# Soft-Labeled Contrastive Pre-training for Function-level Code Representation

**Xiaonan Li[1]*, Daya Guo[2]*, Yeyun Gong[2], Yun Lin[3], Yelong Shen[2], Xipeng Qiu[1]†, 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 [3]National University of Singapore

[1]{lixn20, xpqiu}@fudan.edu.cn, [2]{t-dayaguo, yegong, yeshe, djiang, wzchen, nanduan}@microsoft.com, [3]dcsliny@nus.edu.sg

## Abstract

Code contrastive pre-training has recently achieved significant progress on code-related tasks. In this paper, we present **SCodeR**, a **S**oft-labeled contrastive pre-training framework with two positive sample construction methods to learn functional-level **Code R**epresentation. Considering the relevance between codes in a large-scale code corpus, the soft-labeled contrastive pre-training can obtain fine-grained soft-labels through an iterative adversarial manner and use them to learn better code representation. The positive sample construction is another key for contrastive pre-training. Previous works use transformation-based methods like variable renaming to generate semantically equal positive codes. However, they usually result in the generated code with a highly similar surface form, and thus mislead the model to focus on superficial code structure instead of code semantics. To encourage SCodeR to capture semantic information from the code, we utilize code comments and abstract syntax subtrees of the code to build positive samples. We conduct experiments on four code-related tasks over seven datasets. Extensive experimental results show that SCodeR achieves new state-of-the-art performance on all of them, which illustrates the effectiveness of the proposed pre-training method.

## 1 Introduction

Function-level code representation learning aims to learn continuous distributed vectors that represent the semantics of code snippets (Alon et al., 2019), which has led to dramatic empirical improvements on a variety of code-related tasks such as code search, clone detection, code summarization, etc. To learn function-level code representation on unlabeled code corpus with self-supervised objectives, recent works (Jain et al., 2021; Bui et al., 2021; Ding et al., 2022; Wang et al., 2022a) propose contrastive pre-training methods for programming language. In their contrastive pre-training, they usually pull positive code pairs together in representation space and regard different codes as negative pairs via pushing their representation apart. However, they ignore the potential relevance between codes since different programs in a large code corpus may have some similarities. For example, an ascending sort program and a descending sort program are somewhat similar since they both sort their input in a certain order. More seriously, there are a lot of duplications in code corpus (Lopes et al., 2017; Allamanis, 2019), which can cause the "false negative" problem and deteriorate the model (Huynh et al., 2022; Chen et al., 2021b). The other problem of current code contrastive pre-training methods is their positive sample construction. ContraCode (Jain et al., 2021) and Corder (Bui et al., 2021) design code transformation algorithms like variable renaming and dead code insertion to generate semantically equivalent programs as positive samples, while Code-MVP (Wang et al., 2022a) leverages code structures like abstract syntax tree (AST) and control flow graphs (CFG) to transform a program to different variants. However, as shown in Figure 1, these methods usually result

**Original Code**

```
# Sort the input array
def bubbleSort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0,n-i-1): conditional statement
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

**Variable Renaming**

```
# Sort the input array
def Fun(v1):
    v2 = len(v1)
    for v3 in range(v2):
        for v4 in range(0,v2-v3-1): conditional statement
            if v1[v4] > v1[v4+1]:
                v1[v4], v1[v4+1] = v1[v4+1], v1[v4]
```

Figure 1: An example of applying variable renaming.

*Equal contribution and work is done during the internship at Microsoft Research Asia.

†Corresponding author.



*Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 118 - 129

in generated positive samples with highly similar structures (e.g. double loop statements with a conditional statement) to the original program. To pull such positive pairs closer in representation space, the model will tend to learn function-level code representation from superficial code structure rather than substantial code semantics. To address these limitations, we present **SCodeR**, a **S**oft-labeled contrastive pre-training framework with two positive sample construction methods to learn function-level **Code R**epresentation.

The soft-labeled contrastive pre-training framework can obtain relevance scores between samples and the original program as soft-labels in an iterative adversarial manner to improve code representation. Specifically, we first leverage hard-negative samples from contrastive pre-training to fool discriminators that can explore finer-grained token-level interactions, while discriminators learn to distinguish them and predict relevance scores among samples as soft-labels for contrastive pre-training. Through this adversarial iteration, discriminators can provide progressive feedback to improve code contrastive pre-training through soft-labels.

As for positive sample construction, we propose to utilize code comment and abstract syntax subtree of the source code to construct positive samples for SCodeR pre-training. Generally, user-written code comments highly describe the function of a source code like "sort the input array" in Figure 1, which provides crucial semantic information for the model to capture code semantics. Besides the comment, the code itself also contains rich information. To further explore the intra-code correlation and contextual knowledge for code contrastive pre-training, we randomly select a piece of code via AST like the conditional statement of Figure 1 and its context as a positive pair. These positive pairs require the model to understand code semantics and learn to infer the selected code based on its context and can help the model learn representation from code semantics.

We evaluate SCodeR on four code-related downstream tasks over seven datasets, including code search, clone detection, zero-shot code-to-code search, and markdown ordering in python notebooks. Results show that SCodeR achieves state-of-the-art performance and ablation studies demonstrate the effectiveness of positive sample construction and soft-labeled contrastive pre-training. We release the codes and resources at https://github.com/microsoft/AR2/tree/main/SCodeR.

