You Augment Me: Exploring ChatGPT-based Data Augmentation for Semantic Code Search

Yanlin Wang[a,†] Lianghong Guo[b] Ensheng Shi[c,†] Wenqing Chen[a] Jiachi Chen[a]
Wanjun Zhong[a] Menghan Wang[d] Hui Li[e] Hongyu Zhang[f] Ziyu Lyu[g] Zibin Zheng[a]
[a]Sun Yat-sen University [b]Beijing University of Posts and Telecommunications
[c]Xi'an Jiaotong University [d]eBay Inc. [e]Xiamen University [f]Chongqing University
[g]Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences
{wangylin36chenwq95, chenjch86, zhongwj25, zhzibin}mail.sysu.edu.cn
glhxystxdy123@gmail.com, s1530129650@stu.xjtu.edu.cn
wangmengh@zju.edu.cn hui@xmu.edu.cn
hyzhang@cqu.edu.cn, zy.lv@siat.ac.cn

Abstract*—Code search plays a crucial role in software development, enabling developers to retrieve and reuse code using natural language queries. While the performance of code search models improves with an increase in high-quality data, obtaining such data can be challenging and expensive. Recently, large language models (LLMs) such as ChatGPT have made remarkable progress in both natural and programming language understanding and generation, offering user-friendly interaction via simple prompts. Inspired by these advancements, we propose a novel approach ChatDANCE, which utilizes high-quality and diverse augmented data generated by a large language model and leverages a filtering mechanism to eliminate low-quality augmentations. Specifically, we first propose a set of ChatGPT prompting rules that are specifically designed for source code and queries. Then, we leverage ChatGPT to rewrite code and queries based on the according prompts and then propose a filtering mechanism which trains a cross-encoder from the backbone model UniXcoder to filter out code and query pairs with low matching scores. Finally, we re-train the backbone model using the obtained high-quality augmented data. Experimental results show that ChatDANCE achieves state-of-the-art performance, improving the best baseline by 13.2% (R@1) and 7% (MRR). Surprisingly, we find that this augment-filter-retrain strategy enables the backbone model (UniXcoder) to self-grow. Moreover, extensive experiments show the effectiveness of each component and ChatDANCE has stable performance under different hyperparameter settings. In addition, we conduct qualitative and quantitative analyses to investigate why ChatDANCE works well and find that it learns a more uniform distribution of representations and effectively aligns the code and query spaces. We have made the code and data anonymously available at https://anonymous.4open.science/r/ChatDANCE.

***Index Terms*—Code Search, Data Augmentation, ChatGPT**

I. Introduction

With the rapid growth of open-source code repositories on platforms like GitHub [1], code search has become crucial in software engineering. This task aims to find the code snippet in a repository that best matches users' intention, given a query written in natural language [2]. Code search enables developers to find and reuse relevant code snippets in software development and maintenance [3], [4].

Early studies [5]–[8] in code search often rely on traditional information retrieval (IR) techniques, such as matching keywords based on lexical information of code snippets. With the popularity of deep learning, neural code search models [9]–[23] begin to emerge. For example, DeepCS [9] utilizes neural models to encode queries and codes into a shared vector space and measures similarity using vector distance. Later, pre-trained models [24]–[31] emerge and surpass the conventional neural models in code search. These models better understand source code and natural language by pre-training on vast amounts of code and natural language data. Finetuning such models can achieve excellent results on downstream tasks such as code search. For example, UniXcoder [27] is a unified cross-modal pre-trained model for programming languages that utilize mask attention matrices with prefix adapters to control the model's behavior and leverages cross-modal contents like AST and code comment to enhance code representation. By finetuning, UniXcoder significantly improves most downstream tasks, such as code search and summarization.

Despite the significant advantages of deep learning, two main bottlenecks prevent neural models from achieving high performance: 1) the lack of high-quality labeled training data and 2) the difference in data distribution between the training and testing datasets [32]. To overcome these challenges, a straightforward solution is to increase the size and diversity of the training data by data augmentation. For example, Huang et al. [33] propose Query-Rewritten Augmentation (QRA) to generate augmented queries by conducting minor modifications on queries in the training dataset. Chakraborty et al. [34] propose a code augmentation method called NatGen to generate augmented codes using six semantics-preserving transformations to rewrite codes based on their AST structure. Later, Jain et al. [31] and Bui et al. [30] use similar code augmentation methods to pre-train their models. Recently, Shi et al. [28] propose Soft Data augmentation (SoDa) to pre-train code search models by masking tokens in data dynamically. However, current data augmentation methods still have room

[†] Yanlin Wang and Ensheng Shi are the ding authors.

for improvement. First, there is no unified semantics-similar [1] data augmentation method that can enhance both queries and code simultaneously. The QRA [33] and NatGen [30], [31], [34] are limited to augmenting only one modality of the data, such as code or query. Although SoDa [28] can augment both queries and codes, it is a non-semantics-similar method. Secondly, the scalability and diversity of the augmented data generated by the current methods are limited. Specifically, NatGen requires specific constraints and can not be applicable to all source code. QRA just adopts simple modifications such as random word deletion and falls short in terms of producing diverse augmentations.

