MCodeSearcher: Multi-View Contrastive Learning for Code Search

Jia Li
Peking University
Beijing, China
lijiaa@pku.edu.cn

Fang Liu
Beihang University
Beijing, China
fangliu@buaa.edu.cn

Jia Li ♂
Peking University
Beijing, China
lijia@stu.pku.edu.cn

Yunfei Zhao
Peking University
Beijing, China
zhaoyunfei@pku.edu.cn

Ge Li[*]
Peking University
Beijing, China
lige@pku.edu.cn

Zhi Jin[*]
Peking University
Beijing, China
zhijin@pku.edu.cn

ABSTRACT

Code search has been a critical software development activity in facilitating developers to retrieve a proper code snippet from open-source repositories given a user intent. In recent years, large-scale pre-trained models have shown impressive performance on code representation learning and have achieved state-of-the-art performance on code search task. However, it is challenging for these models to distinguish the functionally equivalent code snippets with dissimilar implementations or the non-equivalent code snippets that look similar. Due to the diversity of the code implementations, it is necessary for the code search engines to identify the functional similarities or dissimilarities of source code so as to return the functionally matched source code for a given query. Besides, existing pre-trained models mainly focus on learning the semantic representations of code snippets. The semantic correlation between the code snippet and natural language query is not sufficiently exploited. An effective code search tool not only needs to understand the relationship between queries and code snippets but also needs to identify the relationship between diversified code snippets. To address these limitations, we propose a novel multi-view contrastive learning model MCodeSearcher for code retrieval, aiming at sufficiently exploiting (1) the semantic correlation between queries and code snippets, and (2) the relationship between functionally equivalent code snippets. To achieve this, we design contrastive training objectives from three views and pre-train our model with these objectives. The experimental results on five representative code search datasets show that our approach significantly outperforms the state-of-the-art methods.

CCS CONCEPTS

• **Software and its engineering**; • **Computing methodologies** → Artificial intelligence;

KEYWORDS

Code search, contrastive learning, deep neural network

ACM Reference Format:
Jia Li, Fang Liu, Jia Li ♂, Yunfei Zhao, Ge Li, and Zhi Jin. 2023. MCodeSearcher: Multi-View Contrastive Learning for Code Search. In *14th Asia-Pacific Symposium on Internetware (Internetware 2023), August 04–06, 2023, Hangzhou, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3609437.3609456

1 INTRODUCTION

Code search has become an essential software development activity for programmers in software engineering. Developers usually spend 19% of their working time searching existing code based on their intents for code reuse [37]. To facilitate software development efficiency, many researchers proposed automatic code search methods. The goal of automatic code search is to retrieve a proper code snippet from code repositories given a user intent (e.g., a natural language description of the code snippet). With the emergence of large amounts of available code repositories (e.g., GitHub [1] and StackOverflow [2]), how to retrieve semantically equivalent code from abundant candidate codes becomes a challenging problem.

Early code search studies mainly pay attention to the lexical information of the code snippets and utilize information retrieval techniques to find the relevant code snippets given a query[23, 27, 29]. Specifically, they apply a sparse vector retriever (e.g., TF-IDF and BM25 [32]) to compute the lexical similarity between the source code and natural language query. However, these approaches are sensitive to the name of code entities (e.g., identifiers and method names) and lack the ability to understand the high-level semantic features of source codes and queries. With the development of deep learning (DL), researchers have begun to utilize deep neural networks for automated code search. Generally, DL-based approaches [3, 7, 10, 11, 21] encode code snippets and natural language queries into representation vectors, then utilize vector distances to approximate semantic correlation between them. Among these DL-based models, the pre-trained models [12, 13, 30, 42] have shown impressive performance in code understanding and achieved state-of-the-art performance on the code search task. Although these

[*] Corresponding author.

Internetware 2023, August 04–06, 2023, Hangzhou, China

ACM ISBN 979-8-4007-0894-7/23/08…$15.00
https://doi.org/10.1145/3609437.3609456

[1] https://github.com/
[2] https://stackoverflow.com/

pre-trained models have acquired initial success in code understanding, these models are not optimal for the code search task for the following reasons.

Firstly, it is challenging for these models to distinguish between the functionally equivalent code snippets with different implementations, or to separate the non-equivalent code snippets with similar implementations. Given a query, a code search engine is expected to retrieve functionally equivalent code snippets. Due to the diversity of code implementations, that is, a certain functional requirement can be implemented in different ways, it is necessary for the code search engines to identify the functional similarities of source code so as to return the functionally matched source code. Secondly, these pre-trained models mainly focus on learning the semantic representations of code snippets. The semantic correlation between the code snippet and the natural language query is not fully exploited during the pre-training process. An effective code search tool should be able to understand the semantics of both code snippets and natural language queries, and further have the ability to distinguish them with different semantics.

To address these limitations, we propose a novel multi-view contrastive learning method for code search as shown in Figure 1. The key idea of multi-view contrastive learning is pulling semantic neighbors together and pushing semantic non-neighbors apart, which aims to sufficiently grasp the similar or dissimilar relationships between the queries and source codes. To achieve this goal, our model jointly learns the representations of queries and source codes by exploiting the following relationships:

- **Semantic relationship between the query and code snippet**: learning the semantic relevance between queries and code snippets is the key task in code search models. To achieve this, we construct the following views of contrastive learning to train our model. (1) *Query-Anchor-View* : the first view aims to identify the semantically equivalent query and code snippet among many non-equivalent snippets; (2) *Query-Variant-View* : to prompt identifying the relationships between queries and code snippets, except for the original code, the model is also required to retrieve another functionally equivalent code snippet. The main idea of this objective is to guarantee that code snippets with the same function should have the same similarity or dissimilarity to a certain query. We propose a back-translation transformation to automatically generate the functionally equivalent code snippet for the original code snippet.
- **Semantic relevance between code snippets**: considering the diversity of the code implementations, it is necessary to identify the functional similarities or dissimilarities of code snippets for retrieving the functionally equivalent code. To achieve this goal, the following view is proposed. (3) *Anchor-Variant-View* : we optimize our model by maximizing the similarity between the vector representation of the original code and its functionally equivalent code (obtained by back-translation transformation), meanwhile minimizing the representational similarity between the original code and other negative snippets.

