samples could greatly affect the model performance in contrastive learning tasks. Therefore, we explore the following three approaches for CodeRetriever fine-tuning: **1.In-Batch Negative.** For a <query, code> pair in a batch, it uses other codes in the batch as negatives (Karpukhin et al., 2020a). Existing code pre-trained models take in-batch negative as the default fine-tuning method (Feng et al., 2020; Guo et al., 2021; Wang et al., 2021a). **2.Hard Negative.** It can pick "hard" representative negative samples other than random negatives. Compared with in-batch negative, the hard negative training is more efficiency (Karpukhin et al., 2020a), which is widely used in text dense retrieval. We follow Gao et al. (2021a) for hard negative fine-tuning. **3.AR2.** It is a recently proposed training framework for dense retrieval (Zhang et al., 2021). It adopts an adversarial-training approach to select "hard" negative samples iteratively. In this paper, we focus on using AR2 to enhance the siamese encoder for code search.

In fine-tuning experiments, we conduct grid search over learning-rate in {2e-5, 1e-5}, batch-size in {32, 64, 128}. Training epoch, warm-up step, and weight decay are set to 12, 1000, and 0.01, respectively on all tasks. We report the average results under 3 different random seeds. The hyper-parameters for AR2 training are listed in Appendix G.

We compare CodeRetriever with state-of-the-art pre-trained models, including: **CodeBERT** (Feng et al., 2020), pre-trained with MLM and replaced token detection tasks; **GraphCodeBERT** (Guo et al., 2021), which integrates data flow baesed on CodeBERT. **SynCoBERT** (Wang et al., 2021a), pre-trained on code-AST pairs with contrastive learning; **ContraCode** (Jain et al., 2020), pre-trained with contrastive learning through semantic-preservin code transformation on Javascript corpus. **UniXcoder** (Guo et al., 2022) is adpated from UniLM and pre-trained on unified cross-modal data like code, AST and text.

[2]For the CodeSearch benchmark, although it has overlapping with the paired data of pre-training corpus, fine-tuning on it is different from CodeRetriever's bimodal contrastive learning. In detail, fine-tuning on CodeSearch only covers one specific programming language's query-code pair while CodeRetriever's bimodal contrastive learning covers 1. six hybrid programming languages for unifying their semantic space 2. extra comment-code pair for further taking advantage of the unpaired data.

5.2.2 Results

Table 1 and Table 2 show the performance comparison on all benchmark datasets. First, we report the performance of CodeRetriever (In-Batch Negative), which uses the same finetuning approach as other baselines to ensure a fair comparison. It shows that CodeRetriever obtains the best overall performance compared with all other compared approaches. Specifically, CodeRetriever improves over GraphCodeBERT by 4.0 average absolute points on the CodeSearch dataset, which demonstrates the effectiveness of CodeRetriever. Meanwhile, CodeRetriever outperforms the previous state-of-the-art model, UniXcoder (Guo et al., 2022), on all tasks with reported results. On the Adv, CoSQA, CoNaLa, SO-DS and StaQC datasets, CodeRetriever also outperforms baselines models, which shows that CodeRetriever consistently outperforms baseline models in various scenarios.

Comparing different fine-tuning approaches, we can see that the AR2 is generally better than In-Batch Negatives and Hard Negatives. i.e., CodeRetriever(AR2) improves over In-Batch Negative by 3.0 absolute points in average, and improves over Hard Negative by 1.1 absolute points in average. The experiment results suggest that selecting a good fine-tuning approach is also very important for downstream code search tasks. From Table 2, an interesting observation is that In-Batch Negative outperforms Hard Negatives and AR2 on StaQC benchmark. A possible explanation is StaQC contains more false query-code pairs in the training set compared with other benchmarks, as it is collected from stackoverflow through a rule-based method without any human annotations, and In-Batch Negative is more noise-tolerant than AR2 and Hard-Negative.

5.3 Analysis

5.3.1 Low-Resource Code Search

We evaluate the performance of CodeRetriever on low resource scenario, i.e., only a few hundreds of paired query-code data for fine-tuning. Table 4 shows the results of CodeRetriever and GraphCodeBERT in the low-resource setting on CoSQA dataset, where the number of training examples is varied from 500 to FULL (19K). We can see that CodeRetriever could reach more reasonable performance in low-resource setting than GraphCodeBERT.