Recently, large language models (LLMs) such as ChatGPT have made remarkable progress in both natural and programming language understanding and generation, offering user-friendly interaction via simple prompts. Inspired by these advancements, we propose **CHATDANCE**, **Chat**GPT-based **D**ata **A**ugme**N**tation for **C**ode s**E**arch). Our method can exploit LLMs such as ChatGPT to generate a large number of high-quality and diverse data using two core components: ChatGPT-based data augmentation and model-based data filtering. First, we use the concept of data augmentation [35] and request the LLM to rewrite data by making semantic-similar modifications to the original data, which can effectively preserve most of the semantics of the generated data. To efficiently interact with the model and perform data augmentation, we designed two prompt templates for queries and codes, respectively. These templates contain essential information for the rewriting task, such as task definition and critical additional information. Moreover, our method allows users to inject specific prior knowledge to guide the LLM in data augmentation. For instance, in code augmentation, we designed five rewriting techniques to guide the model to generate new data according to specific patterns. Second, to further improve the quality of the generated data, we trained a filtering model on the original dataset to score and filter the augmented samples. The filtering model evaluates the quality of the generated data and discards low-quality data. With our proposed CHATDANCE method, we can generate a large number of diverse and high-quality data, which can be used to enhance the generalization ability and performance of models in downstream tasks such as code search.

The overall framework of CHATDANCE is presented in Figure 1. Our approach consists of three stages: (a) data augmentation via ChatGPT, (b) data filtering, and (c) model training. In part (a) of Figure 1, we demonstrate how we achieve data augmentation using ChatGPT. Given a query-code pair, we first construct query and code augmentation prompts using different prompt templates for the query and code, respectively. The prompt template is crucial for effectively interacting with ChatGPT. It includes the definition of the data rewriting task, important additional information such as the number of augmented data, and prior knowledge for completing the task. We then request ChatGPT to rewrite the original data based on the prompt information, generating augmented data. Finally, we extract the augmented data from ChatGPT's response using regular expressions.

In the data filtering stage, we first train a filtering model to remove low-quality augmented data to further improve data quality. Specifically, we train a model based on the cross-encoder architecture, which can directly score the matching degree of query-code pairs. We use the scores generated by the filtering model as the basis for filtering the data. Next, we use the filtering model to score the matching degree of the augmented query-code pairs. We filter out query-code pairs with scores below the filtering threshold and obtain high-quality data. By removing low-quality data, we can ensure that the generated data is of high quality and can effectively enhance the model's performance.

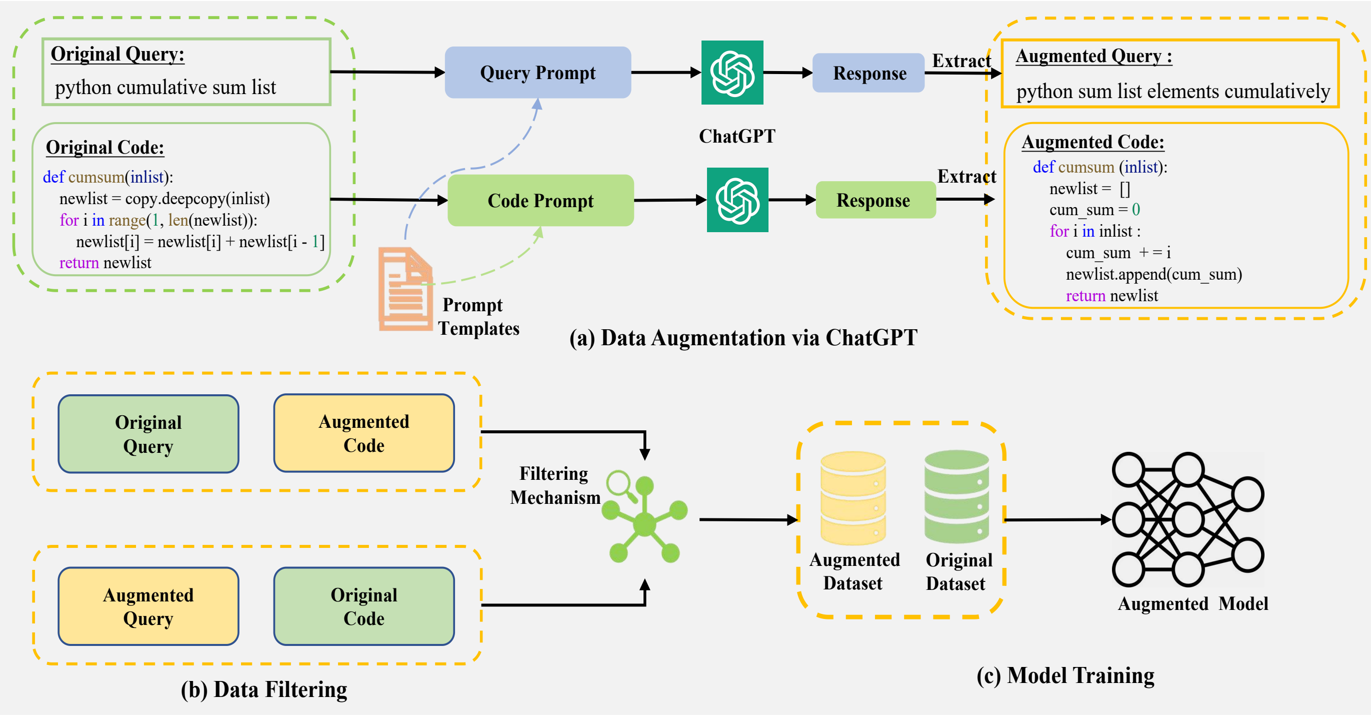
Finally, after filtering the augmented dataset, we combine it with the original dataset. Then, we use UniXcoder as our baseline model with a bi-encoder structure and fine-tune the model on the final dataset using contrastive loss to improve the model's performance.

