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

Xiaonan Li†, Yeyun Gong2, Yelong Shen2, Xipeng Qiu††, Hang Zhang2, Bolun Yao2, Weizhen Qi2, Daxin Jiang2, Weizhu Chen2, Nan Duan2
1 Shanghai Key Laboratory of Intelligent Information Processing, Fudan University
1 School of Computer Science, Fudan University 2Microsoft
†{lixn20, xpqiu}@fudan.edu.cn,
‡{yegong, yeshe, v-zhbang, 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.

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

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

raw text
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 CodeRetriever. 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 CodeRetriever, 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.
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:

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

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

where $\langle \cdot, \cdot \rangle$ 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:

$$L_{\text{uni}} = -\ln \frac{\exp(\tau s(c, c^+))}{\sum_{c' \in C} \exp(\tau s(c, c'))},$$

where $\tau$ is the temperature, for simplicity, we let $\tau = 1$; set $C$ 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:

$$L_{\text{bi}} = -\ln \frac{\exp(\tau s(t, c^+))}{\sum_{c' \in C} \exp(\tau s(t, c'))},$$

where the definitions of $\tau$ and $C$ 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, CodeRetriever takes two types of text-to-code for bimodal contrastive training, which are code-document and code-comment. Therefore, we use $L_{\text{bi}}^1$ and $L_{\text{bi}}^2$ to denote code-document and code-comment contrastive loss, respectively. The overall pre-training objective for CodeRetriever is:

$$L(\theta, \phi) = L_{\text{uni}} + L_{\text{bi}}^1 + L_{\text{bi}}^2$$

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
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 ...”\(^1\); 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 matching 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 \( \mathcal{C}_{\text{Doc}} \), and use \( \mathcal{C}_{\text{Name}} \) 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 \( \mathcal{C}_{\text{Name}} \) and \( \mathcal{C}_{\text{Doc}} \) collected from Step 1 can be noisy, especially for \( \mathcal{C}_{\text{Name}} \) as functions with the same function name can have different functionalities. In this step, we train a binary classifier model, CrossModel (\( \mathcal{M}_{c} \)), for filtering noisy code-code pairs. 1) We take the code-code pairs \( \mathcal{C}_{\text{Doc}} \), which is less noisy, as the training set to train the CrossModel \( \mathcal{M}_{c} \). It takes

---

\(^1\) Linter is a static analysis tool for checking code.
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.

<table>
<thead>
<tr>
<th>Lang</th>
<th>Ruby</th>
<th>Javascript</th>
<th>Go</th>
<th>Python</th>
<th>Java</th>
<th>PHP</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>ContraCode (Jain et al., 2020)</td>
<td>30.6</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>74.0</td>
</tr>
<tr>
<td>SyncoBERT (Wang et al., 2021a)</td>
<td>72.2</td>
<td>67.7</td>
<td>91.3</td>
<td>72.4</td>
<td>72.3</td>
<td>67.8</td>
<td>74.0</td>
</tr>
<tr>
<td>CodeBERT (Feng et al., 2020)</td>
<td>67.9</td>
<td>62.0</td>
<td>88.2</td>
<td>67.2</td>
<td>67.6</td>
<td>62.8</td>
<td>69.3</td>
</tr>
<tr>
<td>GraphCodeBERT (Guo et al., 2021)</td>
<td>70.3</td>
<td>64.4</td>
<td>89.7</td>
<td>69.2</td>
<td>69.1</td>
<td>64.9</td>
<td>71.3</td>
</tr>
<tr>
<td>CodeRetriever (In-Batch Negative)</td>
<td>75.3</td>
<td>69.5</td>
<td>91.6</td>
<td>73.3</td>
<td>74.0</td>
<td>68.2</td>
<td>75.3</td>
</tr>
<tr>
<td>CodeRetriever (Hard Negative)</td>
<td>75.1</td>
<td>69.8</td>
<td>92.3</td>
<td>74.0</td>
<td>69.1</td>
<td>69.1</td>
<td>75.9</td>
</tr>
<tr>
<td>CodeRetriever (AR2)</td>
<td>77.1</td>
<td>71.9</td>
<td>92.4</td>
<td>75.8</td>
<td>76.5</td>
<td>70.8</td>
<td>77.4</td>
</tr>
</tbody>
</table>

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.

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

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.  

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 based 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-preserving code transformation on Javascript corpus. **UniXcoder** (Guo et al., 2022) is adapted from UniLM and pre-trained on unified cross-modal data like code, AST and text.

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 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 hyper-parameters 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.

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 CodeRetriever 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-
uniform / Training Steps (k)

Figure 5: The alignment and uniformity curve.

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).

Table 6: Ablation study.

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
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


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.


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).


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


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.