Specifically, we unite the above three aspects and design the multi-view contrastive learning model, MCodeSearcher, to learn these relationships adequately and further identify functionally similar code snippets among many non-equivalent snippets given a natural language query. Compared with the state-of-the-art code search approach CodeRetriever [20], we train MCodeSearcher on a smaller dataset that only contains the bimodal data (e.g., <query, code snippet> pairs) of CodeSearchNet dataset [15]. To evaluate the effectiveness of MCodeSearcher, we conduct extensive experiments on five representative datasets [14, 15, 28, 43, 44]. Experimental results demonstrate that MCodeSearcher significantly outperforms baselines. Specifically, on CodeSearchNet dataset [15], MCodeSearcher brings 10.21% and 4.01% relative improvements at Python and Java languages on MRR; on CoSQA [14], and StaQC [43] datasets, our model achieves 2.43% and 7.15% relative improvements on MRR, respectively. To further verify MCodeSearcher, we further introduce the question answering scenario, and on WebQuery dataset [28], MCodeSearcher acquires 3.35% and 2.39% relative improvements on F1 and Accuracy score.

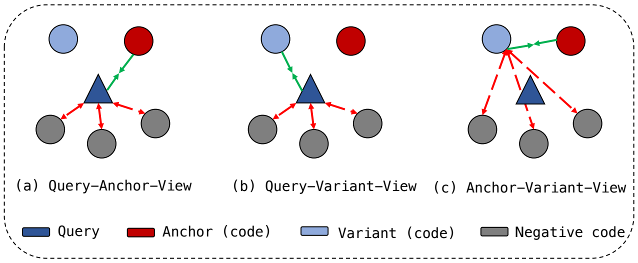


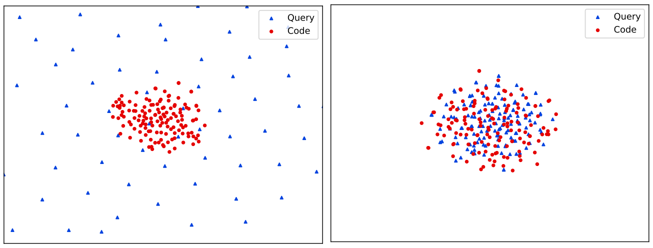
Figure 1: The illustration of our multi-view contrastive learning method. (a), (b) and (c) represent Query-Anchor-View, Query-Variant-View, and Anchor-Variant-View, respectively.

Our contributions are summarized as follows:

- We propose a novel multi-view contrastive learning model, MCodeSearcher, for code search task.
- We design contrastive learning objectives from three views, e.g., Query-Anchor-View, Query-Variant-View, and Anchor-Variant-View, to promote MCodeSearcher identifying the semantic similarities or dissimilarities between natural language queries and code snippets.
- We conduct extensive experiments to evaluate MCodeSeacher on five representative code search datasets. Results show that our model achieves the the-state-of-art performance.

2 MOTIVATING EXAMPLES

Recently, many researchers have devoted to learning the unified representations in a vector space between source code and natural language through co-training them. For the code search task, the representations of semantically equivalent code snippets and queries should ideally have a similar distribution in the high-dimensional vector space. However, we observe that some popular pre-training models, such as GraphCodeBERT [13], have a suboptimal representation for code retrieval. Figure 2 shows a visualization of the representations of queries and code snippets in GraphCodeBERT and MCodeSearcher without fine-tuning by t-SNE [38]. From Figure 2(a), we can find that the representation distribution of code snippets is more centralized, while the representation distribution of natural language queries is scattered. The representation distributions of the code and query are apart from each other. We propose MCodeSearcher: a multi-view contrastive learning method that pulls the functionally equivalent queries and code snippets together in the vector space. Figure 2(b) shows the representation distributions of the code and query obtained from our MCodeSearcher. The representation distributions of code snippets and queries are well entangled, which can be explained by the fact that our multi-view contrastive learning can effectively minimize the representation distance between semantically equivalent queries and code snippets.



(a) GraphCodeBERT (b) MCodeSearcher

Figure 2: t-SNE visualization of representations of queries and code snippets in GraphCodeBERT and MCodeSearcher without fine-tuning. Blue color dot denotes queries and red color dot means code snippets. The representations of code snippets and queries in GraphCodeBERT are scattered. The features in MCoderSearcher are well entangled, suggesting that our model can better support code search.

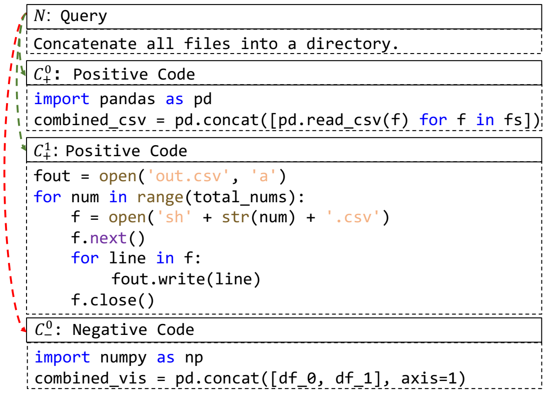
Further, we look closely at the specific examples from the code search dataset. Figure 3 provides some representative samples from the dataset. These samples reveal two common phenomenons: (1) The diversity of code implementations, that is, a certain functional requirement can be implemented in different methods. As shown in Figure 3, code snippets C_+^0 and C_+^1 both realize the function of concatenating files into a directory. Though they have the same semantics, their implementation ways are disparate, and their syntax differs. (2) Code snippets with similar lexicon and syntax may have different semantics. Code snippets C_+^0 and C_-^0 are syntactically semblable yet meet different queries, which easily misleads neural networks to treat them as functionally equivalent code snippets. Therefore, it is challenging for code search engines to retrieve correct code snippets that match the demand of a query from a large number of candidate code snippets (e.g., a codebase).

3 RELATED WORK

3.1 Code Search

The main idea of code search is to select the most semantically related code snippet from a codebase given a query. Many models have been designed for matching the queries and code snippets, which mainly fall into two categories in terms of matching formats: sparse retriever and dense retriever.

The sparse retriever focuses on the lexical matching between natural language queries and code snippets [23, 27, 29]. Linstead et al. [23] propose an information retrieval model based on a code



N: Query

```
Concatenate all files into a directory.
```

C_+^0: Positive Code

```
import pandas as pd
combined_csv = pd.concat([pd.read_csv(f) for f in fs])
```

C_+^1: Positive Code

```
fout = open('out.csv', 'a')
for num in range(total_nums):
    f = open('sh' + str(num) + '.csv')
    f.next()
    for line in f:
        fout.write(line)
    f.close()
```

C_-^0: Negative Code

```
import numpy as np
combined_vis = pd.concat([df_0, df_1], axis=1)
```

Figure 3: A motivating example to better illustrate our motivation from the dataset.

search tool Sourcerer. It combines the software's textual content with structural information. Lv et al. [29] design a sparse phrase-based method, which extracts natural language phrases from code identifiers and matches queries with these phrases. However, these approaches are susceptible to the name of code entities and cannot understand the semantic features of the queries and code snippets.

