CodeRetriever: Unimodal and Bimodal Contrastive Learning for Code Search

Anonymous ACL submission

Abstract

In this paper, we propose the CodeRetriever model, which combines the unimodal and bimodal contrastive learning to train function-level code semantic representations, specifically for the code search task. For unimodal contrastive learning, we design a semantic-guided method to build positive 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 text-code pairs. Both contrastive objectives can fully leverage the large-scale code corpus for pre-training. Experimental results on several public benchmarks, (i.e., CodeSearch, CoSQA, etc.) demonstrate the effectiveness of CodeRetriever in the zero-shot setting. By fine-tuning with domain/language specified downstream data, CodeRetriever achieves the new state-of-the-art performance with significant improvement over existing code pre-trained models. We will make the code, model checkpoint, and constructed datasets publicly available.

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), GraphCodeBERT (Guo et al., 2021), could significantly improve the code search performance via self-supervised pre-training using large-scale code corpus (Husain et al., 2019).

However, existing code pre-training approaches directly 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 Java code. The second one is the cross-language representation issue. In CodeSearchNet corpus (Husain et al., 2019), it contains codes from six commonly used programming languages such as Python, Java, and etc. Since the code with mixed programming languages can hardly appear within the same con-

```
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.

text, 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 : unimodal contrastive loss and bimodal contrastive loss. The former encourages the model to learn the semantic relevance between code and code; the latter helps to model the relevance between code and text. Specifically, in the unimodal contrastive learning, it could provide cross-language code-code pairs with similar functionality as the training samples for optimizing CodeRetriever, which helps mitigate the cross-language representation issue. Through two contrastive learning objectives, CodeRetriever can explicitly model the function-level code semantic representation, which could alleviate the anisotropy representation issue (Gao et al., 2021b; Yan et al., 2021).

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 a code-code paired dataset by an unsupervised semantic-guided approach. Figure 1(a) shows a code-code example: two implementations of the Fibonacci number algorithm. To further take advantage of unpaired data (only code), we extract the code and in-line comment paired dataset to enhance the bimodal contrastive learning in CodeRetriever. Figure 1(b) shows an example to indicate that the in-line comment (comment shortly) is also semantically related to the code. In detail, the underlying logic of "if adjacent elements appear in descending order, swap them" corresponds to sort the input array into an ascending order.

The main contributions of this paper can be summarized as: 1) We propose the CodeRetriever model which leverages unimodal and bimodal contrastive learning for function-level code representation learning. 2) We construct large-scale code-to-comment and code-to-code datasets from CodeSearchNet by an unsupervised approach. The datasets will be publicly available to the research community. 3) Experimental results demonstrate that CodeRetriever achieves a new state-of-the-art performance on eleven code search datasets.

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). As described in (Husain et al., 2019), the document of the code is extracted from the function-header comments.

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 not optimized for code search scenarios.

In this work, we propose the CodeRetriever to learn the function-level code semantic presentation. As illustrated in Figure 2, CodeRetriever is initialized with 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.

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

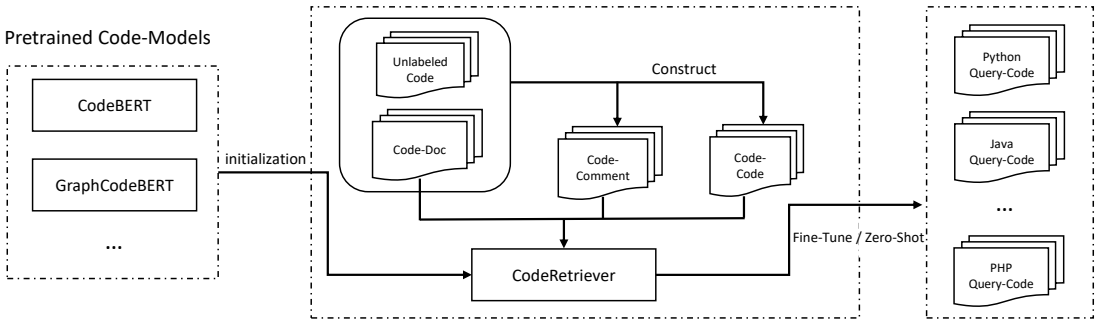


Figure 2: The illustration of the CodeRetreiver training pipeline.

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 $\langle \cdot \rangle$ 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, 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).

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. Figure 3 gives an example to show the unimodal/bimodal contrastive learning in CodeRetreiver.