## 2 Related Works

**Pre-trained Models for Programming Language.** With the great success of pre-trained models in natural language processing field (Devlin et al., 2018; Lewis et al., 2019; Raffel et al., 2019; Brown et al., 2020), recent works attempt to apply pre-training techniques on programming languages to facilitate the development of code intelligence. Kanade et al. (2019) pre-train CuBERT on a large-scale Python corpus using masked language modeling (MLM) and next sentence prediction objectives. Feng et al. (2020) pre-train CodeBERT on code-text pairs in six programming languages via MLM and replaced token detection objectives to support text-code related tasks such as code search. GraphCodeBERT (Guo et al., 2020) leverages data flow as additional semantic information to enhance code representation. To support code completion, Svyatkovskiy et al. (2020) and Lu et al. (2021) respectively propose GPT-C and CodeGPT. Both of them are decoder-only models and pre-trained by unidirectional language modeling. Some recent works (Ahmad et al., 2021; Wang et al., 2021; Guo et al., 2022) explore unified pre-trained models to support both understanding and generation tasks. PLBART (Ahmad et al., 2021) and CodeT5 (Wang et al., 2021) are based on the encoder-decoder framework. PLBART uses denoising objective to pre-train the model and CodeT5 considers the crucial token type information from identifiers. However, these pre-trained models usually result in poor function-level code representation (Guo et al., 2022) due to the anisotropy representation issue (Li et al., 2020). In this work, we mainly investigate how to learn function-level code semantic representations.

**Contrastive Pre-training for Code Representation.** To learn function-level code semantic representation, several attempts have been made to leverage contrastive pre-training on programming languages. ContraCode (Jain et al., 2021) and Corder (Bui et al., 2021) design transformation algorithms like variable renaming and dead code insertion to generate semantically equivalent programs as positive instances, while Ding et al. (2022) design structure-guided code transformation algorithms that inject real-world security bugs to build hard negative pairs for contrastive pre-training. Instead of using semantic-preserving program transformations, SynCoBERT (Wang et al., 2022b) and Code-

MVP (Wang et al., 2022a) construct the positive pairs through the compilation process of programs like AST and CFG. However, these works usually generate positive samples with highly similar structures as the original program. To distinguish these positive samples from candidates, the model might learn code representation from code surface forms according to hand-written patterns, instead of code semantics. In this paper, we propose to utilize the comment and abstract syntax sub-tree of the code to construct positive samples and present a method to obtain relevance scores among samples as soft-labels for contrastive pre-training.

## 3 Positive Sample Construction

In this section, we describe how to construct positive pairs for SCodeR. Different from previous works that design transformation algorithms to generate semantically equivalent but highly similar programs, we propose to leverage comment and abstract syntax sub-tree of the code for positive sample construction to encourage the model to capture semantic information.

### 3.1 Code Comment

User-written code comments usually summarize the functionality of the codes and provide crucial semantic information about the source code. Taking the code in Figure 2 as an example, the comment "sort the input array" highly describes the goal of the code and can help the model to understand code semantics from the natural language. Therefore, we take source code $c$ with the corresponding comment $t$ as positive pair $(t, c)$. Such positive pairs not only enable the model to understand the code semantics but also align the representation of different programming languages with a unified natural language description as a pivot.

### 3.2 Abstract Syntax Sub-Tree

Besides the comment, the code itself also contains rich information. To further explore the intra-code correlation and contextual knowledge for contrastive pre-training, we propose a method, called **A**bstract **S**yntax **S**ub-**T**ree Extraction (**ASST**), that leverages the abstract syntax sub-tree of the source code to construct positive code pairs. We give an example of a Python code with its AST in Figure 2. We first randomly select the sub-tree of the AST like "if statement", and then take the corresponding code of the sub-tree and the remaining code as positive code pairs. The procedure of extraction is

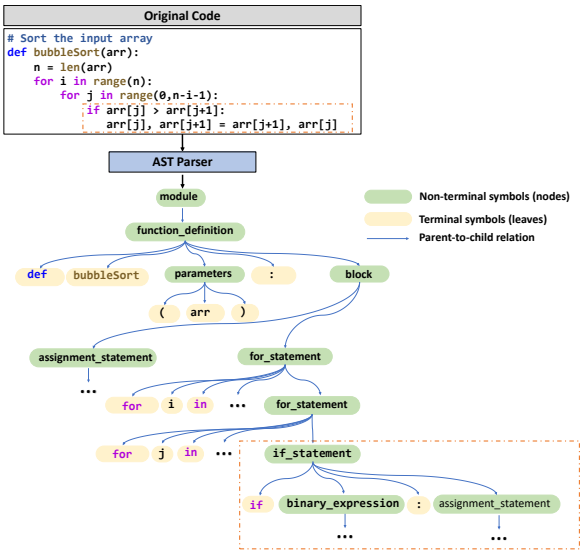


Figure 2: An ASST example of bubble sort.

illustrated in Algorithm 1. Specifically, we first predefine a set $N$ of node types whose sub-trees can be used to construct positive pairs. The set mainly consists of statement-level types like "for_statement" that usually contain a complete and functional code snippet. We then start from a randomly selected leaf node (line 1-2) and find an eligible node in the pre-defined set $N$ along the direction of the root node (line 3-10). Finally, we take the corresponding code $s$ (i.e. leaf children) of the eligible node and the remaining code context $\tilde{s}$ as a positive code pairs $(s, \tilde{s})$ for contrastive pre-training. To avoid extracting those code spans that are too short or meaningless, we set a minimum length $l_{min}$ for the extracted code spans $s$.