To achieve semantic matching, the dense retriever [16, 22, 31, 45, 46] is introduced for code search, which encodes code snippets and queries into dense vectors and measures their matching degree based on their high-dimensional vectors. Gu et al. [11] and Wan et al. [40] design multi-modal attention neural networks to effectively integrate structural information of code snippets and acquire syntax-augmented representations for code retrieval. Recently, pre-trained language models [1, 9, 42] have made great progress on code search. Feng et al. [9] present CodeBERT and train it with the replaced token detection task (RTD) and the masked language modeling task (MLM). Later, GraphCodeBERT [13] and SPT-Code [30] are proposed to explore the syntactic structure of code snippets for better code understanding and representation. These pre-trained models pay more attention to exploring the source code features with the token-level objectives, such as MLM [9, 33] and RTD [9]. They ignore modeling the semantic relationship between code snippets and queries. To mitigate these issues, this paper proposes a multi-view contrastive learning model that focuses on grasping the semantic similarities or dissimilarities between queries and code snippets for the code search task.

3.2 Self-supervised Contrastive Learning

Contrastive learning [4] is a proven effective method for program language processing [2, 6, 17], which minimizes the distance of representations between similar examples while maximizing the distance between dissimilar instances. Since effective contrastive data pairs are usually scarce and require much human effort to collect, self-supervised contrastive learning is proposed to augment a given sample by constructing a similar counterpart without human interference, then force the model to recognize the similar counterpart from several randomly selected samples. Bui et al. [2] firstly design a contrastive learning framework for programming language. It utilizes programming transformation techniques, such as variable renaming and permutation of statements, to generate semantically equivalent code snippets and then train neural networks to identify semantically equivalent code snippets from a large set of transformed code. Ding et al. [6] propose source-to-source code augmentation methods to generate positive code snippets, including misusing variables, changing function calls, and function renaming. Then a pre-trained neural network is optimized to minimize the distance between the original code snippets and their generated code snippets with similar semantics.

However, existing code transformation methods for contrastive learning [2, 6, 17] are mainly based on rule-based operations. These limited augmentation operations are easy for deep neural networks to learn during their early training process, leading to the sub-optimal representations for code snippets [8]. To prevent the above limitation, we propose an automatic translation-based source-to-source transformation to generate a series of functionally equivalent yet syntactically different snippets.

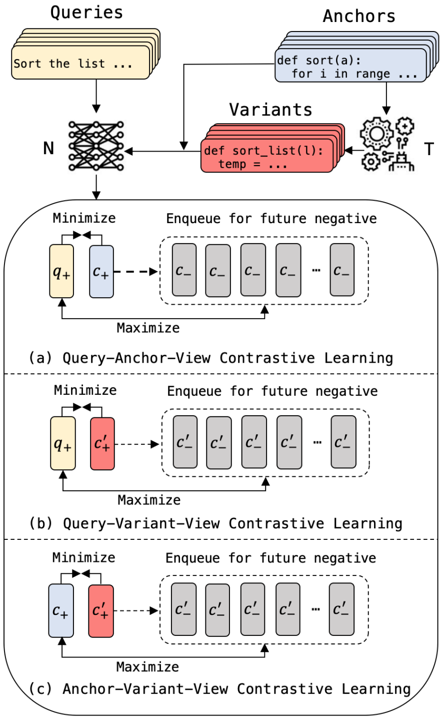
3.3 Code Translation

Source-to-source translation devotes to converting source code from a programming language to another language, such as Java to Python. Existing translation studies [19, 24, 35] are mainly based on deep learning methods, among which the unsupervised learning methods achieve impressive performances. Lachaux et al. [19] propose TransCoder that does not require any parallel data and can easily generalize high-quality code in other languages. To further improve the quality of translated code, Roziere et al. [34] introduce TransCoder-ST, which utilizes an automated test generation unit to filter out invalid translations and reduce the noise from the back-translation procedure, which achieves state-of-the-art performance on program translation. Back-translation [35] is an effective data-augmentation approach where the model re-translates programs from the target language back to its source language. In this work, we employ back-translation transformation to generate code snippets that are functionally equivalent but different in forms for contrastive learning based on TransCoder-ST.

4 METHOD

4.1 An overview

Figure 4 presents an overview of our proposed MCodeSearcher. Overall, MCodeSearcher contains three submodules:
(i) **Source code transformation module**. This module transforms a given code snippet c_i into a new code snippet $\tilde{c_i}$ which is functionally equivalent but in a different implementation with c_i. In this work, we name c_i as *anchor* and $\tilde{c_i}$ as a *variant*.
(ii) **Neural network encoder for the query and source code**. The module maps a query q_i into a vector representation Q_i, and converts a code snippet c_i into C_i. In our case, it maps c_i and $\tilde{c_i}$ into two different code vectors (C_i and $\tilde{C_i}$), respectively.
(iii) **Multi-view contrastive learning**. The module learns the semantic correlation between queries and code snippets through Query-Anchor-View (QAV), Query-Variant-View (QVV), and Anchor-Variant-View (AVV) contrastive learning. We will elaborate on each view of contrastive learning in Section 4.4.



(a) Query-Anchor-View Contrastive Learning

(b) Query-Variant-View Contrastive Learning

(c) Anchor-Variant-View Contrastive Learning

Figure 4: The architecture of our proposed MCodeSearcher. (a), (b), and (c) represent our multi-view contrastive learning method. "N" denotes the encoder for queries and code snippets. "T" means translation transformation.

4.2 Source Code Transformation

In software engineering, a functional requirement usually is implemented in different ways. Given a natural language query, a satisfying code search engine should retrieve all semantically equivalent code snippets that can be different in forms. To achieve this goal, we require a formally diverse but functionally equivalent program for each original code snippet. Manually collecting them is expensive and time-consuming. Thus, we leverage the back-translation transformation to automatically generate such diverse code snippets.

Automatic code translation has been an active search area in the software engineering community, which converts source code from a specific programming language (such as Python) to another (such as Java). With the increasing ability of large language models, many reliable tools [18, 19, 34] are present for code translation. In this paper, we apply TransCoder-ST [34], pre-trained on a large amount of parallel data, to augment programs, since it is trained with an automated unit test pipeline to filter out invalid translations.

4.3 Encoder for Query and Source Code

4.3.1 Encoder Architecture. We use the multi-layer bidirectional Transformer [39] as the encoder backbone to map the query q_i and code snippet c_i to vector representations Q_i and C_i. The encoder f_θ consists of a 12-layer Transformer with 768 hidden size and 12 attention heads in each layer. We initialize it with GraphCodeBERT [13] and further optimize it with our multi-view contrastive learning method. Once pre-trained, f_θ is applied to code search.