3.3 Overall Objective

As illustrated in Figure 2, CodeRetreiver takes two types of text-to-code for bimodal contrsative 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 contrsative loss, respectively. The overall training objective for CodeRetreiver is:

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

where α, β and γ are three scalar values, we let $\alpha = \beta = \gamma = 1$ in our experiment.

4 Data Collection

In the section, we give the instruction on building the code-comment and code-code paired datasets from CodeSearchNet corpus.

4.1 Code-Comment

In-line comment as shown in Figure 1(b) can reflect the code's semantic, despite certain noisy signals. 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 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__", "__str__" etc, are removed.

After cleaning, we collect about 1.9 million code-comment pairs.

4.2 Code-Code

code-code paired datasets can provide explicit training signals for models to learn the semantic repre-

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

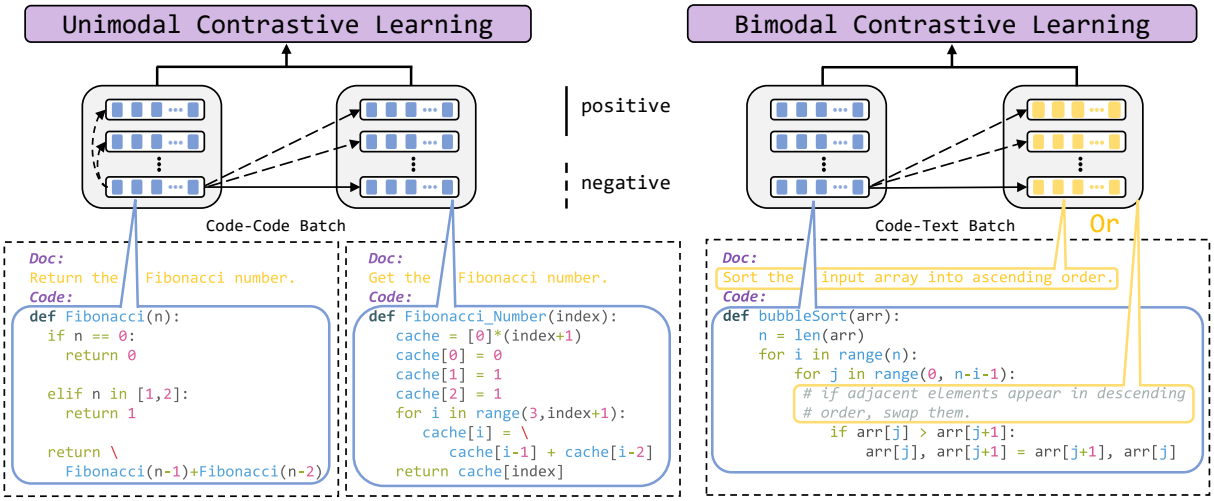


Figure 3: Unimodal and bimodal contrastive learning in CodeRetriever.

sentation of code across different programming languages. However, it is challenging to build large-scale and high-quality semantically relevant code-to-code pairs from an unlabeled corpus. Note that human annotation is usually costly and not scalable. By leveraging the unsupervised-learning techniques introduced in the following section, we collect a large-scale code-to-code corpus.

Step 1. we train two unsupervised SimCSE (Gao et al., 2021b) models on function names and documents, named "NameMatcher" and "DocMatcher", respectively.

Step 2. for a given function in the corpus, we retrieve its similar functions through function name using the "NameMatcher". We keep function pairs if their similarity scores are greater than threshold (0.75). After iterating over all the functions in the corpus, we obtain a paired code-code set denoted as $\mathbb{C}_{\text{Name}}$. One similar manner is applied to "DocMatcher", which retrieves code-code pairs from document-code corpus by matching their corresponding documents. The code-code pairs collected by "DocMatcher" is denoted as $\mathbb{C}_{\text{Doc}}$.

Step 3. we train a code-code binary classifier model M_c on $\mathbb{C}_{\text{Doc}}$, where negative code-pairs are randomly-sampled in batch.

Step 4. The code-code pairs with their prediction scores by M_c smaller than certain threshold are removed from $\mathbb{C}_{\text{Name}}$ and $\mathbb{C}_{\text{Doc}}$. Let $\mathbb{C}^*_{\text{Name}}$ and $\mathbb{C}^*_{\text{Doc}}$ be the cleaned subsets of $\mathbb{C}_{\text{Name}}$ and $\mathbb{C}_{\text{Doc}}$. The final code-code corpus is the joint of set $\mathbb{C}^*_{\text{Name}}$ and $\mathbb{C}^*_{\text{Doc}}$.