We evaluate the effectiveness of our approach on the CoSQA dataset, which contains a large number of real-world queries. We apply our approach to the state-of-the-art model UniXcoder in the CoSQA dataset and compare our approach with semantic-preserving data augmentation methods: QRA and NatGen. We also conduct ablation studies to investigate the effectiveness of each component of our approach and explore the impact of different hyperparameters on our method. Finally, we conduct qualitative and quantitative analyses to investigate why our approach works. The results of the experiments demonstrate that: (1) Our approach can significantly improve the model's performance and outperform the baselines. (2) Each component of our approach contributes significantly to improving the model's performance. (3) Our method ensures stable performance for the model across various hyperparameter settings, including query filtering threshold ranging θ_q from 0.7 to 0.95, code filtering threshold θ_c ranging from 0.7 to 0.9, learning rate ranging from 1e-5 to 5e-5 and the average number of augmented samples greater than 5. (4) Compared to baselines, our method can effectively improve the alignment and uniformity of the representations learned by the model.

We summarize the contributions of this paper as follows:

- We propose a new data augmentation approach for code search, which uses ChatGPT to generate a large number of high-quality data. We also introduce a prompt schema to improve the interaction with ChatGPT, allowing users to design their own prompts and provide the necessary information to complete the task effectively.
- We propose a cross-encoder-based data filtering mechanism that scores code-query pairs and filters low-quality pairs. This approach can be applied to data collection and augmentation, resulting in improved data quality.

[1] By design, our goal is to generate semantic-preserving augmentations, and the majority of them are deemed semantic-preserving upon manual checking. However, due to the inherent uncertainty and opacity of ChatGPT, we cannot guarantee that every generated augmentation is strictly semantic-preserving. Therefore, we use the term "semantic-similar" to describe these augmentations.



```
def cumsum(inlist):
    newlist = copy.deepcopy(inlist)
    for i in range(1, len(newlist)):
        newlist[i] = newlist[i] + newlist[i - 1]
    return newlist
```

```
def cumsum (inlist):
    newlist = []
    cum_sum = 0
    for i in inlist :
        cum_sum  + = i
        newlist.append(cum_sum)
    return newlist
```

Fig. 1: An overview of CHATDANCE.

- We conduct extensive experiments to evaluate the effectiveness of our method on the CoSQA dataset. The experimental results show that our method can significantly outperform baselines. Furthermore, our method can effectively improve the alignment and uniformity of the learned representations and exhibits stable performance across a range of hyperparameters.