4.3.2 Input-Output Representations. The encoder takes the token sequence of the query and code snippet as input, respectively. Given a code snippet c_i, we utilize the tokenizer to split c_i to a sequence of tokens $\left\{c_{i,1}, c_{i,2}, ..., c_{i,n_c}\right\}$, where n_c denotes the sequence length. Then a classification symbol [CLS] and a segment separation symbol [SEP] are concatenated to the sequence, forming the input as $\{[\texttt{CLS}], c_{i,1}, c_{i,2}, ..., c_{i,n_c}, [\texttt{SEP}]\}$. We perform the same pre-processing procedure for the query q_i, and acquire the final query input as $\{[\texttt{CLS}], q_{i,1}, q_{i,2}, ..., q_{i,n_q}, [\texttt{SEP}]\}$.

The output of the encoder f_θ includes: (1) the vector representation of the code snippet $\{E_{c_{i,[\texttt{CLS}]}}, E_{c_{i,1}}, ..., E_{c_{i,n_c}}, E_{c_{i,[\texttt{SEP}]}}\}$; and (2) the vector representation of the query $\left\{E_{q_{i,[\texttt{CLS}]}}, E_{q_{i,1}}, ..., E_{q_{i,n_q}}, E_{q_{i,[\texttt{SEP}]}}\right\}$. Following previous works [9, 13], we utilize $E_{q_{i,[\texttt{CLS}]}}$ and $E_{c_{i,[\texttt{CLS}]}}$ as the entity representation of q_i and c_i since they are the aggregated representations. That is $Q_i = E_{q_{i,[\texttt{CLS}]}}$ and $C_i = E_{c_{i,[\texttt{CLS}]}}$.

4.4 Multi-View Contrastive Learning

To sufficiently explore the semantic correlation between queries and code snippets, the multi-view contrastive objectives contain three views, e.g., Query-Anchor-View, Query-Variant-View, and Anchor-Variant-View.

4.4.1 Query-Anchor-View Contrastive Learning. Learning the semantic relevance between code snippets and queries is the key to code search task. To achieve this goal, we design the Query-Anchor-View contrastive learning to maximize the similarity of semantically equivalent queries and code snippets, meanwhile, minimize the similarity of semantically non-equivalent examples. Specifically, in the mini-batch examples, N code snippets and their corresponding natural language queries are set as N positive examples. For the negative instances, we construct a queue to retain code snippets by enqueuing the samples in the current mini-batch and dequeuing the samples in the oldest mini-batch. In this work, the queue can hold $N \times K$ instances. Therefore, the mini-batch is extended to N positive examples $\mathbb{P}_{QAV}$, and $N \times (N \times K - 1)$ negative instances $\mathbb{N}_{QAV}$. The Query-Anchor-View contrastive loss $\mathcal{L}_{QAV}$ is formulated as:

$$\mathcal{L}_{QAV} = -\mathbb{E}_{Q_i \sim \mathbb{P}_{QAV}}\left[log \frac{e^{s(Q_i, C_i)/\tau}}{\sum_{C_j \in \mathbb{N}_{QAV}} e^{s(Q_i, C_j)/\tau}}\right] \tag{1}$$

where τ is a temperature parameter. $s(\cdot)$ calculates the cosine similarity of two vectors.

Based on the objective, MCodeSearcher incorporates the ability to identify functionally equivalent queries and code snippets among many non-equivalent snippets.

4.4.2 Query-Variant-View Contrastive Learning. To prompt identifying the semantic similarity between queries and code snippets, we further propose the Query-Variant-View contrastive learning. Given a specific query, this objective requires the model to retrieve another functionally equivalent code snippet (variant) except for the original code (anchor). The objective can guarantee that code snippets with the same function could have a consistent similarity or dissimilarity to a certain query. Precisely, given a mini-batch of N query-code pairs, we first utilize the TransCoder-ST tool to automatically generate a variant for each anchor in the mini-batch through the back-translation method. The variant has functionally equivalent semantics but is different in implementation compared with the anchor. Then we pair the query and the generated variant and acquire N positive instances $\mathbb{P}_{QVV}$. We use the same method as subsection 4.4.1 to construct $N \times (N \times K - 1)$ negative examples $\mathbb{N}_{QVV}$, where the queue holds the generated variants. The Query-Variant-View contrastive loss $\mathcal{L}_{QVV}$ is fedined as:

$$\mathcal{L}_{QVV} = -\mathbb{E}_{Q_i \sim \mathbb{P}_{QVV}}\left[log \frac{e^{s(Q_i, \tilde{C_i})/\tau}}{\sum_{\tilde{C_j} \in \mathbb{N}_{QVV}} e^{s(Q_i, \tilde{C_j})/\tau}}\right] \tag{2}$$

Through objective, our model learns to ensure that code snippets that look different but are functionally equivalent could have a similar semantic relationship with a specific query.

4.4.3 Anchor-Variant-View Contrastive Learning. As the diversity of the code implementations, it is necessary to identify the functional similarities or dissimilarities of code snippets to find the functionally equivalent source code. Therefore, we design the Anchor-Variant-View contrastive learning to pull the representations of functionally equivalent snippets together and push functionally non-equivalent snippets apart. Specifically, given a mini-batch of N code snippets as anchors. We match each anchor with its corresponding variant (obtained by back-translation transformation) as a positive instance, and acquire N positive examples $\mathbb{P}_{AVV}$. Meanwhile, anchors with the other variants in the queue construct $N \times (N \times K - 1)$ negative examples $\mathbb{N}_{AVV}$. The Anchor-Variant-View contrastive loss is defined as:

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

Anchor-Variant-View contrastive learning makes code snippets with the same functionality have similar representations in vector space, accelerating the model to retrieve all the proper code snippets based on a specific query.

4.4.4 Multi-View Contrastive Learning. As illustrated in Figure 4, we optimize MCodeSearcher from the above three contrastive views. To better learn the relationship between queries and snippets, we apply a bidirectional way to optimize our model at each view. For instance, in the aspect of Query-Anchor-View contrastive learning, the queue contains natural language queries by enqueuing and dequeuing peer mini-batch queries, which are regarded as negative instances, and code snippets are used as positive samples. For concision, we utilize $\mathcal{L}_{QAV}$, $\mathcal{L}_{QVV}$, and $\mathcal{L}_{AVV}$ to represent the bidirectional objective of each view contrastive learning. The final multi-view contrastive objective $\mathcal{L}_{MVCS}$ is formulated as:

$$\mathcal{L}_{MVCS} = \mathcal{L}_{QAV} + \mathcal{L}_{QVV} + \mathcal{L}_{AVV} \tag{4}$$

Based on multi-view contrastive learning, MCodeSearcher can effectively retrieve semantically equivalent code snippets from the codebase given a specific query.