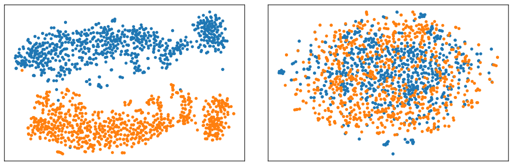
5.3.2 Cross-Language Code Search

Performance Since building pairs of real user query and code is labor-intensive and costly, Existing code search datasets of real-world scenario only cover few programming language, including Python (Yao et al., 2018; Heyman and Cutsem, 2020; Yin et al., 2018; Huang et al., 2021), Java (Nie et al., 2017; Li et al., 2019) and SQL (Yao et al., 2018). Here, we introduce a new setting, cross-language code search, where we fine-tune model with 'A' programming language and test it on 'B' programming language. This can alleviate the data scarcity problem of other programming languages. For evaluating our method on this setting, we finetune the model with query-Python corpus (CoNaLa (Yin et al., 2018)) and evaluate it with query-Java test set (Li et al., 2019). The queries in the Python corpus and Java corpus are both collected from stackoverflow. In Table 3, it shows that unimodal contrastive loss in CodeRetriever significantly helps the cross-language code search task. By combining bimodal contrastive loss, CodeRetriever could obtain better performance. This result indicates CodeRetriever's potential utility for real scenarios.

Visualization To further analyze the effect of unimodal contrastive learning, we visualize the 2-D latent space of representations with or without unimodal contrastive learning by t-SNE (van der Maaten and Hinton, 2008). In the Figure 4(a), we can see the representations of Java and Python code appear in two separate clusters for the model without unimodal contrastive learning (GraphCodeBERT) while in Figure 4(b), their representation space are overlapped. It shows that the unimodal contrastive learning helps to learn a unified representation space of code with different programming languages.

5.3.3 Code-to-Code Search Results

We fine-tune and evaluate CodeRetriever on code-to-code search task. In this task, given a code, the model is asked to return a semantically related code. We conduct experiment on POJ-104 dataset (Mou et al., 2016; Lu et al., 2021) and use the same hyperparameters as previous works (Lu et al., 2021). We evaluate by Mean Average Precision (MAP), as shown in table 5. We see that CodeRetriever outperforms other pre-trained models, which demonstrates its scalability and potentiality for other code understanding tasks.



(a) without $\mathcal{L}_{\text{uni}}$ (b) with $\mathcal{L}_{\text{uni}}$

Figure 4: The 2-D visualizations of Python and Java's representation, where • and • represent samples of Java and Python, respectively.

Method	MRR
GraphCodeBERT	41.6
CodeRetriever ($\mathcal{L}_{\text{uni}}$)	48.4
CodeRetriever ($\mathcal{L}_{\text{uni}} + \mathcal{L}_{\text{bi}}$)	53.3

Table 3: The comparison on cross language code search.

5.3.4 Uniformity and Alignment

To study the effect of CodeRetriever on the function-level representation space, we use the alignment and uniformity metrics (Wang and Isola, 2020) to see function-level representation distribution changes during training, shown in Figure 5. We see that the uniformity loss of CodeRtriever descends gradually, indicating the anisotropy is alleviated. We find that the alignment loss also has a declining trend, which shows the training of CodeRetriever can help align the representation of code and natural language and better understand them. The two metrics indicate that the CodeRetriever reduces the gap between pre-training and fine-tuning, compared with previous code pre-trained models.

5.3.5 Ablation Study

To understand the effect of each component in CodeRetriever, we conduct ablation study on the CodeSearch Java dataset and SO-DS. We start from the initial model and add components of CodeRe-

Train Size	500	1000	2000	4000	FULL
GraphCodeBERT	43.2	49.9	54.0	57.2	67.5
CodeRetriever	54.7	55.6	58.4	60.5	70.6

Table 4: The performance comparison on CosQA with different training size.