We provide a more detailed description on building code-code dataset in Appendix B. The collected code-code corpus contains 8.6 million pairs.

4.3 Implementation Details

The 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. We use FAISS (Johnson et al., 2017) to accelerate the matching of similar function names and documentations. The overall training corpus for CodeRetreiver contains 2.1 million code-doc pairs, 8.6 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 for it 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 of is 256, 4e-5 and 100K, respectively. The max sequence length of the text and code is set as 128 and 320, respectively.

5 Experiment

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 language each to evaluate models' comprehensive performance on various programming languages. The Adv dataset normalizes the method names and variable names in the dev/test set, which makes it more challenging. Dataset CoNaLa, SO-DS, and StaQC are collected from stackoverflow

questions, and queries in 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. The statistics of benchmark datasets are listed in Appendix A. We use Mean Reciprocal Rank (MRR) (Hull, 1999) as the evaluation metric on all datasets.

5.2 Experiment: Zero-Shot

To compare with existing code pre-trained models, we evaluate CodeRetriever on code search benchmarks in the zero-shot learning setting. In experiments, we take GraphCodeBERT (Guo et al., 2021) and ContraCode (Jain et al., 2020) for comparison. GraphCodeBERT is trained with token-level masked language model on CodeSearchNet corpus (Husain et al., 2019). Since GraphCodeBERT doesn't explicitly give the function-level representation, we take the hidden states of the "[CLS]" token of the last layer to represent the whole code/text, denoted as $\text{GraphCodeBERT}_{\text{cls}}$. Correspondingly, the average of hidden states over all tokens of the last layer is denoted as $\text{GraphCodeBERT}_{\text{avg}}$. We use inner product similarity to retrieve and measure the relevance between query and code. ContraCode (Jain et al., 2020) is specifically pre-trained only for JavaScript, which adopts a data augmentation approach to generate code-code pairs for contrastive learning.

5.2.1 Results

The top-half of Table 1 and Table 2 shows the performance of CodeRetriever on eleven code-search datasets without any language/domain specific fine-tuning. CodeRetriever significantly outperforms existing code pre-trained models on all datasets, which demonstrates that function-level code representation with contrastive learning is critical for code search tasks. We also report the performance of CodeRetriever trained with each contrastive loss individually in Table 1 and Table 2. As we can see, CodeRetriever with single unimodal contrastive loss: $\text{CodeRetreiver}_{\text{uni}}$, could not achieve good enough performance on zero-shot code search. But it still outperforms existing baseline approaches significantly. The CodeRetreiver model trained with combined unimodal and bimodal contrastive losses achieves the best performance on all datasets.

5.3 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.

5.3.1 Fine-tunning Approaches

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.

In-Batch Negative For a <query, code> pair in a batch, it uses other codes as negatives in the batch (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)

Hard Negative It can pick "hard" representative negative samples other than random negatives. We follow Gao et al. (2021a) for Hard Negative fine-tuning.

AR2 **A**dversarial **R**etriever-**R**anker is a recently proposed training framework for contrastive learning (Zhang et al., 2021). It adopts an adversarial-training approach to select "hard" negative samples iteratively.

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 compare CodeRetriever with existing code pretrained models: **CodeBERT** (Feng et al., 2020), pre-trained with MLM and replaced token detection tasks; **GraphCodeBERT** (Guo et al., 2021) integrates data flow of code as input tokens, pre-trained with MLM, data flow edge prediction and node alignment tasks. We also use its original pre-training objective further pre-train it, with the same steps as CodeRetriever, indicated by **GraphCodeBERT**$^{+}$; **SynCoBERT** (Wang et al., 2021a), pre-trained on code-AST pairs with contrastive learning.