4.5 Fine-tuning on Code Search

After the model is pre-trained on the multi-view contrastive objectives, we fine-tune it on code search task. Code search aims to retrieve the code snippet that matches the semantics of a given natural language query from a codebase. In this section, we mathematically formalize the code search task using some basic notations and terminologies. Specifically, we have a set $\mathcal{D} = \{\hat{q}_i, \hat{c}_i\}_{i=1}^{K}$ where

$\hat{q}_i$ and $\hat{c}_i$ denote a query and a code snippet. K denotes the total number of paired queries and code snippets in $\mathcal{D}$. In the training phase, given a mini-batch of R paired instances from $\mathcal{D}$, we feed each query $\hat{q}_i$ and code snippet $\hat{c}_i$ into the pre-trained encoder, and obtain semantic representation vectors $\hat{Q}_i$ and $\hat{C}_i$, respectively. For each query, we take other code snippets in the mini-batch as negative samples. Therefore, the label $\hat{y}_i$ of each example belongs to $\{0, ..., R-1\}$, which is denoted as the one-hot format. Then, we optimize MCodeSearcher to select the proper code snippet from candidate snippets. Finally, the training loss $\mathcal{L}_{CS}$ is formulated as:

$$sim(\hat{q}_i, \hat{c}_i) = cos(\hat{Q}_i, \hat{C}_i) = \frac{\hat{Q}_i^T \hat{C}_i}{\|\hat{Q}_i\| \|\hat{C}_i\|} \tag{5}$$

$$\mathcal{L}_{CS} = -\frac{1}{R} \sum_{i=0}^{R-1} \hat{y}_i log(sim(\hat{q}_i, \hat{c}_i)) \tag{6}$$

where $sim(q_i, c_i)$ denotes the cosine similarity for each instance. R is the number of examples in the mini-batch.

After fine-tuning, we evaluate MCodeSearcher as follows. Given a query, our model calculates the similarity between the query and each snippet in the codebase, utilizing Formula 5. Then the model ranks their similarities and selects snippets with higher similarities.

5 STUDT DESIGN

To verify the effectiveness of MCodeSearcher, we conduct a large-scale study to answer three research questions. Next, we will describe the details of our study, including datasets, baselines, and evaluation metrics.

5.1 Research Questions

Our study mainly answers the following research questions (RQ).

RQ1: How does MCodeSearcher perform in code search compared with the state-of-the-art models? RQ1 aims to evaluate that MCodeSearcher is more effective than other methods. We compare it with eight advanced models. Then, we assess their performance on five representative benchmarks.

RQ2: What is the performance of MCodeSearcher on zero-shot code search setting? RQ2 evaluates the robustness of MCodeSearcher on the zero-shot setting, where models are directly evaluated on the test set without fine-tuning.

RQ3: What is the contribution of each view contrastive learning for the code search performance? MCodeSearcher consists of multi-view contrastive objectives, e.g., Query-Anchor-View, Query-Variant-View, and Anchor-Variant-View. We verify the contribution of each view for code search.

5.2 Datasets

Datasets for milti-view contrastive learning. We utilize CodeSearchNet [15] dataset for multi-view contrastive learning. Specifically, we optimize MCodeSearcher using only the bimodal Java and Python corpus of CodeSearchNet, respectively, including 542,991 and 503,502 query-code pairs. For each code snippet, we generate one variant utilizing the translation tool and filter out the syntactic error programs. For the program that can not be translated, we use its original code snippet as the variant. Compared to the existing state-of-the-art retrieval model [20], the size of our contrastive dataset is much smaller. Following common practices [9, 13], we use the RoBERTa tokenizer to tokenize the code and the natural language query, which is based on Byte-Pair Encoding (BPE) algorithm [35]. The size of the queue is set to 128. The τ is set to 0.05, and the dropout probability is set to 0.1. We use the AdamW [26] for optimizing. The initial learning rate is 1e-4, and the warm-up step is 2000. We optimize MCodeSearcher for 100,000 steps on 4×NIVDIA V100S with a total batch size of 128.

Table 1: Statistics of code search datasets.

Dataset	Train	Valid	Test	Codebase
CodeSearchNet-Java [15]	164923	5183	10955	40,347
CodeSearchNet-Python [15]	251820	13914	14918	43827
CoSQA [14]	19604	500	500	6267
StaQC [43]	203700	2600	2749	14579
WebQuery [28]	–	–	1,046	–

Datasets for Code Search. We conduct experiments on five representative datasets including CodeSearchNet [15], CoSQA [14], StaQC [43], and WebQuery [28]. For CodeSearchNet dataset, we process it and make sure no overlapping instances with the dataset used in multi-view contrastive learning. We only evaluate MCodeSearcher on Java and Python languages of CodeSearchNet since TransCoder-ST only supports Java, Python, and C++ languages during multi-view contrastive learning. CoSQA dataset [14] consists of 19,604 labeled instances in Python language, where the queries are collected from the search logs of the Microsoft Bing search engine [3] and the code snippets come from Github [4]. StaQC dataset [43] is crawled from Stack Overflow [5], which contains a large collection of annotated code snippets, and some queries are linked to one or more relevant snippets in the collection.

Considering that code search engines aim to find code snippets that are semantically equivalent to a query, it can be divided into two scenarios: code retrieval task and code question answering task. To further evaluate MCodeSearcher, we further introduce WebQuery dataset [28] for code question answering. The dataset only contains the test set in Python language. We use the training set of the CoSQA dataset to train our model and test it on WebQuery since the two datasets have almost the same collections and data format. The statistics of all datasets are shown in Table 1.

5.3 Baselines

We compare our MCodeSearcher with eight advanced models. They are all transformer-based pre-trained methods and can be divided into two categories according to their architectures: the encoder-only method and the encoder-decoder method. The encoder-only method includes RoBERTa [25], CodeBERT [9], GraphcodeBERT [13], SyncoBERT [41], CoCoSoDa [36], and CodeRetriever [20]. The encoder-decoder method contains SPT-Code [30] and UniXcoder [12], which have more parameters than the encoder-only model. Our MCodeSearcher is the encoder-only method.

- **RoBERTa** [25]: Pre-trained on CodeSearchNet [15] corpus with masked language modeling [5] task.

[3] https://www.bing.com/
[4] https://github.com/
[5] https://stackoverflow.com/

Table 2: Comparison of the performances on code retrieval task and code question answering task. All models are fine-tuned on the same datasets. CSN denotes the CodeSearchNet dataset. Numbers in bold mean the best score among all models.