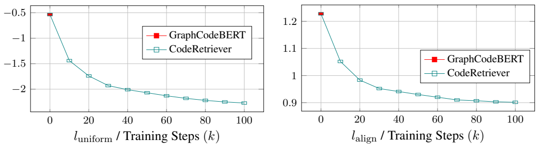


Figure 5: The alignment and uniformity curve.

Model	MAP@R
RoBERTa (Liu et al., 2019)	76.67
CodeBERT (Feng et al., 2020)	82.67
GraphCodeBERT (Guo et al., 2021)	85.16
SynCoBERT (Wang et al., 2021a)	88.24
DISCO (Ding et al., 2021)	82.77
Corder (Bui et al., 2021)	84.10
CodeRetriever	**88.85**

Table 5: The performance comparison on the code-to-code retrieval task (Mou et al., 2016). Compared models' results are from previous papers (Wang et al., 2021a; Ding et al., 2021; Bui et al., 2021).

triever to it one-by-one. We find that using code-code pairs without denoising for unimodal contrastive learning brings slight performance degradation while with denoising, it achieves significant performance improvement. This demonstrates the effectiveness of the denoising step and shows that the unimodal contrastive learning depends on the quality of positive pairs construction. Here, we verify a simple and effective positive pairs construction method, we leave the development of more powerful method as future work. From the results of using doc-code and comment-code for bimodal contrastive learning, we see that the model achieves further performance improvement, which shows the bimodal contrastive learning can leverage crucial semantic information in documents or comments to help better understand the code.

6 Related Work

Token-Level Code Pre-training Token-level pre-trained models have been widely-used for the programming languages. Karampatsis and Sutton (2020) pre-train ELMo on JavaScript corpus for program-repair task. Kanade et al. (2020) use a large-scale Python corpus to pre-train the BERT model. C-BERT (Buratti et al., 2020) is pre-trained on a lot of repositories in C language and achieves significant improvement in abstract syntax tree (AST) tagging task. CodeBERT (Feng et al., 2020) is pre-trained by the masked language model and replaced token detection tasks on the text-code pairs of six programming languages. GraphCodeBERT (Guo et al., 2021) introduces the information of dataflow based on CodeBERT. Besides these BERT-like models, CodeGPT (Svyatkovskiy et al., 2020), PLBART (Ahmad et al., 2021), CoTexT (Phan et al., 2021) and CodeT5 (Wang et al., 2021b) are pre-trained for code generation tasks based on the GPT, BART (Lewis et al., 2019) and T5 (Raffel et al., 2020) respectively. However, token-level objectives cause the anisotropy problem (Guo et al., 2022) and have a gap with code search which is based on function-level representations. Different from these works, CodeRetriever utilizes the contrastive-learning framework to enhance the function-level representation.

Contrastive Learning for Code Recently, several works try to use contrastive learning on programming language, whose key is building effective positive or negative samples. ContraCode (Jain et al., 2020) and Corder (Bui et al., 2021) use semantics-preserving transformations, such as identifier renaming and dead code insertion to build positive pairs. Ding et al. (2021) develop bug-injection to build hard negative pairs. SynCoBERT (Wang et al., 2021a) and Code-MVP (Wang et al., 2022) build positive pairs through programs'compilation process like AST and CFG. However, their methods usually generate positive samples with similar structure or the same variable names as the original code, whose naturalness and diversity is limited by hand-written rules(Li et al., 2022). In CodeRetriever, we construct positive pairs from code-code, code-documentation, and code-comment. For code-code, we design a more natural and diverse positive pairs construction method based on real-world codes.

7 Conclusion

In this paper, we introduce CodeRetriever that combines unimodal and bimodal contrastive learning as pre-training tasks for code search. For unimodal contrastive learning, we propose a semantic-guided method to build positive code pairs. For bimodal contrastive learning, we utilize the document and in-line comment to build positive text-code pairs. Extensive experimental results on several publicly available benchmarks show that the

Methods	CodeSearch	SO-DS
GraphCodeBERT (Our Initial)	69.1	25.3
+ Code-to-Code (no denoising)	68.9	25.2
+ Code-to-Code (denoising)	71.1	25.9
+ Doc-to-Code	72.2	26.6
+ Comment-to-Code	74.0	27.1

Table 6: Ablation study.

proposed CodeRetriever brings significant improvement and achieves new state-of-the-art on all benchmarks. Further analysis results show that CodeRetriever is also powerful on low resource and cross-language code search tasks, and demonstrate the effectiveness of unimodal and bimodal contrastive learning.