While transformation-based methods generate programs with similar structures, the structures of positive code pairs generated by ASST are different since they belong to different parts of a program. Meanwhile, they are logically relevant because they compose a program of full functionality. To estimate which code is complementary to a given code context in contrastive pre-training, the model needs to understand code semantics and learn to reason based on its context, which encourages the model to understand code semantics.

There are similar mechanisms to learn text representation such as Inversed Cloze Task (ICT) (Lee et al., 2019) that takes a random span of natural language tokens and their context as a positive pair. However, ICT cannot be directly applied to code because code has an explicit structure. If we randomly select code spans on token-level, the selected code

**Algorithm 1** Abstract Syntax Sub-Tree Extraction

**Require:** The AST $T$ of a code $c$ and pre-defined selectable node types $N$.
Collect leaf children $C$ of nodes whose types are in $N$
Randomly sample a node $a$ from $C$
**while** True **do**
  $s \leftarrow$ the corresponding code of $a$
  **if** $length(s) \geq l_{min}$ and $a \in N$ **then**
    **return** $s$
  **else**
    $a \leftarrow$ the parent node of $a$
  **end if**
**end while**

spans might be ungrammatical such as *"for i"*, which will mislead the model to focus on structural matching rather than semantic matching.

## 4 Soft-Labeled Contrastive Pre-training

Previous code contrastive pre-training methods usually take different programs in a code corpus as negative pairs and push them apart in the representation space. However, different programs in an unlabeled code corpus may have some similarities. Taking a program that sorts the input in ascending order as an example, even though another "descendingly sort" program is not semantically equal with it, they both sort their input in a certain order and thus are somewhat similar. Another problem is the "false negative" issue (Huynh et al., 2022; Chen et al., 2021b) due to the duplication in the code corpus (Lopes et al., 2017; Allamanis, 2019). To alleviate these problems, we propose soft-labeled contrastive pre-training framework that uses relevance scores between different samples as soft-labels to learn function-level code representation.

### 4.1 Overview

The soft-labeled contrastive pre-training framework involves three components: (1) A dual-encoder $G_\theta$ that aims to learn function-level code representation (2) Two discriminators $D_\phi$ and $D_\psi$ that calculate relevance scores between two inputs for text-code and code-code pairs, respectively. These components compute the similarity between two samples $(x, y)$ as follows:

$$G_\theta(x, y) = E_\theta(x)^T E_\theta(y) \quad (1)$$

$$D_\phi(x, y) = \mathbf{w}_\phi{}^T E_\phi([x; y]) \quad (2)$$

$$D_\psi(x, y) = \mathbf{w}_\psi{}^T E_\psi([x; y]) \quad (3)$$

where $E_\theta$, $E_\phi$ and $E_\psi$ are multi-layer Transformer (Vaswani et al., 2017) encoders with mean-pooling. $w_\phi$ ($w_\psi$) is a linear layer to obtain similarity score

**Algorithm 2** Soft-Labeled contrastive pre-training

**Require:** A dual-encoder $G_\theta$, two discriminators $D_{\phi(\psi)}$, and a set $X$ of positive pairs with a unlabeled code corpus $C$.
1: Initialize the dual-encoder and discriminators.
2: Train the warm-up dual-encoder.
3: Get top-$K$ negative codes $C^{x_k}_{hard}$ from $C$ for each positive pair $(x_k, x^+_k) \in X$ using the dual-encoder.
4: **for** $i$ in $1 \cdots I$ **do**
5:  **for** Discriminators training step **do**
6:  Sample hard negative codes from $C^{x_k}_{hard}$.
7:  Update parameters of discriminators $D_\phi$ and $D_\psi$.
8:  **end for**
9:  **for** Dual-encoder training step **do**
10:  Sample hard negative codes from $C^{x_k}_{hard}$ and obtain relevance scores from discriminators.
11:  Update parameters of the dual-encoder $G_\theta$.
12:  **end for**
13:  Refresh Top-$K$ negative codes $C^{x_k}_{hard}$ using new $G_\theta$.
14: **end for**

and $[\cdot;\cdot]$ indicates the concatenation operator. If the input $(x, y)$ is a text-code pair, we use $D_\phi$ to calculate the similarity, otherwise we use $D_\psi$.

While the dual-encoder encodes samples separately, discriminators take the concatenation of two samples as the input and fully explore finer-grained token-level interactions through the self-attention mechanism, which can predict more accurate relevance scores between two samples. Therefore, we propose to utilize relevance scores from discriminators as soft-labels to help the encoder $E_\theta$ learn better code representation.

We show the detailed illustration of our proposed soft-labeled contrastive pre-training in Algorithm 2. Specifically, we first initialize all encoders with a pre-trained model like UniXcoder (Guo et al., 2022) and follow Li et al. (2022) to train a warm-up dual-encoder using a simple strategy where negative samples come from other positive pairs in the same batch $\mathbb{X}_b$ (line 1-2 of Algorithm 2). The loss is calculated as follows,

$$p_\theta(x^+|x, \mathbb{X}_b) = \frac{e^{G_\theta(x,x^+)}}{\sum_{x' \in \mathbb{X}_b} e^{G_\theta(x,x')}} \quad (4)$$

$$L^\theta_{warm} = -\log p_\theta(x^+|x, \mathbb{X}_b), \quad (5)$$

where $(x, x^+) \in X$ is a positive pair as described by Section 3.