	Models	Python (CSN)	Java (CSN)	CoSQA	StaQC	WebQuery	
		MRR	MRR	MRR	MRR	F1	Acc
Encode-Only	RoBERTa [25]	58.70	59.90	60.30	20.85	40.55	45.41
	CodeBERT [9]	67.20	67.60	64.70	23.40	41.48	51.27
	GraphcodeBERT [13]	69.20	69.10	67.50	23.80	44.31	54.39
	SyncoBERT [41]	72.40	72.30	–	–	–	–
	CoCoSoDa [36]	71.70	72.60	–	–	–	–
	CodeRetriever [20]	75.80	76.50	75.40	24.20	–	–
Encode-Decoder	SPT-Code [30]	69.90	70.00	–	–	–	–
	UniXcoder [12]	72.17	72.67	70.10	25.74	45.02	55.74
	MCodeSearcher	**83.54** **(↑ 10.21%)**	**79.57** **(↑ 4.01%)**	**77.23** **(↑ 2.43%)**	**25.93** **(↑ 0.74%)**	**46.53** **(↑ 3.35%)**	**57.07** **(↑ 2.39%)**

- **CodeBERT** [9]: Learn unified representations for both code snippets and natural language queries with masked language modeling and replaced token detection tasks.
- **GraphCodeBERT** [13]: Consider the data flow of code snippets, and introduce two structure-aware tasks, e.g., data flow edge prediction task and node alignment task.
- **SyncoBERT** [41]: Construct a syntax-guided contrastive pre-training approach. Besides, the model also introduces the identifier prediction task and the abstract syntax tree (AST) edge prediction task.
- **SPT-Code** [30]: Propose three code-oriented sequence-to-sequence tasks, including masked language task, code-AST prediction task, and method name generation task.
- **UniXcoder** [12]: It is a unified contrastive pre-trained method that leverages multi-modal contents, i.e., queries and ASTs, to support code-related tasks.
- **CoCoSoDa** [36]: It augments queries and programs, and proposes multimodal contrastive objectives for code search.
- **CodeRetriever** [20]: The model combines unimodal and bimodal contrastive objectives to learn function-level semantic representations of code snippets.

5.4 Evaluation Metrics

For automatic evaluation, we adopt Mean Reciprocal Rank (MRR) to measure the effectiveness of MCodeSearcher. MRR is a statistic measure algorithm that is widely used in code search task [13, 30, 40]. Specifically, MRR is the average of the reciprocal ranks of results for a set of queries. The reciprocal rank of a query is the inverse of the rank of the first matched result. The MRR is formulated as:

$$\text{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\text{Rank}_i} \tag{7}$$

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

For code question answering, it asks models to judge whether a code snippet answers a given natural language query, which can be formulated into a binary classification problem. We use the F1 score and Accuracy value to measure results.

6 RESULTS AND ANALYSES

In our first research question, we evaluate the performance of MCodeSearcher with respect to other code search methods.

RQ1: How does MCodeSearcher perform in code search compared with state-of-the-art models?

Setup. We compare MCodeSearcher with eight advanced code search approaches (Section 5.3). Then, we use MRR, F1, and Accuracy (Section 5.4) to measure the performance of different approaches on five code search datasets (Section 5.2).

Results. The results on five datasets are shown in Table 2. The values in parentheses are relative improvements compared to the state-of-the-art baseline.

Analyses. (1) MCodeSearcher achieves the best performance among all models. Compared to the state-of-the-art baselines, in terms of MRR, MCodeSearcher brings 10.21% and 4.01% relative improvements on CodeSearchNet-Python and -Java corpus; on CoSQA and StaQC datasets, our method achieves 2.43% and 0.74% relative improvements. The significant improvements prove the effectiveness of our MCodeSearcher. (2) MCodeSearcher is superior to other contrastive learning methods. Among all baselines, besides the first three baselines, other models all introduce the contrastive learning method, especially CoCoSoDa and CodeRetriever designed for code search. The performance of our MCodeSearcher is better than the counterpart of these methods, which demonstrates the superiority of our multi-view contrastive learning. We attribute the impressive performance to two reasons. The first is that MCodeSearcher learns the semantic correlation between code snippets and queries from Query-Anchor-View and Query-Variant-View contrastive learning. Another reason is that Anchor-Variant-View contrastive learning prompts the representations of source codes, despite facing the diversity of code competition scenarios. (3) The performance of MCodeSearcher with an encode-only backbone is better than the encode-decoder models. As described in Section 5.3, SPT-Code and UniXcoder are both encoder-decoder models that have more parameters than MCodeSearcher. Our CodeSearcher can still retrieve more correct code snippets which demonstrates the effectiveness and efficiency of our method. (4) MCodeSearcher also has an advantage to the code question answering. It achieves the best results on the WebQuery dataset. The satisfying results show that our

Table 3: Evaluation results of different models on the zero-shot setting. All models are evaluated without fine-tuning.

Models	Python (CSN)	Java (CSN)	CoSQA	StaQC	WebQuery	
	MRR	MRR	MRR	MRR	F1	Acc
CodeBERT	0.03	0.02	1.16	0.00	2.25	4.32
GraphCodeBERT	0.40	0.70	0.80	0.20	3.72	6.27
CodeRetriever	67.70	69.00	47.50	15.50	–	–
MCodeSearcher	**76.38** (↑ **12.82%**)	**75.71** (↑ **9.72%**)	**71.64** (↑ **50.82%**)	**17.30** (↑ **11.61%**)	**26.95** (↑ **389.11%**)	**31.87** (↑ **328.94%**)

model can correctly judge whether a code snippet answers a given query, and further demonstrate that MCodeSearcher can effectively identify the semantic similarity of queries and code snippets.

Answer to RQ1: MCodeSearcher achieves the best performance on five datasets. In terms of MRR, MCodeSearcher outperforms it by up to 10.21% in CSN (Python), 4.01% in CSN (Java), 2.43% in CoSQA and 0.74% in StaQC. Besides, AceCoder is effective in code question answering with 3.35% relative improvements on F1. The significant improvements prove the effectiveness of MCodeSearcher in code search task.

RQ2: What is the performance of MCodeSearcher on zero-shot code search setting?

Setup. In this RQ, we verify MCodeSearcher on the zero-shot setting. Specifically, we directly evaluate baselines and our model on the test set without fine-tuning. Since the parameters of SyncoBERT, CoCoSoDa and SPT-Code are unavailable, we only compare our MCodeSearcher with the remaining models.

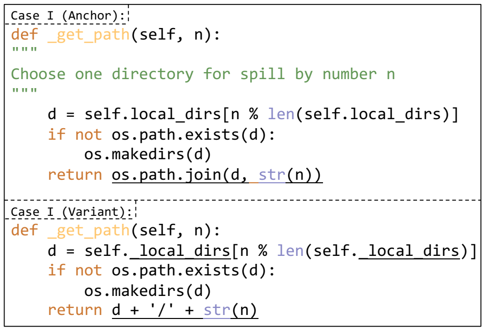
Results. The experimental results on five datasets are shown in Table 3. The values in parentheses are relative improvements compared to other models.