Limitations

CodeRetriever mainly has two limitations:

1) Due to the limited computing infrastructure, only GraphCodeBERT is used as the initialization model in the experiments. We leave experiments based on other code pre-trained models such as UniXcoder (Guo et al., 2022) as future work.

2) The code-code pairs and code-comment pairs still contain certain noise. We will explore stronger denoising methods in future work.

3) We pre-train CodeRetriever on the CodeSearchNet corpus. In future work, we will consider using more pre-training corpora such as full Github repositories.

Acknowledgements

This work was supported by the National Key Research and Development Program of China (No.2020AAA0106700) and National Natural Science Foundation of China (No.62022027).

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online. Association for Computational Linguistics.

Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In *SIGIR '21: The 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, Virtual Event, Canada, July 11-15, 2021*, pages 511–521. ACM.

Luca Buratti, Saurabh Pujar, Mihaela A. Bornea, J. Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, and Giacomo Domeniconi. 2020. Exploring software naturalness through neural language models. *CoRR*, abs/2006.12641.

Yangruibo Ding, Luca Buratti, Saurabh Pujar, Alessandro Morari, Baishakhi Ray, and Saikat Chakraborty. 2021. Contrastive learning for source code with structural and functional properties. *CoRR*, abs/2110.03868.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Luyu Gao, Yunyi Zhang, Jiawei Han, and Jamie Callan. 2021a. Scaling deep contrastive learning batch size under memory limited setup. In *Proceedings of the 6th Workshop on Representation Learning for NLP*.

Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021b. Simcse: Simple contrastive learning of sentence embeddings. *CoRR*, abs/2104.08821.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Geert Heyman and Tom Van Cutsem. 2020. Neural code search revisited: Enhancing code snippet retrieval through natural language intent. *CoRR*, abs/2008.12193.

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. CoSQA: 20,000+ web queries for code search and question answering. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 5690–5700, Online. Association for Computational Linguistics.

Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. 2013. Learning deep structured semantic models for web search using clickthrough data. ACM International Conference on Information and Knowledge Management (CIKM).

David A. Hull. 1999. Xerox TREC-8 question answering track report. In *Proceedings of The Eighth Text REtrieval Conference, TREC 1999, Gaithersburg, Maryland, USA, November 17-19, 1999*, volume 500-246 of *NIST Special Publication*. National Institute of Standards and Technology (NIST).

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436.

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E. Gonzalez, and Ion Stoica. 2020. Contrastive code representation learning. *CoRR*, abs/2007.04973.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2017. Billion-scale similarity search with gpus. *CoRR*, abs/1702.08734.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Pre-trained contextual embedding of source code. *CoRR*, abs/2001.00059.

Rafael-Michael Karampatsis and Charles Sutton. 2020. Scelmo: Source code embeddings from language models. *CoRR*, abs/2004.13214.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick S. H. Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020a. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 6769–6781. Association for Computational Linguistics.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020b. Dense passage retrieval for open-domain question answering. *CoRR*, abs/2004.04906.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*.

Bohan Li, Hao Zhou, Junxian He, Mingxuan Wang, Yiming Yang, and Lei Li. 2020. On the sentence embeddings from pre-trained language models. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9119–9130, Online. Association for Computational Linguistics.

Hongyu Li, Seohyun Kim, and Satish Chandra. 2019. Neural code search evaluation dataset. *CoRR*, abs/1908.09804.

Xiaonan Li, Daya Guo, Yeyun Gong, Yun Lin, Yelong Shen, Xipeng Qiu, Daxin Jiang, Weizhu Chen, and Nan Duan. 2022. Soft-labeled contrastive pre-training for function-level code representation.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 1287–1293. AAAI Press.

Liming Nie, He Jiang, Zhilei Ren, Zeyi Sun, and Xiaochen Li. 2017. Query expansion based on crowd knowledge for code search. *CoRR*, abs/1703.01443.

Md. Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. *CoRR*, abs/2108.11601.