We then iteratively alternate two training procedures: (1) The dual-encoder is used to obtain hard-negative codes to train the discriminators (line 5-8). (2) The optimized discriminators predict relevance scores among samples as soft-labels to improve the dual-encoder (line 9-12). Through this iterative training, the dual-encoder gradually produces

harder negative samples to train better discriminators, whereas the discriminators provide better progressive feedback to improve the dual-encoder. The details about training procedures for the discriminators and dual-encoder will be described next.

### 4.2 Discriminators Training

Given a text $x$ from positive text-code pairs $(x, x^+)$, the discriminator $D_\phi$ is optimized by maximizing the log likelihood of selecting positive code $x^+$ from candidates $\mathbb{X}$ as follows,

$$p_\phi(x^+|x, \mathbb{X}) = \frac{e^{D_\phi(x,x^+)}}{\sum_{x' \in \mathbb{X}} e^{D_\phi(x,x')}} \quad (6)$$

$$L^\phi = -log p_\phi(x^+|x, \mathbb{X}), \quad (7)$$

where $\mathbb{X}$ is the set of negative codes $\mathbb{X}^-$ with a positive code $x^+$. If $x$ is a code from positive code-code pairs, the calculation of $p_\psi$ and $L^\psi$ are analogous to $p_\phi$ and $L^\phi$, respectively.

To better train discriminators, we take those hard-negative examples that are not positive samples but closed to the original example $x$ in the vector space as the negative candidates $\mathbb{X}^-$. In practice, we first get the top-K code samples that are closest to $x$ using $G_\theta$ as the distance function and randomly sample examples from them to obtain a subset $\mathbb{X}^-$.

### 4.3 Dual-Encoder Training

After training discriminators, we utilize relevance scores predicted by discriminators as soft-labels and follow Zhang et al. (2021) to use adversarial and distillation losses to optimize the dual-encoder.

**Adversarial loss:**

$$L^\theta_{adv} = -\sum_{x^- \in \mathbb{X}^-} w(x^-) * \log p_\theta(x^-|x, \mathbb{X}^-) \quad (8)$$

where $w(x^-)$ is $-\log p_\phi(x^+|x, \{x^+, x^-\})$ if $x$ is a text otherwise $w$ is $-\log p_\psi(x^+|x, \{x^+, x^-\})$. We apply the same approach to obtain hard-negative candidates $\mathbb{X}^-$ as described in Section 4.2.

When optimizing $G_\theta$, $w$ in Equation 8 is a constant and adjusts weight for each negative example. When $-\log p_{\phi(\psi)}(x^+|x, \{x^+, x^-\})$ is small, i.e. discriminators predict that $x$ and $x^-$ are semantically relevant, $w$ will be a high weight and force $G_\theta$ to draw the representation of $x$ and $x^-$ closer among $\mathbb{X}^-$. Since we optimize the dual-encoder on negative codes under different weight $w$, the representation of negative codes with high relevance score will be closer to $x$, and those with low relevance score will be pushed away.

**Distillation loss:**

$$L^\theta_{distill} = H(p_{\phi(\psi)}(\cdot|x, \mathbb{X}), p_\theta(\cdot|x, \mathbb{X})) \quad (9)$$

We also use a distillation loss function (Hinton et al., 2015) to encourage the dual-encoder to fit the probability distribution of discriminators over $\{x^+\} \cup \mathbb{X}^-$ using KL divergence loss $H$. Through $L^\theta_{distill}$, we can inject discriminators' knowledge into the dual-encoder by soft-labels $p_{\phi(\psi)}$.

**Training Objective of Dual-Encoder** The overall loss function of the dual-encoder is the integration of adversarial loss and distillation loss as follows, where $\lambda$ is a pre-defined hyper-parameter.

$$L^\theta = \lambda * L^\theta_{adv} + (1 - \lambda) * L^\theta_{distill} \quad (10)$$

Through $L_\theta$, we can provide discriminators' progressive feedback to the dual-encoder through soft-labels. After this adversarial iteration, we will use $E_\theta$ to serve for downstream tasks.

## 5 Experiment

### 5.1 Model Comparison

We compare SCodeR with various state-of-the-art pre-trained models. **RoBERTa** (Liu et al., 2019) is pre-trained on text corpus by masked language model (MLM). **CodeBERT** (Feng et al., 2020) is pre-trained on large scale code corpus with MLM and replaced token detection. **GraphCodeBERT** (Guo et al., 2020) is based on CodeBERT and integrates the data flow information to enhance code representation. **PLBART** (Ahmad et al., 2021) is adapted from the BART (Lewis et al., 2019) architecture and pre-trained using denoising objective on Java, Python and stackoverflow corpus. **CodeT5** (Wang et al., 2021) is based on the T5 (Raffel et al., 2020) architecture, considering the identifier token information and applying multi-task learning. **UniXcoder** (Guo et al., 2022) is adapted from the UniLM (Dong et al., 2019) architecture, pretrained by different tasks (understanding and generation) on unified cross-modal data (code, AST and text). We also compare SCodeR with those code pre-trained models that utilize contrastive pre-training. **SynCoBERT** (Wang et al., 2022b) and **Code-MVP** (Wang et al., 2022a) construct positive pairs through multiple views of code like AST and CFG. **Corder** (Bui et al., 2021) and **DISCO** (Ding et al., 2022) construct positive code pairs from semantic-preserving transformations, and the latter additionally uses bug-injected

| Dataset | CSN | | | | | | | AdvTest | CosQA |
|---|---|---|---|---|---|---|---|---|---|
| Lang | Ruby | Javascript | Go | Python | Java | PHP | Average | Python | Python |
| CodeBERT | 67.9 | 62.0 | 88.2 | 67.2 | 67.6 | 62.8 | 69.3 | 27.2 | 64.7 |
| GraphCodeBERT | 70.3 | 64.4 | 89.7 | 69.2 | 69.1 | 64.9 | 71.3 | 35.2 | 67.5 |
| SyncoBERT | 72.2 | 67.7 | 91.3 | 72.4 | 72.3 | 67.8 | 74.0 | 38.1 | - |
| CodeRetriever | 75.3 | 69.5 | 91.6 | 73.3 | 74.0 | 68.2 | 75.3 | 43.0 | 69.6 |
| Code-MVP | - | - | - | - | - | - | - | 40.4 | 72.1 |
| UniXcoder | 74.0 | 68.4 | 91.5 | 72.0 | 72.6 | 67.6 | 74.4 | 41.3 | 70.1 |
| SCodeR | **77.5** | **72.0** | **92.7** | **74.2** | **74.8** | **69.2** | **76.7** | **45.5** | **74.5** |

Table 1: The comparison on code search task. The results of compared models are from their original papers.