A Statistics of Bimodal Pairs

<table>
<thead>
<tr>
<th>Language</th>
<th># Code-Doc Pairs</th>
<th># Code-Comment Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ruby</td>
<td>48,527</td>
<td>172,385</td>
</tr>
<tr>
<td>JavaScript</td>
<td>123,858</td>
<td>604,678</td>
</tr>
<tr>
<td>Go</td>
<td>315,921</td>
<td>441,976</td>
</tr>
<tr>
<td>Python</td>
<td>452,847</td>
<td>404,424</td>
</tr>
<tr>
<td>Java</td>
<td>520,088</td>
<td>301,708</td>
</tr>
<tr>
<td>PHP</td>
<td>2,137,293</td>
<td>1,964,627</td>
</tr>
</tbody>
</table>

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

B Code-Code Pairs Construction

Algorithm 1: Construct code-code pairs

Data: Paired text-code

\((d_1, c_1), (d_2, c_2), \ldots, (d_m, c_m)\); Unpaired
code data \(c_1^*, c_2^*, \ldots, c_n^*\).

Result: CodePair

1. DocMatcher ← SimCSE\((d_1, \ldots, d_m)\);
2. NameMatcher ← SimCSE\((name_1, \ldots, name_n)\);
3. CodePair_doc ← [];
4. CodePair_name ← [];
5. for \(i ← 1 \ldots m\) do
6.   for \(j ← i \ldots m\) do
7.     if \(\text{sim}(d_i, d_j, \text{DocMatcher}) > \tau_1\) then
8.       CodePair_doc.append((\(c_i, c_j\)))
9.     end
10. end
11. end
12. for \(i ← 1 \ldots n\) do
13.   for \(j ← i \ldots n\) do
14.     if \(\text{sim}(name_i, name_j, \text{NameMatcher}) > \tau_1\) then
15.       CodePair_name.append((\(c_i^*, c_j^*\)))
16.     end
17. end
18. end
19. Filter ← CrossModel(CodePair_doc)
20. CodePair ← [];
21. for \(c_i, c_j \in \text{CodePair}_\text{doc}\) do
22.   if \(\text{Filter}(c_i, c_j) > \tau_2\) then
23.     CodePair.append((\(c_i, c_j\)))
24. end
25. end
26. for \(c_i^*, c_j^* \in \text{CodePair}_\text{name}\) do
27.   if \(\text{Filter}(c_i^*, c_j^*) > \tau_2\) then
28.     CodePair.append((\(c_i^*, c_j^*\)))
29. end
30. end

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

<table>
<thead>
<tr>
<th>Hyper-parameters</th>
<th>Matcher</th>
<th>CrossModel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>GraphCodeBERT</td>
<td>GraphCodeBERT</td>
</tr>
<tr>
<td>Epoch</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Batch</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>Learning Rate</td>
<td>2e-5</td>
<td>2e-5</td>
</tr>
<tr>
<td>Optimizer</td>
<td>AdamW</td>
<td>AdamW</td>
</tr>
<tr>
<td>Temperature</td>
<td>0.05</td>
<td>0.998</td>
</tr>
<tr>
<td>Positive Threshold</td>
<td>0.75</td>
<td>0.998</td>
</tr>
</tbody>
</table>

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

D Statistics of Unimodal Pairs

<table>
<thead>
<tr>
<th>Language</th>
<th>Ruby</th>
<th>JavaScript</th>
<th>Go</th>
<th>Python</th>
<th>Java</th>
<th>PHP</th>
</tr>
</thead>
<tbody>
<tr>
<td># Code-Doc Pairs</td>
<td>354K</td>
<td>76K</td>
<td>38K</td>
<td>58K</td>
<td>78K</td>
<td>54K</td>
</tr>
<tr>
<td># Code-Comment Pairs</td>
<td>239K</td>
<td>1936K</td>
<td>132K</td>
<td>158K</td>
<td>203K</td>
<td>155K</td>
</tr>
<tr>
<td># Code-Code Pairs</td>
<td>181K</td>
<td>302K</td>
<td>3494K</td>
<td>146K</td>
<td>264K</td>
<td>123K</td>
</tr>
</tbody>
</table>

Table 9: The statistics of code-code pairs in CodeRe-
triever.

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 CodeRetriever 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

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

Table 10: The statistics of downstream datasets.

G Hyper-parameters of AR2

<table>
<thead>
<tr>
<th>Hyper-Parameters</th>
<th>G</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>GraphCodeBERT</td>
<td>GraphCodeBERT</td>
</tr>
<tr>
<td>Optimizer</td>
<td>AdamW</td>
<td>AdamW</td>
</tr>
<tr>
<td>Scheduler</td>
<td>Linear</td>
<td>Linear</td>
</tr>
<tr>
<td>Warmup proportion</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>Negative size</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Batch size</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>Learning rate</td>
<td>5e-6</td>
<td>1e-6</td>
</tr>
<tr>
<td>Max step</td>
<td>16000</td>
<td>4000</td>
</tr>
</tbody>
</table>

Table 11: The Hyper-parameters of AR2