II. Related Work

A. Code Search

Code search is a crucial aspect of software development and maintenance [3], [4]. In general, models for code search can be classified into two categories: information retrieval (IR)-based models [5]–[8] and deep learning-based models [9]–[31]. IR-based models often use keyword matching or text similarity to retrieve relevant code. In recent years, deep learning-based models have become the mainstream approach for code search and have achieved promising results. For instance, Gu et al. [9] proposed the first neural code search model, CODEnn, which embeds queries and code into a shared vector space and calculates similarity by vector distance. Since then, various deep learning-based models have been developed, including sequence models [10], [12], [15], [18], [19], [22], convolutional neural networks [14], [17], [21], and graph neural networks [11], [18]. Recently, pre-trained models [24]–[31] have emerged as a powerful tool for code search, outperforming traditional neural models. Pre-trained models leverage massive amounts of programming language and natural language data to develop strong code understanding capabilities and achieve excellent performance on various code-related tasks, including code search. For example, Feng et al. [25] proposed CodeBERT, a bimodal pre-trained model that learns representations for both programming language and natural language. Guo et al. proposed GraphCodeBERT [26], which utilizes data flow to pre-train the model and improve the representation of code. UniXcoder [27], a unified cross-modal pre-trained model for programming languages, uses mask attention matrices with prefix adapters to control the model's behavior and leverages cross-modal contents, such as abstract syntax trees and code comments, to enhance code representation. In this paper, we adopt UniXcoder as our baseline model, as it is the state-of-the-art model on the CoSQA dataset [33].

B. Large Language Model and In-context Learning

Large language models (LLMs) typically refer to language models with hundreds of billions or more parameters [36]. Trained on large amounts of text data, LLMs such GPT-3 [37], PaLM [38], and LLaMA [39] demonstrate impressive performance on various downstream tasks such as machine translation, code generation, and more. As the model parameters and size of training data further increase, some emergent abilities of LLMs have been observed when the model size exceeds a certain level.

One of large language models' impressive emergent abilities is their in-context learning capability. The in-context learning ability is formally introduced in GPT-3 [37], which enables the model to generate the expected output for test instances by completing the input text's word sequence, given natural language instructions and/or task demonstrations [36]. Inspired by the concept of in-context learning, we introduce a novel data augmentation approach that leverages large language models such as ChatGPT. By providing task instructions to the model, we can prompt it to rewrite existing data and generate high-quality augmented data.

C. Data Augmentation in Code Search

Data augmentation is a common method used in code search to help models achieve better generalization ability and performance. For instance, Huang et al. [33] propose Query-Rewritten Augmentation (QRA) to generate augmented queries by conducting small modifications on queries. The QRA performs query augmentation by three transformations such as (1) deleting a word randomly, (2) copying a word randomly, and (3) switching the position of two words randomly. In addition, Chakraborty et al. [34] propose a code augmentation method called NatGen, which includes six semantics-preserving transformations: (1) Loop Transformation, (2) DeadCode Injection, (3) Operand Swap, (4) Block Swap, (5) Variable Renaming, (6) Confusing Code Insertion. This method chooses appropriate code transformations based on the AST structure to rewrite the code. Later, Jain et al. [31] and Bui et al. [30] adopt similar code augmentation methods for pre-training their models. Recently, Shi et al. [28] proposed a non-semantics-similar method called Soft Data Augmentation (SoDa) to pre-train code search models by dynamically masking tokens in the data. Compared to the previous approach, ChatGPT Data Augmentation is semantics-similar and suitable for both queries and codes. Moreover, by exploiting the powerful generation ability of LLM, our method can generate data with better diversity and scalability.

III. ChatDANCE Framework

This section introduces our straightforward and effective data augmentation framework via ChatGPT for augmenting training data on code search. The framework consists of three subsequent stages, *the data augmentation stage*, *the data filtering stage*, and *the model training stage*. An overview of the framework, when applied to augment query-code pairs in the training dataset, is shown in Figure 1.

In the first stage, we separately augment both query and code modalities to create augmented samples. For query augmentation, we request ChatGPT to rewrite an input query without changing its semantics. Then we pair the rewritten query with its original code to form a new augmented sample. For code augmentation, we request ChatGPT to rewrite the code with the guidance of the five given rewriting techniques, and then we pair the rewritten code with its original query to form a new augmented sample.

Next, considering the existence of low-quality augmented samples that may introduce noise for model training, in the second stage, we trained a cross-encoder model to compute matching scores for augmented query-code pairs. The matching scores serve as the basis for filtering out sample pairs that do not meet a certain threshold.

Finally, we train the model on the augmented dataset to improve its performance. In the following, we will provide a detailed explanation of the design of each stage.

A. The Data Augmentation Stage

1) Prompt Schema: Pre-trained on massive and unlabeled corpora, Large Language Models (LLMs) such as ChatGPT have demonstrated impressive emergent capabilities when subjected to model scaling. Instead of fine-tuning large models on specific tasks, complex problems such as machine translation and code generation can be solved simply by interacting with the LLMS using appropriate prompts. Therefore, constructing a good prompt is critical to effectively using a large language model. Inspired by *Natural Instruction* [40], we develop a similar prompt schema to build our data augmentation request to ChatGPT. Below we present the ingredients of our schema:

- Instruction provides detailed content about the task, which often includes task input, task output, and approach to complete the task.
- Emphasis and Caution provides important additional requirements to ensure the effective completion of the task.
- Prior Knowledge provides ChatGPT with prior knowledge to efficiently accomplish the rewriting task.
- Task Input provides the input content for the task.
- Outputs Context provides ChatGPT with the context of returning task outputs.