codes as hard negatives. **CodeRetriever** (Li et al., 2022) builds code-code pairs by corresponding documents and function name automatically. For fair comparison, we use the same model architecture, pre-training corpus, and downstream hyperparameters as previous works (Li et al., 2022; Guo et al., 2022). To accelerate the training process, we initialize dual-encoder and discriminators with the released parameters of UniXcoder (Guo et al., 2022). More details about pre-training and fine-tuning can be found in the Appendix A and B.

| | POJ-104 | BigCloneBench | | |
|---|---|---|---|---|
| | MAP@R | Recall | Precision | F1-score |
| RoBERTa | 76.67 | 95.1 | 87.8 | 91.3 |
| CodeBERT | 82.67 | 94.7 | 93.4 | 94.1 |
| GraphCodeBERT | 85.16 | 94.8 | 95.2 | 95.0 |
| SyncoBERT | 88.24 | - | - | - |
| CodeRetriever | 88.85 | - | - | - |
| Corder | 84.10 | - | - | - |
| DISCO | 82.77 | 94.6 | 94.2 | 94.4 |
| PLBART | 86.27 | 94.8 | 92.5 | 93.6 |
| CodeT5-base | 88.65 | 94.8 | 94.7 | 95.0 |
| UniXcoder | 90.52 | 92.9 | **97.6** | 95.2 |
| SCodeR | **92.45** | **96.2** | 94.5 | **95.3** |

Table 2: Performance on code clone detection. The results of compared models are from their original papers.

### 5.2 Natural Language Code Search

Given a natural language query as the input, code search aims to retrieve the most semantically relevant code from a collection of code candidates. We conduct experiments on CSN (Guo et al., 2020), AdvTest (Lu et al., 2021) and CosQA (Huang et al., 2021) to evaluate SCodeR. CSN contains six programming languages, including Ruby, Javascript, Python, Java, PHP and Go. The dataset is constructed from CodeSearchNet Dataset (Husain et al., 2019) and noisy queries with low quality are filtered. AdvTest normalizes the function name and variable name of python code and thus is more challenging. The queries of CosQA are from Microsoft Bing search engine, which makes it closer to real-world code search scenario. Following previous works (Feng et al., 2020; Guo et al., 2020, 2022), we adopt Mean Reciprocal Rank (MRR) (Hull, 1999) as the evaluation metric.

The results are shown in Table 1. We can see that SCodeR outperforms previous code pre-trained models and achieves the new state-of-the-art performance on all datasets. Specifically, SCodeR outperforms UniXcoder by 2.3 points on the CSN dataset, and improves over state-of-the-art models about 2.5 points on AdvTest and CosQA datasets, which demonstrates the effectiveness of SCodeR.

### 5.3 Code Clone Detection

Code clone detection aims to identify the semantic similarity between two codes. We consider POJ-104 (Mou et al., 2016) and BigCloneBench (Svajlenko et al., 2014a) to evaluate SCodeR. POJ-104 dataset (C/C++) consists of codes from online judge (OJ) system. It aims to find the semantically similar codes given a code as query and evaluates by Mean Average Precision (MAP). BigCloneBench dataset (Java) is to judge whether two codes are similar and evaluates by Precision, Recall, and F1-score. We show the results in Table 2.

Compared with previous pre-trained models, SCodeR achieves the overall best performance on both datasets. On POJ-104 dataset, SCodeR surpasses all other methods. Specifically, SCodeR outperforms UniXcoder by 1.93 points. Although the pre-training corpus does not cover C/C++ programming languages, the superior performance reflects that SCodeR learns better general code knowledge. On the BigCloneBench dataset, SCodeR also achieves comparable performance. These results show that SCodeR learns better function-level code representation for code clone detection.