Long N. Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James T. Anibal, Alec Peltekian, and Yanfang Ye. 2021. Cotext: Multi-task learning with code-text transformer. *CoRR*, abs/2105.08645.

Yingqi Qu, Yuchen Ding, Jing Liu, Kai Liu, Ruiyang Ren, Wayne Xin Zhao, Daxiang Dong, Hua Wu, and Haifeng Wang. 2021. Rocketqa: An optimized training approach to dense passage retrieval for open-domain question answering. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 5835–5847. Association for Computational Linguistics.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67.

Yelong Shen, Xiaodong He, Jianfeng Gao, Li Deng, and Gregoire Mesnil. 2014. A latent semantic model with convolutional-pooling structure for information retrieval. In *CIKM*.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: code generation using transformer. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1433–1443. ACM.

Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(86):2579–2605.

Tongzhou Wang and Phillip Isola. 2020. Understanding contrastive representation learning through alignment and uniformity on the hypersphere. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 9929–9939. PMLR.

Xin Wang, Yasheng Wang, Yao Wan, Jiawei Wang, Pingyi Zhou, Li Li, Hao Wu, and Jin Liu. 2022. Code-mvp: Learning to represent source code from multiple views with contrastive pre-training. *arXiv preprint arXiv:2205.02029*.

Xin Wang, Yasheng Wang, Pingyi Zhou, Fei Mi, Meng Xiao, Yadao Wang, Li Li, Xiao Liu, Hao Wu, Jin Liu, and Xin Jiang. 2021a. CLSEBERT: contrastive learning for syntax enhanced code pre-trained model. *CoRR*, abs/2108.04556.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021b. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *CoRR*, abs/2109.00859.

Lee Xiong, Chenyan Xiong, Ye Li, Kwok-Fung Tang, Jialin Liu, Paul N. Bennett, Junaid Ahmed, and Arnold Overwijk. 2021. Approximate nearest neighbor negative contrastive learning for dense text retrieval. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Yuanmeng Yan, Rumei Li, Sirui Wang, Fuzheng Zhang, Wei Wu, and Weiran Xu. 2021. Consert: A contrastive framework for self-supervised sentence representation transfer. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, pages 5065–5075. Association for Computational Linguistics.

Ziyu Yao, Daniel S. Weld, Wei-Peng Chen, and Huan Sun. 2018. Staqc: A systematically mined question-code dataset from stack overflow. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*, pages 1693–1703. ACM.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 476–486. ACM.

Hang Zhang, Yeyun Gong, Yelong Shen, Jiancheng Lv, Nan Duan, and Weizhu Chen. 2021. Adversarial retriever-ranker for dense text retrieval.

A Statistics of Bimodal Pairs

Language	# Code-Doc Pairs	# Code-Comment Pairs
Ruby	48,527	172,385
JavaScript	123,858	604,678
Go	315,921	172,385
Python	449,216	441,976
Java	452,847	404,424
PHP	520,088	301,708
Overall	2,137,293	1,964,627

Table 7: The statistics of code-text pairs in CodeRetriever.

B Code-Code Pairs Construction

Algorithm 1: Construct code-code pairs

Data: Paired text-code $(d_1, c_1), (d_2, c_2) \cdots, (d_m, c_m)$; Unpaired code data $c_1^*, c_2^* \cdots, c_n^*$.
Result: $CodePair$

DocMatcher $\leftarrow$ SimCSE($d_1 \cdots, d_m$);
NameMatcher $\leftarrow$ SimCSE($name_1 \cdots, name_n$);
$CodePair_{doc} \leftarrow$ [];
$CodePair_{name} \leftarrow$ [];
for $i \leftarrow 1 \cdots m$ **do**
  for $j \leftarrow i \cdots m$ **do**
    if $sim(d_i, d_j$,DocMatcher$) > \tau_1$ **then**
      $CodePair_{doc}$.append((c_i, c_j))
    end
  end
end
for $i \leftarrow 1 \cdots n$ **do**
  for $j \leftarrow i \cdots n$ **do**
    if $sim(name_i, name_j$,NameMatcher$) > \tau_1$ **then**
      $CodePair_{name}$.append((c_i^*, c_j^*))
    end
  end
end
$Filter \leftarrow$ CrossModel($CodePair_{doc}$)
$CodePair \leftarrow$ [];
for $c_i, c_j \in CodePair_{doc}$ **do**
  if $Filter(c_i, c_j) > \tau_2$ **then**
    $CodePair$.append((c_i, c_j))
  end
end
for $c_i^*, c_j^* \in CodePair_{name}$ **do**
  if $Filter(c_i^*, c_j^*) > \tau_2$ **then**
    $CodePair$.append((c_i^*, c_j^*))
  end
end

C The Hyper-parameters for building code-code pairs.