Analyses. (1) MCodeSearcher achieves state-of-the-art performance compared with other baselines in the zero-shot setting. Specifically, 12.82% and 9.72% relative improvements are achieved on MRR at CodeSearchNet dataset than the competitive model CodeRetriever. The impressive performance demonstrates that our multi-view contrastive learning can sufficiently grasp the semantic relationship of code snippets and queries even though no labeled instances for fine-tuning. (2) The parameters of MCodeSearcher are more suitable for code search than the parameters of GraphCodeBERT that initialize our model. The performance of MCodeSearcher is significantly superior to GraphCodeBERT, which reveals that our multi-view contrastive learning makes the model effectively capture the relationship between queries and code snippets.

Answer to RQ2: MCodeSearcher also has an advantage in code search on the zero-shot setting. That is, without fine-tuning, our MCodeSearcher achieves the best results on all datasets, which verifies the rationality and effectiveness of our novel multi-view contrastive learning.

RQ3: What is the contribution of each view contrastive learning for the code search performance?

Setup. We complete a comprehensive ablation study to investigate the impact of each view of contrastive learning. We remove each contrastive objective individually from our model.



```
Case I (Anchor):
def _get_path(self, n):
"""
Choose one directory for spill by number n
"""
    d = self.local_dirs[n % len(self.local_dirs)]
    if not os.path.exists(d):
        os.makedirs(d)
    return os.path.join(d, str(n))

Case I (Variant):
def _get_path(self, n):
    d = self._local_dirs[n % len(self._local_dirs)]
    if not os.path.exists(d):
        os.makedirs(d)
    return d + '/' + str(n)
```

Figure 5: The automatically generated code snippets through back-translation transformation.

Results. The experimental results are shown in Table 4. "w/o Γ" denotes removing Γ module, where Γ includes the Query-Anchor-View contrastive learning (QAV), Query-Variant-View contrastive learning (QVV) and Anchor-Variant-View contrastive learning (AVV).

Analyses. (1) All three views of contrastive learning are beneficial for code search. After removing any of the views, the performance drops. It validates our multi-view contrastive learning can grasp the relationship between queries and code snippets from different perspectives that benefits code search. (2) The importance rank of the three views in terms of metrics is that QAV > QVV > AVV. Particularly, QAV and QVV play an important role in improving code search task. The reason might be that these two objectives focus more on grasping the relationship between queries and code snippets, which are directly related to the code search task and can provide more effective guidance.

Answer to RQ3: Each view of contrastive learning is effective for code search. QAV and QVV are more important than AVV.

7 DISCUSSION

7.1 Quality Analysis of Generated Variants by Translation

As described in Section 4.2, TransCoder-ST is pre-trained on a large amount of parallel programming data with an automated unit test pipeline, resulting in being a reliable tool. To intuitively demonstrate the quality and diversity of augmented code snippets, we present a parallel example (e.g., anchor and its variant) in Figure 5.

We can find that the generated snippet has similar semantics to its anchor. Meanwhile, its variant is different in lexicon and implementations compared with the anchor. Specifically, both of

Table 4: Effects of each view contrastive learning.

Models	Python (CSN)	Java (CSN)	CoSQA	SO-DS	StaQC	WebQuery	
	MRR	MRR	MRR	MRR	MRR	F1	Acc
MCodeSearcher	83.54	79.57	77.23	32.19	25.93	46.53	57.07
w/o QAV	77.17	77.28	74.46	29.23	24.57	45.39	55.81
w/o QVV	78.24	77.53	73.92	30.03	25.05	45.76	56.35
w/o AVV	80.53	78.09	74.85	30.47	24.82	46.16	56.43

```
class MyClass:
    def __getitem__(self, key):
        return key * 2
```

(a) GraphCodeBERT (Top 1)

```
def isclass(object):
    #Return true if the object is a class.
    return isinstance(object, (type, types.ClassType)
```

(b) MCodeSearcher (Top 1)

```
import inspect
class X(object):
    inspect.isclass(X)
x = X()
isinstance(x, X)
y = 25
isinstance(y, X)
```

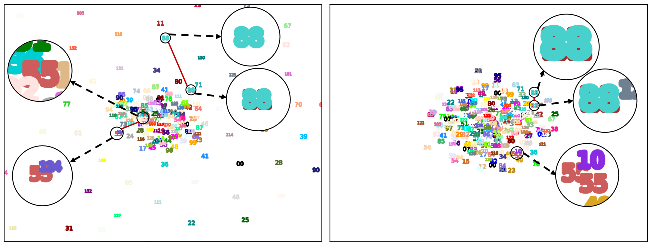
(c) MCodeSearcher (Top 2)

Figure 6: Retrieved results of GraphCodeBERT and MCodeSearcher for the query "Check whether a variable is a class".

"os.path.join(d, str(n))" and "d + '/' + str(n)" means getting a directory, but they are implemented in disparate methods with different syntax structures. Thus, the transformation has the reliable ability to generate semantically equivalent and diverse code snippets for multi-view contrastive learning.

7.2 Quality Analysis on Code Search

7.2.1 Features visualization of queries and code snippets. To intuitively understand why our multi-view contrastive learning achieves impressive performance on code search, we give a visualization of code vectors and query vectors on GraphCodeBERT and MCodeSearcher by t-SNE [38]. Specifically, we randomly select one hundred query-code pairs in CoSQA dataset [14] for visualizing since the dataset can reflect the distribution of real user queries issued in search engines. Then we feed them into pre-trained GraphCodeBERT and MCodeSearcher, where both of them are without fine-tuning, to obtain representation vectors of queries and code snippets, respectively. Then t-SNE is applied to reduce the dimensionality of vectors into two-dimensional space. As presented in Figure 7, each number represents a query or a code snippet, and each query-code pair shares the same color and number. Figure 7(a) shows the result of GraphCodeBERT. We can observe that the vectors of paired query and code snippet are far apart though they have the same semantics. Figure 7(b) illustrates the results of MCodeSearcher, where the features of code snippets and queries are well entangled, and each query-code pair in the vector space is clustered by functionality. For example, the distance of vectors of the "55" (or "88") query-code pair is much smaller after our multi-view contrastive learning. This phenomenon can be explained by the fact that our multi-view contrastive learning can effectively cluster the semantically equivalent queries and code snippets, and facilitate MCodeSearcher to retrieve the semantically equivalent code snippets from the codebase given a certain query.