| Query PL | Ruby | | | Python | | | Java | | | Overall |
|---|---|---|---|---|---|---|---|---|---|---|
| Target PL | Ruby | Python | Java | Ruby | Python | Java | Ruby | Python | Java | |
| CodeBERT | 13.55 | 3.18 | 0.71 | 3.12 | 14.39 | 0.96 | 0.55 | 0.42 | 7.62 | 4.94 |
| GraphCodeBERT | 17.01 | 9.29 | 6.38 | 5.01 | 19.34 | 6.92 | 1.77 | 3.50 | 13.31 | 9.17 |
| PLBART | 18.60 | 10.76 | 1.90 | 8.27 | 19.55 | 1.98 | 1.47 | 1.27 | 10.41 | 8.25 |
| CodeT5-base | 18.22 | 10.02 | 1.81 | 8.74 | 17.83 | 1.58 | 1.13 | 0.81 | 10.18 | 7.81 |
| UniXcoder | 29.05 | 26.36 | 15.16 | 23.96 | 30.15 | 15.07 | 13.61 | 14.53 | 16.12 | 20.45 |
| SCodeR | **33.87** | **30.25** | **17.10** | **26.48** | **33.02** | **16.95** | **16.5** | **19.06** | **18.87** | **23.57** |

Table 3: The comparison on zero-shot code-to-code search. Baselines' results are reported by Guo et al. (2022).

| Model | Kendall's Tau |
|---|---|
| CodeBERT (Feng et al., 2020) | 81.9 |
| GraphCodeBERT (Guo et al., 2020) | 84.7 |
| PLBART (Ahmad et al., 2021) | 84.7 |
| CodeT5 (Wang et al., 2021) | 84.7 |
| UniXcoder (Guo et al., 2022) | 85.9 |
| SCodeR | **86.6** |

Table 4: Experiment results on markdown ordering in python notebooks.

### 5.4 Zero-Shot Code-to-Code Search

We also evaluate SCodeR in zero-shot code-to-code search. Given a code snippet as query, the task aims to find semantically similar codes from a collection of code candidates in zero-shot setting. Since the annotation of code-to-code search is labor-intensive and costly (Svajlenko et al., 2014b; Li et al., 2022), the zero-shot performance can indicate the model's utility in real-world scenario, where a lot of programming languages do not have an annotated dataset for code-to-code search. We follow Guo et al. (2022) to conduct the experiment on CodeNet (Puri et al., 2021) and evaluate models using MAP score. The results are listed in Table 3. The first and the second row correspond to query and target programming languages.

We can see that SCodeR outperforms all other compared models and improves over the state-of-the-art model, i.e. UniXcoder, by 3.12 average absolute points. Meanwhile, SCodeR has a consistent improvement on the cross-PL setting, which can help users to translate programs from one PL to another via retrieving semantically relevant codes.

### 5.5 Markdown Ordering in Python Notebooks

This task is to reconstruct the order of markdown cells in a given notebook according to the ordered code cells. We conduct experiments on the dataset provided by Kaggle[1] and use the official evaluation metric, *Kendall's tau ($\tau$)*. It is computed as $1 - 2 * N/\binom{n}{r}$ where $N$ is the number of pairs in the predicted sequence with incorrect relative order and $n$ is the sequence length.

We take the normalized markdown cell's position in a given notebook as labels for each markdown cell (0$\sim$1), and solve this task as a regression task. To test performance of function-level code representation, we use pre-trained models to encode each cell to function-level representation as features. We use a randomly initialized Transformer that takes extracted features of cells in the python notebook to predict position of each cell. Note that parameters of pre-trained models are fixed in the fine-tuning procedure, and thus the performance of this task depends on function-level feature extracted from pre-trained models.

We show the results in Table 4. SCodeR outperforms other pre-trained models and achieves 0.5 points higher than UniXcoder. This indicates that SCodeR learns better representation for both code and natural language comments, and can help better understand the fine-grained relationship of codes and comments in the python notebook.

### 5.6 Analysis

**Ablation Study** To evaluate the effect of our positive sample construction methods and soft-labeled contrastive pre-training framework, we conduct ablation study on the CSN dataset and take the pre-trained model with no enhancement as the baseline (i.e. UniXcoder). At first, we individually compare the proposed ASST with the transformation-based positive sample construction method (Jain et al., 2020; Bui et al., 2021). Notice that previous works do not apply their transformation-based methods on all six programming languages covered by our pre-training corpus. For fair comparison and keeping the pre-training corpus consistent, we follow Lu et al. (2022) to implement the widely used transformations including variable renaming and dead code insertion on six programming languages by ourselves. Then, we add the remaining modules of SCodeR to evaluate their performance. The results

[1] https://www.kaggle.com/competitions/AI4Code/overview

| Methods | Ruby | Javascript | Go | Python | Java | Php | Overall |
|---|---|---|---|---|---|---|---|
| Baseline | 74.0 | 68.4 | 91.5 | 72.0 | 72.6 | 67.6 | 74.4 |
| Baseline + Code Transformation | 74.5 | 68.7 | 91.9 | 72.2 | 72.6 | 67.7 | 74.6 |
| Baseline + ASST | 76.1 | 70.1 | 92.1 | 73.0 | 73.3 | 68.1 | 75.4 |
| Baseline + ASST + Code Comment | 76.2 | 71.2 | 92.2 | 73.4 | 73.7 | 68.5 | 75.9 |
| Baseline + ASST + Code Comment + Soft-Labled | **77.5** | **72.0** | **92.7** | **74.2** | **74.9** | **69.2** | **76.8** |

Table 5: Ablation study on natural language code search.

are shown in Table 5.