2) Prompt Design: Following the prompt schema in Section III-A1, We design different prompts based on the characteristics of query and code data, respectively. Next, we separately elaborate on the content of the query prompt and code prompt.

Query Augmentation Prompt: The details of the query augmentation prompt is shown in Table I. (1) *Instruction*: We aim to generate augmented queries through data reformulation to increase the diversity of queries. Therefore, we request ChatGPT reformulate the query without changing its original semantics. (2) *Emphasis*: We firstly specify to ChatGPT the desired quantity of generated queries. And then, we illustrate to ChatGPT using the CoSQA dataset example that queries are brief, ensuring that it generates concise queries. (3) *Caution*: To further ensure the brevity of queries, we limit the length of the augmented query by Equation 1. In the experiment, we set the α to 1.6.

$$Length_{origianl} <= Length_{aug} <= \alpha * Length_{original} \quad (1)$$

(4) *Prior Knowledge*: Due to the brevity of queries, the task is relatively simple to complete without providing prior knowledge for query reformulation. So we do not provide prior knowledge here. (5) *Task Input & Output Context*: We provide the original query as the task input and provide the text “Rewritten Queries” as the context for ChatGPT to directly return results.

TABLE I: Structure of query augmentation prompt.

Component	Content
Instruction	Given a query, your task is to reformulate the query while ensuring that its semantics remain unchanged. .
Emphasis	You must generate (15) queries. Note that in real-life scenarios, users'queries are often brief. For example, the average length of queries in CoSQA dataset is 6.6. So you must aim to generate concise queries in this task.
Caution	You must limit the length of each rewritten query to between (query_length) and 1.6 *(query_length).
Prior Knowledge	\
Task Input	Original Query: <Query>
Output Context	Rewritten Queries:

Code Augmentation Prompt: The details of the code augmentation prompt are shown in Table II. (1) *Instruction*: Similar to the query augmentation prompt, we request ChatGPT to rewrite code without changing its functionality in code enhancement. In addition, we provide a rewriting technique to guide ChatGPT in rewriting the code. (2) *Emphasis*: We firstly specify to ChatGPT the desired quantity of generated codes. And then, we ask ChatGPT to return codes following a given template so that the rewritten codes can be automatically extracted via regular expression. If the given rewriting technique is not suitable for rewriting, we allow ChatGPT to use different methods to rewrite the code to avoid returning an empty response. (3) *Prior Knowledge*: Compared to query, code is usually longer and contains more rich information, which provides more space for code rewriting. Here we propose five rewriting techniques to guide ChatGPT in efficiently rewriting code based on different levels of information. The details are follows.

- **Rename the method without changing the function names it calls internally.**
- **Rewrite the code with more meaningful variable names.**
- **Use different library functions for the code snippet.**
- **Rewrite the code with the same semantics.**
- **Simplify the code by removing unnecessary statements or tokens.**

(4) *Task Input & Output Context*: We provide the original code as the task input and provide the text "Rewritten Codes" as the context for ChatGPT to directly return results based on the given template.

3) Data Augmentation via ChatGPT: We use the ChatGPT API (gpt-3.5-turbo-0301) with default parameter settings to perform data augmentation. For query augmentation, we request ChatGPT to generate 15 augmented queries for each query based on the prompt template shown in Table I. For code augmentation, we utilize five different rewriting techniques mentioned in Section III-A2 to create five prompts based on the templates shown in Table II. We generated 15 augmented codes per original code, using each of the five prompts to generate three new codes. After receiving a response from ChatGPT, we use regular expressions to extract the generated data.

TABLE II: Structure of code augmentation prompt.

Component	Content
Instruction	Given a method-level code snippet, your job is to rewrite the code snippet based on a given rewriting technique, while ensuring that the generated code performs the same functionality as the original code.
Emphasis	You must generate (3) codes. And use "' to wrap each code based on this template : Code (number such as 1)\n"'python\n<returned code>\n"'. If current rewriting technique is not suitable for the original code, you can rewrite it using different technique, while ensuring the generated code has the same functionality as the original code.
Prior Knowledge	Rewriting Technique:<Rewriting Technique>
Task Input	Original Code: <Code>
Output Context	Rewritten Code:

As an example, Table III shows the five augmented queries generated by ChatGPT through the rewriting of the original query. We can see that ChatGPT has a good understanding of the semantics of the query and can produce high-quality and diverse queries that preserve the original semantics by replacing synonyms, changing syntax structures, and using other techniques. Overall, the generated queries are of high quality and exhibit good diversity.