5.3.2 Results

Table 1 and Table 2 show the performance of CodeRetriever and baseline methods on all benchmark datasets. First, we report the performance of CodeRetriever (In-Batch Negative), which uses

Lang	Ruby	Javascript	Go	Python	Java	PHP	Overall
Zero-Shot							
GraphCodeBERT$_{\text{avg}}$	0.6	0.5	0.4	0.2	0.2	0.2	0.35
GraphCodeBERT$_{\text{cls}}$	1.5	0.4	0.2	0.4	0.7	2.1	0.88
ContraCode	-	1.1	-	-	-	-	-
CodeRetriever ($\mathcal{L}_{\text{uni}}$)	33.3	26.0	53.8	23.5	27.9	20.7	30.9
CodeRetriever ($\mathcal{L}^1_{\text{bi}}$)	64.8	61.6	85.4	64.9	66.8	59.8	67.2
CodeRetriever ($\mathcal{L}^2_{\text{bi}}$)	59.3	50.8	64.8	49.3	50.0	39.1	52.2
CodeRetriever	**68.7**	**63.7**	**87.6**	**67.7**	**69.0**	**62.8**	**69.1**
Fine-Tuning							
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
GraphCodeBERT$^+$	70.7	64.8	89.6	69.2	69.2	64.7	71.4
CodeRetriever (In-Batch Negative)	75.3 (+5.0)	69.5 (+5.1)	91.6 (+1.9)	73.3 (+4.1)	74.0 (+4.9)	68.2 (+3.3)	75.3 (+4.0)
CodeRetriever (Hard Negative)	75.1 (+4.8)	69.8 (+5.4)	92.3 (+2.6)	74.0 (+4.8)	74.9 (+5.8)	69.1 (+4.2)	75.9 (+4.6)
CodeRetriever (AR2)	**77.1 (+6.8)**	**71.9 (+7.5)**	**92.4 (+2.7)**	**75.8 (+6.6)**	**76.5 (+7.4)**	**70.8 (+5.9)**	**77.4 (+6.1)**

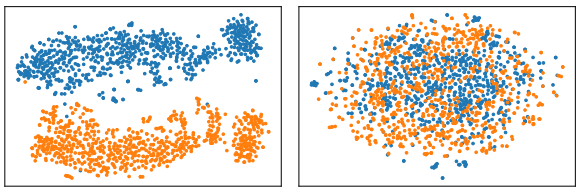
Table 1: The comparison on the CodeSearch dataset. We get the ContraCode's result by fine-tuning the released checkpoint (Jain et al., 2020). Other results of compared models are reported by previous papers (Feng et al., 2020; Guo et al., 2021; Wang et al., 2021a).

Dataset	Adv	CoSQA	CoNaLa	SO-DS	StaQC	Overall
Zero-Shot						
GraphCodeBERT$_{\text{avg}}$	1.1	0.2	0.6	0.2	0.3	0.48
GraphCodeBERT$_{\text{cls}}$	0.5	0.8	2.5	0.6	0.5	0.98
CodeRetriever ($\mathcal{L}_{\text{uni}}$)	15.9	16.1	8.1	4.0	4.6	9.7
CodeRetriever ($\mathcal{L}^1_{\text{bi}}$)	33.8	45.3	22.1	16.9	15.3	26.7
CodeRetriever ($\mathcal{L}^2_{\text{bi}}$)	32.7	39.6	24.2	16.4	15.1	25.6
CodeRetriever	**34.7**	**47.5**	**25.8**	**17.2**	**15.5**	**28.1**
Fine-Tuning						
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
GraphCodeBERT$^+$	35.9	67.4	23.7	25.2	24.1	35.3
CodeRetriever (In-Batch Negative)	43.0 (+7.8)	69.6 (+2.1)	29.6 (+6.1)	27.1 (+1.8)	**25.5 (+1.7)**	39.0 (+3.9)
CodeRetriever (Hard Negative)	45.1 (+9.9)	74.1 (+6.6)	**29.9 (+6.4)**	31.8 (+6.5)	24.6 (+0.8)	41.1 (+6.0)
CodeRetriever (AR2)	**46.9 (+11.7)**	**75.4 (+7.9)**	29.1 (+5.6)	**33.9 (+7.6)**	24.2 (+0.4)	**41.9 (+6.8)**

Table 2: The comparison on datasets which are closer to the real scenario. The results of CodeBERT, GraphCodeBERT and SyncoBERT on the Adv dataset are reported by previous papers, other results are from our implementation since they are not reported previously.

the same finetuning approach as other baselines to ensure a fair comparison. We can see that CodeRetriever obtains the best overall performance compared with all other baseline approaches. Specifically, CodeRetriever improves over GraphCodeBERT by 4.0 absolute points overall, which demonstrates the effectiveness of contrastive pre-training for code search while GraphCodeBERT$^+$ does not get significant improvement. Meanwhile, CodeRetriever outperforms the previous state-of-the-art SyncoBERT (Wang et al., 2021a) model on all tasks with reported results.

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-tolerance than AR2 and Hard-Negative.



(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.

5.4 Analysis

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 number of training examples is varied from 500 to FULL (19K). We can see that CodeRetriever could reach reasonable performance in the low-resource setting.

Cross-Language Code Search In this setting, we finetune the code pre-trained models 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.

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). However, 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.

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 the results are shown in in table 5. We see that CodeRetriever achieves better performance compared to other baselines, which demonstrates its scalability and potentiality for other code understanding tasks.