Compared with transformation-based methods, we can see that our positive sample construction (ASST) achieves better performance. Meanwhile, positive pairs from ASST can bring significant improvement over the baseline, which reflects its effectiveness. After using the text-code pairs, the performance improves over 0.5 points, which shows that code comments provide rich semantic information to help model learn better code representation. When adding soft-labeled contrastive pre-training, the model performance increases by 0.9 points, which demonstrates that applying relevance among samples as soft-labels for contrastive learning can further improve code representation.

| | Ruby | Python | Java | Overall |
|---|---|---|---|---|
| SCodeR | **33.87** | **33.02** | **18.87** | **28.6** |
| SCodeR$_T$ | 32.97 | 30.78 | 18.01 | 27.2 |
| SCodeR$_L$ | 33.22 | 31.31 | 18.23 | 27.6 |

Table 6: Experiment results of different strategies of code splitting on zero-shot code-to-code search. SCodeR$_T$ and SCodeR$_L$ use token-level and line-level ICT to replace ASST.

**Effect of AST-based Splitting** We conduct experiments on zero-shot code-to-code search to analyze the effect of AST-based splitting strategy of ASST by comparing ASST with two variants of splitting strategy. The first strategy is token-level ICT that takes a random span of code tokens and their context as positive pairs. The second strategy is line-level ICT that considers random consecutive code lines and the remaining lines as positive pairs. Compared with our AST-based splitting method, these two splitting strategies will cause ungrammatical codes and mislead the model to focus on structural matching rather than semantic matching. The results are shown in Table 6 and we can see that the two variants of splitting strategy lead to worse performance, which shows the effectiveness of our AST-based splitting method.

**Comment :** `Compute Mean Value`

| Code id | Code Content | Soft Label |
|---|---|---|
| Code$^+$ | `def mean(x):`<br>`  result = sum(x)/len(x)`<br>`  return result` | 0.5284 |
| Code$_1^-$ | `def sort(inp):`<br>`  inp.sort(reversed=True)` | 0.0003 |
| Code$_2^-$ | `def average(inp):`<br>`  total = 0`<br>`  for i in inp:`<br>`  total += i`<br>`  return total/len(inp)` | 0.4713 |

Figure 3: Case study on the discriminator. The soft-label is the relevance scores between the comment and codes from the discriminator, $p_\phi(\cdot|x, \mathbb{X})$.

**Case Study** We give a case study in Figure 3 to show the importance of soft-labels for contrastive pre-training. The figure includes one paired code and two other codes with soft-labels provided by the discriminators. We can see that the soft-label of negative Code$_1^-$ is close to 0 since the code is unrelated with the comment and the discriminators can predict correct relevance score between them for contrastive pre-training. Code$_2^-$ is a false negative that has the same functionality as Code$^+$ and should be assigned similar weights when we apply contrastive pre-training. As we can see in the figure, the discriminator can understand code semantics and provide similar soft-labels (i.e 0.5284 vs 0.4713) about Code$^+$ and Code$_2^-$ for contrastive pre-training, which can alleviate the influence of false negative issue and learn better code representation through soft-labels.

## 6 Conclusion

In this paper, we present SCodeR to learn function-level code representation with soft-labeled contrastive pre-training. To alleviate the "false negative" issue in code corpus, we propose a soft-labeled contrastive pre-training framework that takes relevance scores among samples as soft-labels for contrastive pre-training in an iterative

adversarial manner. Besides, we propose to utilize code comment and abstract syntax sub-tree of the source code to build positive samples that can facilitate the model to capture semantic information from the source code. Experimental results show that SCodeR achieves state-of-the-art performance on four code-related tasks over seven datasets. Further ablation studies show the effectiveness of our soft-labeled contrastive pre-training framework and positive sample construction methods.

## Limitations

There are two limitations of this work: 1) In the adversarial iteration, we introduce discriminators to provide soft-labels for the training of dual-encoder, which increases GPU memory occupation. To solve it, we can obtain these soft-labels offline, which may complicate the pipeline of data processing. 2) We only use UniXcoder as the backbone model in the experiments due to the computation resources limitation. We leave pre-training based on other code pre-trained models like CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2020) and Codex (Chen et al., 2021a) as future work.

## 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.

Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2019, Athens, Greece, October 23-24, 2019*, pages 143–153. ACM.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 511–521.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021a. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Tsai-Shien Chen, Wei-Chih Hung, Hung-Yu Tseng, Shao-Yi Chien, and Ming-Hsuan Yang. 2021b. Incremental false negative detection for contrastive learning. *CoRR*, abs/2106.03719.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Yangruibo Ding, Luca Buratti, Saurabh Pujar, Alessandro Morari, Baishakhi Ray, and Saikat Chakraborty. 2022. Towards learning (dis)-similarity of source code from program contrasts. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6300–6312.

Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. 2019. Unified language model pre-training for natural language understanding and generation. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 13042–13054.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

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, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcode-bert: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. 2015. Distilling the knowledge in a neural network. *CoRR*, abs/1503.02531.

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. *arXiv preprint arXiv:2105.13239*.

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. *arXiv preprint arXiv:1909.09436*.

Tri Huynh, Simon Kornblith, Matthew R. Walter, Michael Maire, and Maryam Khademi. 2022. Boosting contrastive self-supervised learning with false negative cancellation. In *IEEE/CVF Winter Conference on Applications of Computer Vision, WACV 2022, Waikoloa, HI, USA, January 3-8, 2022*, pages 986–996. IEEE.

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. 2021. Contrastive code representation learning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 5954–5971.

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

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2019. Pre-trained contextual embedding of source code. *arXiv preprint arXiv:2001.00059*.