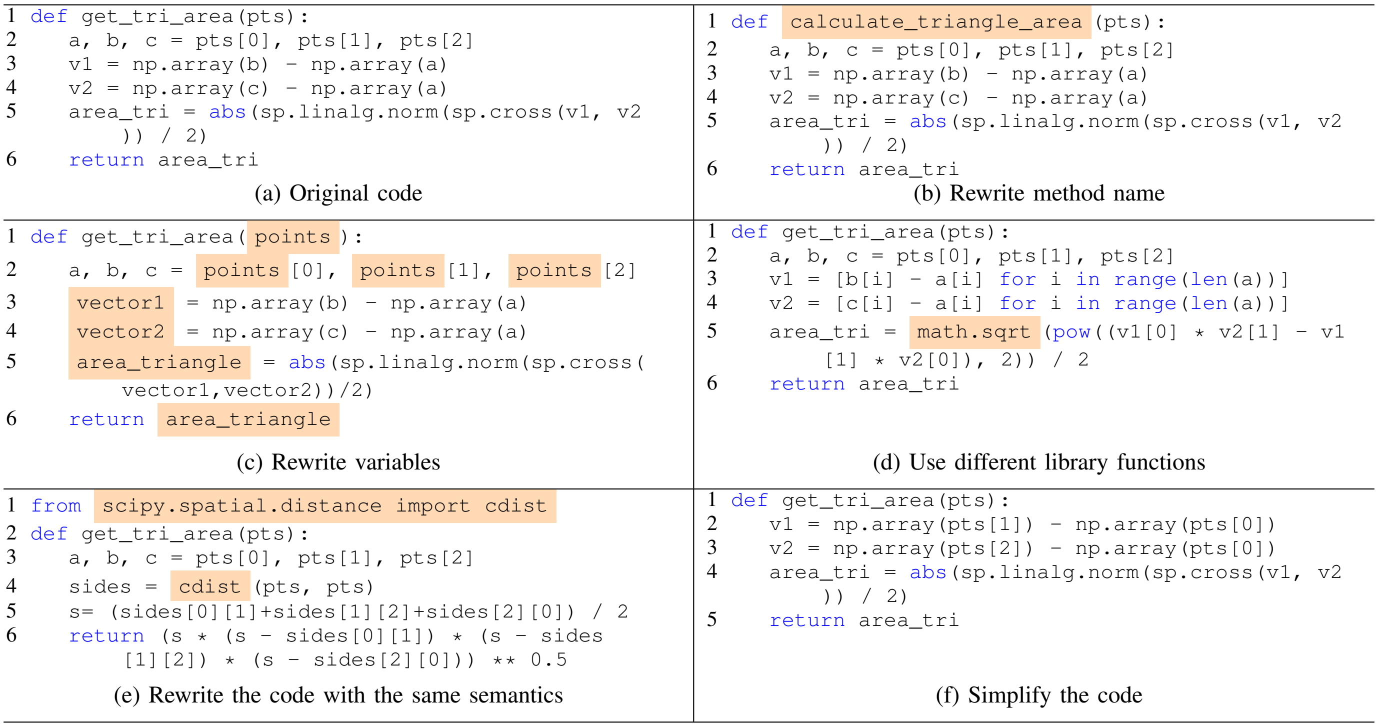
TABLE III: Augmented query samples generated by ChatGPT query augmentation.

Original Query	Math function for area of triangle python
Augmented Query 1	Calculate triangle area in Python
Augmented Query 2	Triangle area formula in Python
Augmented Query 3	Triangle area algorithm in Python
Augmented Query 4	Python area calculation for triangle
Augmented Query 5	Formula to calculate triangle area in Python

Figure 2 shows the five augmented code examples generated by ChatGPT under the guidance of the five proposed code rewriting techniques. In Figure 2b, it is shown that ChatGPT can understand the meaning of the code and generate augmented data by rewriting the function name from *get_tri_area* to *calculate_triangle_area*. Figure 2c shows that ChatGPT can understand variable abbreviations and convert them into semantically precise variable names, such as converting *pts* to *points*. In Figure 2d, ChatGPT implements the function using a different library function, *math.sqrt*. In Figure 2e, ChatGPT generates enhanced code by using a different library function, *cdst*, and a different mathematical formula while ensuring that the semantics remain unchanged. Finally, in Figure 2f, we see that ChatGPT simplifies the code snippet by removing statement 2 from the original code.