Hyper-parameters	Matcher	CrossModel
Initialization	GraphCodeBERT	GraphCodeBERT
Epoch	2	2
Batch	256	256
Learning Rate	2e-5	2e-5
Optimizer	AdamW	AdamW
Temperature	0.05	-
Positive Threshold	0.75	0.998

Table 8: The hyper-parameters of Matchers and Cross-Model.

D Statistics of Unimodal Pairs

Language	Ruby	JavaScript	Go	Python	Java	PHP
Ruby	354K	76K	38K	58K	78K	54K
JavaScript	239K	1936K	132K	158K	203K	155K
Go	181K	302K	3494K	146K	264K	123K
Python	512K	645K	305K	2038K	395K	316K
Java	380K	676K	445K	310K	4700K	388K
PHP	381K	575K	241K	246K	375K	2510K

Table 9: The statistics of code-code pairs in CodeRetriever.

E Implementation Details

CodeRetriever is a siamese-encoder model with shared code encoder and text encoder. CodeRetriever is initialized with pre-trained **GraphCodeBERT** checkpoint released by Guo et al. (2021), which is a 12 layers Transformer encoder, with hidden sizes of 768 and attention heads of 12. To save the number of model parameters, the text encoder and code encoder in CodeRetriever share their model weights during training which follows previous work (Feng et al., 2020; Guo et al., 2021). We use FAISS (Johnson et al., 2017) for efficient dense indexing/retrieval. i.e., accelerate the matching of similar function names and documentations. For NameMatcher, we normalize function names according to the naming patterns. For example, “openFile” with Camel-case and “open_file” with Snake-case are both normalized to “open file”. The overall training corpus for CodeRetreiver contains 2.1 million code-doc pairs, 23.4 million code-code pairs, and 1.9 million code-comment pairs. When a code has multiple positive text or code samples, we randomly sample one of them everytime during training. The CodeRetriever is trained with 8 NVIDIA Tesla V100s-32GB for 1.8 days. The batch-size, learning rate and training step are 256, 4e-5 and 100K, respectively. The max sequence length of the text and code is set as 128 and 320, respectively.

F Statistics of Fine-tuning Data

Dataset	Train	Dev	Test
CodeSearch-Ruby (Husain et al., 2019)	25K	1.4K	1.2K
CodeSearch-JS (Husain et al., 2019)	58K	3.9K	3.3K
CodeSearch-Go (Husain et al., 2019)	16.7K	7.3K	8.1K
CodeSearch-Python (Husain et al., 2019)	25K	13.9K	14.9K
CodeSearch-Java (Husain et al., 2019)	16.4K	5.2K	10.9K
CodeSearch-PHP (Husain et al., 2019)	24.1K	13.0K	14.0K
Adv (Lu et al., 2021)	28.0K	9.6K	19.2K
CoSQA (Huang et al., 2021)	19K	0.5K	0.5K
CoNaLa (Yin et al., 2018)	2.8K	-	0.8K
SO-DS (Heyman and Cutsem, 2020)	14.2K	0.9K	1.1K
StaQC (Yao et al., 2018)	20.4K	2.6K	2.7K

Table 10: The statistics of downstream datasets.

G Hyper-parameters of AR2

Hyper-Parameters	G	D
Initialization	GraphCodeBERT	GraphCodeBERT
Optimizer	AdamW	AdamW
Scheduler	Linear	Linear
Warmup proportion	0.1	0.1
Negative size	7	7
Batch size	128	128
Learning rate	5e-6	1e-6
Max step	16000	4000

Table 11: The Hyper-parameters of AR2