Kenton Lee, Ming-Wei Chang, and Kristina Toutanova. 2019. Latent retrieval for weakly supervised open domain question answering. In *ACL*, pages 6086–6096. Association for Computational Linguistics.

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.

Xiaonan Li, Yeyun Gong, Yelong Shen, Xipeng Qiu, Hang Zhang, Bolun Yao, Weizhen Qi, Daxin Jiang, Weizhu Chen, and Nan Duan. 2022. Coderetriever: Unimodal and bimodal contrastive learning. *CoRR*, abs/2201.10866.

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. *arXiv preprint arXiv:1907.11692*.

Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. Déjàvu: a map of code duplicates on github. *Proc. ACM Program. Lang.*, 1(OOPSLA):84:1–84:28.

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seungwon Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 6227–6240. Association for Computational Linguistics.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv: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.

Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, and Ulrich Finkler. 2021. Project codenet: A large-scale AI for code dataset for learning a diversity of coding tasks. *CoRR*, abs/2105.12655.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou,

Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*.

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.

Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal Kumar Roy, and Mohammad Mamun Mia. 2014a. Towards a big data curated benchmark of inter-project code clones. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 476–480. IEEE Computer Society.

Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal Kumar Roy, and Mohammad Mamun Mia. 2014b. Towards a big data curated benchmark of inter-project code clones. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 476–480. IEEE Computer Society.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. *arXiv preprint arXiv:2005.08025*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

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

Xin Wang, Fei Mi Yasheng Wang, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2022b. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Hang Zhang, Yeyun Gong, Yelong Shen, Jiancheng Lv, Nan Duan, and Weizhu Chen. 2021. Adversarial retriever-ranker for dense text retrieval. *CoRR*, abs/2110.03611.

## A Pre-training Settings

For fair comparison, we adopt the same model architecture and the same pre-training corpus as previous works (Feng et al., 2020; Guo et al., 2020). The used corpus is CodeSerachNet (Husain et al., 2019), which includes 2.3M functions paired with documents in six programming languages. We leverage tree-sitter[2] to get the AST information for ASST. The node set $N$ for ASST includes: "for_statement", "while_statement", "if_statement", "with_statement", "try_statement", "assignment_statement", etc. We also consider the "function_call" node if it is not under an indivisible node like "assignment_statement". The dual-encoder consists of 12 layers transformer with 768 hidden dimensional hidden states and 12 attention heads. The architecture of discriminators is the same as dual-encoder. To accelerate the training process, we adopt the released parameters of UniXcoder (Guo et al., 2022) to initialize the dual-encoder and discriminators. ScodeR is trained on 8 Nvidia Tesla A100 with 40GB memory and costs about 37 hours. We show the pre-training hyper-parameters in Table 7.

| Hyper-Parameters | Dual-encoder | Discriminators |
|---|---|---|
| Initialization | UniXcoder | UniXcoder |
| Optimizer | AdamW | AdamW |
| Scheduler | Linear | Linear |
| Warmup proportion | 0.1 | 0.1 |
| Negative size | 7 | 7 |
| Batch size | 64 | 64 |
| Learning rate | 5e-6 | 1e-5 |
| Max step | 24000 | 16000 |
| Iterations | 4 | 4 |
| Loss Weight $\lambda$ | 0.2 | - |

Table 7: The hyper-parameters of pre-training.

## B Fine-tuning Settings

### B.1 Natural Language Code Search

Given a natural language query as the input, code search aims to retrieve the most semantically relevant code from a collection of code candidates. We conduct experiments on CSN (Guo et al., 2020), AdvTest (Lu et al., 2021) and CosQA (Huang et al., 2021) to evaluate SCodeR.

On CSN, we follow Li et al. (2022) to set the batch size as 128, learning rate as 2e-5, and max sequence length of PL and NL as 256 and 128. We finetune the model for 10 epochs using AdamW optimzier and select the best checkpoint based on the development set.

On AdvTest dataset, we finetune the model for 2 epochs and keep other hyper-parameters same as CSN dataset.

On CosQA dataset, we use the same hyper-parameters as CSN dataset.

### B.2 Code Clone Detection

Code clone detection aims to identify the semantic similarity between two codes. We consider POJ-104 (Mou et al., 2016) and BigCloneBench (Svajlenko et al., 2014a) to evaluate SCodeR.

On POJ-104 dataset, we follow Guo et al. (2022) to set the batch size as 8, the learning rate as 2e-5, and the max sequence length as 400. We finetune the model using AdamW optimzier for 2 epochs.

On BigCloneBench dataset, we follow Guo et al. (2022) to set the batch size as 16, learning rate as 5er-5 and the max sequence length as 512. We use AdamW optimizer to fine-tune the model and select the best checkpoint based on the development set.

### B.3 Markdown Ordering in Python Notebooks

This task is to reconstruct the order of markdown cells in a given notebook according to the ordered code cells. We conduct experiments on the dataset provided by Kaggle. we use pre-trained models to encode each cell to function-level representation as features and set the max sequence length of each cell as 128. We use a randomly initialized Transformer that takes extracted features of cells in the python notebook to predict position of each cell. We set this Transformer's layers, hidden size, and attention heads as 6, 768, and 12, respectively. For training it, we set the batch size as 128, learning rate as 2e-5, the max number of cells as 256, and the optimizer as AdamW.

[2]https://github.com/tree-sitter/tree-sitter