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.

Train Size	500	1000	2000	4000	FULL
GraphCodeBERT	43.2	49.9	54.0	57.2	67.5
CodeRetriever	54.7	55.6	58.1	60.1	69.6

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

Uniformity and Alignment To study the effect of CodeRetriever on the sequence-level representation space, we use the alignment and uniformity metrics (Wang and Isola, 2020) to see sequence-level representation distribution changes during training, shown in Figure 5. We see that the uniformity loss of CodeRtriever descend 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.

5.5 Ablation Study

To understand the effect of each component in CodeRetriever, we conduct ablation study on the CodeSearch Java dataset and SO-DS. We add the components of CodeRetriever to the initial model one-by-one. We find that using code-code pairs without denoising for unimodal contrastive learning brings performance degradation. And with

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
Boost (Ding et al., 2021)	82.77
Corder (Bui et al., 2021)	84.10
CodeRetriever	**88.85**

Table 5: The performance compraison on the code-to-code retrieval task (Mou et al., 2016; Lu et al., 2021).

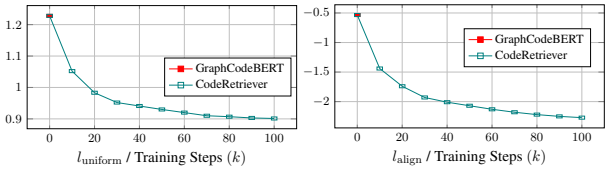


Figure 5: The alignment and uniformity during pre-training.

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.

denoising, it achieves performance improvement. This demonstrates the effectiveness of the denoising step and illustrates 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 construction 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.

6 Related Work

6.1 Token-Level Code Pre-training

Token-level pre-trained models have been widely-used for the programming languages (Kanade et al., 2020; Karampatsis and Sutton, 2020; Buratti et al., 2020; Feng et al., 2020; Guo et al., 2021). 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, and T5, respectively. However, token-level objectives cause the anisotropy problem and have a gap with code search which is based on sequence-level representations. Different from these works, CodeRetriever utilizes the contrastive-learning framework to enhance the sequence-level representation.

6.2 Contrastive Learning for Code

Recently, several works try to use contrastive learning on the programming language. The key of contrastive learning is building effective positive or negative samples. ContraCode (Jain et al., 2020) and Corder (Bui et al., 2021) use the semantics-preserving transformation, such as identifier renaming and dead code insertion to build positive pairs. Ding et al. (2021) develop bug-injection to build hard negative pairs. The codes constructed from these methods are generally unnatural and very different from the real code. SynCoBERT (Wang et al., 2021a) uses the code and its AST/documentation as positive pair. In CodeRetriever, we construct the positive pairs from code-code, code-documentation, and code-comment. For the code-code, we design a more natural and diverse positive pairs construction method based on codes from real world.

7 Conclusion

In this paper, we introduce CodeRetriever which combines the 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 proposed CodeRetriever brings significant improvement and achieves the new state-of-the-art performance on all benchmarks for both zero-shot and downstream data fine-tuning settings. Further analysis results demonstrate the CodeRetriever are also powerful on low resource and cross-language code search tasks.

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, 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.

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.

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.

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.

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, 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 Benchmark Datasets

Dataset	Training	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 7: The statistics of benchmark datasets.

B Code-Code Pairs Building

Algorithm 1: Construct code-code pairs

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

DocMatcher $\leftarrow$ SimCSE($d_1 \cdots, d_m$);
NameMatcher $\leftarrow$ SimCSE($name_1 \cdots, name_n$);
$CodePair_d \leftarrow []$;
$CodePair_n \leftarrow []$;
for $i \leftarrow 1 \cdots m$ **do**
  for $j \leftarrow i \cdots m$ **do**
    if *sim*$(d_i, d_j,$DocMatcher$)>\tau_1$ **then**
      $CodePair_d$.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**
      $CodePair_n$.append((c_i, c_j))
    end
  end
end
$Filter \leftarrow$ CrossModel($CodePair_d$)
$CodePair \leftarrow []$;
for $c_i, c_j \in CodePair_d$ **do**
  if *Filter*$(c_i, c_j) > \tau_2$ **then**
    $CodePair$.append((c_i, c_j))
  end
end
for $c_i, c_j \in CodePair_n$ **do**
  if *Filter*$(c_i, c_j) > \tau_2$ **then**
    $CodePair$.append((c_i, c_j))
  end
end