(a) GraphCodeBERT (b) MCodeSearcher

Figure 7: t-SNE visualization of representations of queries and code snippets in GraphCodeBERT and MCodeSearcher without fine-tuning. Each number represents a query or a code snippet, and each query-code pair shares the same color and number.

7.2.2 Case Study. To further provide some insights on MCodeSearcher, we show some retrieved cases on GraphCodeBERT and MCodeSearcher. Figure 6 shows the retrieved result of the query "Check whether a variable is a class". "Top K" means the result is ranked in the K-th position among the retrieved list. We can find that the retrieved code snippets of GraphCodeBERT are completely unmatched with the query. When turning to MCodeSearcher, the first two retrieved results both satisfy the demand of the query, which demonstrates that our multi-view contrastive learning method can facilitate the model to identify semantically equivalent code snippets that match the query.

8 CONCLUSION AND FUTURE WORK

In this paper, we propose a novel multi-view contrastive learning model, MCodeSearcher, for code search task. MCodeSearcher considers the Query-Anchor-View, Query-Variant-View, and Anchor-Variant-View contrastive objectives to promote identifying the semantic similarities or dissimilarities between queries and code snippets. We conduct extensive experiments on five representative datasets. The experimental results show that the proposed MCodeSearcher substantially outperforms the state-of-the-art models.

ACKNOWLEDGMENTS

All authors except Fang Liu belong to Key Lab. of High Confidence Software Technologies (PKU), Ministry of Education. This research is supported by the National Natural Science Foundation of China under Grant No. 62072007, 62192733, 61832009, 62192731, 62192730. We also would like to thank all the anonymous reviewers for constructive comments and suggestions to this paper.

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).

[2] 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*. 511–521.

[3] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 964–974.

[4] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*. PMLR, 1597–1607.

[5] 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* (2018).

[6] Yangruibo Ding, Luca Buratti, Saurabh Pujar, Alessandro Morari, Baishakhi Ray, and Saikat Chakraborty. 2021. Contrastive Learning for Source Code with Structural and Functional Properties. *arXiv preprint arXiv:2110.03868* (2021).

[7] Lun Du, Xiaozhou Shi, Yanlin Wang, Ensheng Shi, Shi Han, and Dongmei Zhang. 2021. Is a single model enough? mucos: A multi-model ensemble learning approach for semantic code search. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 2994–2998.

[8] Mengnan Du, Varun Manjunatha, Rajiv Jain, Ruchi Deshpande, Franck Dernoncourt, Jiuxiang Gu, Tong Sun, and Xia Hu. 2021. Towards interpreting and mitigating shortcut learning behavior of NLU models. *arXiv preprint arXiv:2103.06922* (2021).

[9] 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* (2020).

[10] Jian Gu, Zimin Chen, and Martin Monperrus. 2021. Multimodal Representation for Neural Code Search. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 483–494.

[11] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944.

[12] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. *arXiv preprint arXiv:2203.03850* (2022).

[13] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[14] 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* (2021).

[15] 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* (2019).

[16] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2073–2083.

[17] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E Gonzalez, and Ion Stoica. 2020. Contrastive code representation learning. *arXiv preprint arXiv:2007.04973* (2020).

[18] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. 2014. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. 173–184.

[19] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511* (2020).

[20] 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. *arXiv preprint arXiv:2201.10866* (2022).

[21] Chunyang Ling, Zeqi Lin, Yanzhen Zou, and Bing Xie. 2020. Adaptive deep code search. In *Proceedings of the 28th International Conference on Program Comprehension*. 48–59.

[22] Xiang Ling, Lingfei Wu, Saizhuo Wang, Gaoning Pan, Tengfei Ma, Fangli Xu, Alex X Liu, Chunming Wu, and Shouling Ji. 2021. Deep graph matching and searching for semantic code retrieval. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 15, 5 (2021), 1–21.

[23] Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. 2009. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery* 18, 2 (2009), 300–336.

[24] Fang Liu, Jia Li, and Li Zhang. 2023. Syntax and Domain Aware Model for Unsupervised Program Translation. *arXiv preprint arXiv:2302.03908* (2023).

[25] 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 (2019). arXiv:1907.11692 http://arxiv.org/abs/1907.11692

[26] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).

[27] Meili Lu, Xiaobing Sun, Shaowei Wang, David Lo, and Yucong Duan. 2015. Query expansion via wordnet for effective code search. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 545–549.

[28] 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* (2021).

[29] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on api understanding and extended boolean model (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 260–270.

[30] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. SPT-Code: Sequence-to-Sequence Pre-Training for Learning the Representation of Source Code. *arXiv preprint arXiv:2201.01549* (2022).

[31] Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Raymond J Mooney. 2020. Deep Just-In-Time Inconsistency Detection Between Comments and Source Code. *arXiv preprint arXiv:2010.01625* (2020).

[32] Stephen Robertson and Hugo Zaragoza. 2009. *The probabilistic relevance framework: BM25 and beyond*. Now Publishers Inc.

[33] Baptiste Roziere, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. 2021. DOBF: A deobfuscation pre-training objective for programming languages. *arXiv preprint arXiv:2102.07492* (2021).

[34] Baptiste Roziere, Jie M Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2021. Leveraging Automated Unit Tests for Unsupervised Code Translation. *arXiv preprint arXiv:2110.06773* (2021).

[35] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909* (2015).

[36] Ensheng Shi, Wenchao Gub, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2022. Enhancing Semantic Code Search with Multimodal Contrastive Learning and Soft Data Augmentation. *arXiv preprint arXiv:2204.03293* (2022).

[37] Jianhang Shuai, Ling Xu, Chao Liu, Meng Yan, Xin Xia, and Yan Lei. 2020. Improving code search with co-attentive representation learning. In *Proceedings of the 28th International Conference on Program Comprehension*. 196–207.

[38] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[40] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip Yu. 2019. Multi-modal attention network learning for semantic source code retrieval. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 13–25.

[41] Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2021. SynCoBERT: Syntax-Guided Multi-Modal Contrastive Pre-Training for Code Representation. *arXiv preprint arXiv:2108.04556* (2021).

[42] 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* (2021).

[43] 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*. 1693–1703.

[44] 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 *2018 IEEE/ACM 15th international conference on mining software repositories (MSR)*. IEEE, 476–486.

[45] Kechi Zhang, Zhuo Li, Zhi Jin, and Ge Li. 2023. Implant Global and Local Hierarchy Information to Sequence based Code Representation Models. *arXiv preprint arXiv:2303.07826* (2023).

[46] Kechi Zhang, Wenhan Wang, Huangzhao Zhang, Ge Li, and Zhi Jin. 2022. Learning to represent programs with heterogeneous graphs. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 378–389.