B. The Data Filtering Stage

1) Bi-encoder & Cross-encoder: In code search, we use deep learning models to score query-code pairs as the basis for their matching. There are two common architectures for code search models: bi-encoder [9]–[12], [15], [16], [18]–[20], [22], [26]–[28] and cross-encoder [13], [14], [17], [21], [23], [25],



```python
def get_tri_area(pts):
    a, b, c = pts[0], pts[1], pts[2]
    v1 = np.array(b) - np.array(a)
    v2 = np.array(c) - np.array(a)
    area_tri = abs(sp.linalg.norm(sp.cross(v1, v2)) / 2)
    return area_tri
```

(a) Original code

```python
def calculate_triangle_area(pts):
    a, b, c = pts[0], pts[1], pts[2]
    v1 = np.array(b) - np.array(a)
    v2 = np.array(c) - np.array(a)
    area_tri = abs(sp.linalg.norm(sp.cross(v1, v2)) / 2)
    return area_tri
```

(b) Rewrite method name

```python
def get_tri_area(points):
    a, b, c = points[0], points[1], points[2]
    vector1 = np.array(b) - np.array(a)
    vector2 = np.array(c) - np.array(a)
    area_triangle = abs(sp.linalg.norm(sp.cross(vector1,vector2))/2)
    return area_triangle
```

(c) Rewrite variables

```python
def get_tri_area(pts):
    a, b, c = pts[0], pts[1], pts[2]
    v1 = [b[i] - a[i] for i in range(len(a))]
    v2 = [c[i] - a[i] for i in range(len(a))]
    area_tri = math.sqrt(pow((v1[0] * v2[1] - v1[1] * v2[0]), 2)) / 2
    return area_tri
```

(d) Use different library functions

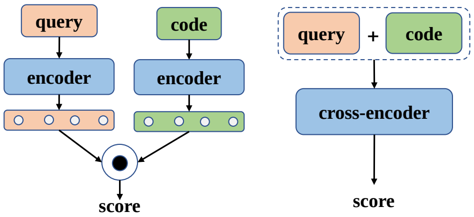
```python
from scipy.spatial.distance import cdist
def get_tri_area(pts):
    a, b, c = pts[0], pts[1], pts[2]
    sides = cdist(pts, pts)
    s= (sides[0][1]+sides[1][2]+sides[2][0]) / 2
    return (s * (s - sides[0][1]) * (s - sides[1][2]) * (s - sides[2][0])) ** 0.5
```

(e) Rewrite the code with the same semantics

```python
def get_tri_area(pts):
    v1 = np.array(pts[1]) - np.array(pts[0])
    v2 = np.array(pts[2]) - np.array(pts[0])
    area_tri = abs(sp.linalg.norm(sp.cross(v1, v2)) / 2)
    return area_tri
```

(f) Simplify the code

Fig. 2: Augmented code samples generated by ChatGPT.



(a) Bi-encoder (b) Cross-encoder

Fig. 3: The architectures of bi-encoder and cross-encoder.

[33]. We denote bi-encoder model as f_{bi} and cross-encoder model as f_{cross}. As shown in Figure 3(a), given a query-code pair $< Q, C >$, the bi-encoder model encodes the query sequence and code sequence into query vector $\vec{q}$ and code vector $\vec{c}$, respectively:

$$\vec{q} = f_{bi}(Q) \qquad \vec{c} = f_{bi}(C) \tag{2}$$

And then we calculate the cosine similarity between $\vec{q}$ and $\vec{c}$ as the matching score for the code pair:

$$Score_{bi} = sim(\vec{q}, \vec{c}), \quad sim(\vec{q}, \vec{c}) = \frac{\vec{q} \cdot \vec{c}}{\|\vec{q}\| \, \|\vec{c}\|} \tag{3}$$

For the cross-encoder model shown in Figure 3(b), we first concatenate the query sequence Q and code sequence C into a single sequence and then input it into the cross-encoder model to generate the matching score end-to-end:

$$Score_{cross} = f_{cross}([Q, C]) \tag{4}$$

The bi-encoder model typically has a faster retrieval speed than the cross-encoder model. Assuming there are m query pairs to be searched against a codebase with n code snippets, the bi-encoder model requires m+n model inferences, while the cross-encoder model requires m*n inferences due to the need to concatenate different queries and codes into input sequences. Therefore, the bi-encoder model is commonly used in practical retrieval scenarios. However, the cross-encoder model has better retrieval accuracy than the bi-encoder model. After concatenating the query and code sequences, the cross-encoder model enables token-level interaction between query and code through attention mechanisms, whereas in the bi-encoder model, the interaction between query and code is limited to vector-level. Therefore, the cross-encoder model is considered to be better at matching queries and codes [41]. In this paper, we employ the cross-encoder model with better matching capability as the filtering model to filter the augmented data.

2) Filtering Algorithm: After obtaining the augmented data, we filter the data and generate the augmented training dataset using the algorithm shown in Algorithm 1. We first train a neural model to measure the semantic relevance between a query and a code snippet, and then filter out code and query pairs with low semantic relevance scores. Specifically, in line 2, we first train a filtering model M based on cross-encoder architecture on training dataset D. In line 3-21, we generate an augmented sample for each query-code pair $< q, c >$ in training dataset D. In line 4, we initialize a list to collect the filtered code. In line 5-12, we iterate through the augmented codes and score the original query q and augmented codes using the filtering model. Then we filter out the augmented code c_{aug} with a score below the threshold, and we synthesize augmented samples by combining q and q_{aug}. Meanwhile, we add the filtered codes to $code\_list$. Similarly, we filter the queries using the same method in line 13-20. But note that in line 17-18, we generate an augmented sample by combining augmented query q_{aug} and c_{sample} sampled from $code\_list$ randomly to increase the diversity of augmented samples. In our experiments, we have manually checked the quality of the augmented data, and the results show that the augmented code can correctly answer the query. Details can be found in Appendix of replication package [42].

Algorithm 1 Filtering Algorithm

```
Input: original training dataset $D$
Input: query augmentation dictionary $dict_q$
Input: code augmentation dictionary $dict_c$,
Input: filtering threshold for augmented queries $\theta_q$
Input: filtering threshold for augmented codes $\theta_c$
Output: augmented training dataset $D_{aug}$
 1: Initialize $D_{aug} \leftarrow D$
 2: Train a filtering model $M$ on dataset $D$
 3: for $(q, c)$ in $D$ do
 4:     $code\_list \leftarrow [c]$
 5:     for $c_{aug}$ in $dict_c[c]$ do
 6:         $input\_sequence = concatenate(q, c_{aug})$
 7:         $score = M(input\_sequence)$
 8:         if $score \geq \theta_c$ then
 9:             add $(q, c_{aug})$ to $D_{aug}$
10:             add $c_{aug}$ to $code\_list$
11:         end if
12:     end for
13:     for $q_{aug}$ in $dict_q[q]$ do
14:         $input\_sequence = concatenate(q_{aug}, c)$
15:         $score = M(input\_sequence)$
16:         if $score \geq \theta_q$ then
17:             $c_{sample} = random.choice(code\_list)$
18:             add $(q_{aug}, c_{sample})$ to $D_{aug}$
19:         end if
20:     end for
21: end for
22: return $D_{aug}$
```

C. Model Training

After the data filtering stage, we finetune the bi-encoder model on augmented dataset D_{aug}. Following previous studies related to code search [24]–[29], we finetune model on training dataset by this loss function:

$$L = -\frac{1}{bs}\sum_{i=1}^{bs}\left[\log \frac{(e^{sim(\vec{q_i},\vec{c_i})/\tau})}{\sum_{j=1}^{bs} e^{(sim(\vec{q_i},\vec{c_j})/\tau)}}\right] \tag{5}$$

where bs denotes the batch size during model training. And $\vec{q_i}$ and $\vec{c_i}$ are vector representations generated from query q and code c using bi-encoder model. The τ is a hyperparameter. After finetuning the model on the augmented data, we conduct evaluations on the test dataset three times with different random seeds and report the average MRR as the result of the evaluations.

IV. Experimental Design

A. Evaluated Dataset

We evaluate our approach on a high-quality dataset CoSQA [33]. It contains 20,604 web queries collected from the Microsoft Bing search engine and 6,267 Python functions from GitHub. Each instance in CosQA contains a pair of a web query and a code snippet, where one code snippet could be paired with multiple queries. Following the original settings in Guo et al. [27], the training, validation, and testing sets contain 19604, 500, and 500 instances, respectively, and the codebase for code retrieval contains 6,267 code snippets for evaluation. Table IV provides detailed information about the dataset. Additionally, due to the maximum input token length constraint of ChatGPT, we need to ensure that the token lengths of code and query in the CoSQA dataset do not exceed 4096. We conducted a data statistics on the CoSQA dataset and found that the maximum token numbers of codes and queries in the dataset are 1806 and 21, respectively, which do not exceed the token limit (4096) of ChatGPT. More details can be found in Appendix of replication package [42].

TABLE IV: Details of CoSQA dataset.

	# of instances	**# of queries**	**# of codes**
Train	19604	19604	6127
Validation	500	500	6267
Test	500	500	6267

B. Baselines

We compare ChatDANCE with two previous data augmentation methods, namely **QRA** and **NatGen**, and a strong baseline named Unixcoder [27] in code search to evaluate the effectiveness of ChatDANCE:

1) **QRA** (Query-Rewritten Augmentation) [33] assumes that queries with minor modifications share the same semantics as the original query. Based on this assumption, QRA performs query augmentation by rewriting queries in three ways: deleting a word randomly, copying a word randomly, and switching the position of two words randomly. In our experiments, we use all three ways to perform Query-Rewritten Augmentation.
2) **NatGen** (De-Naturalizing Source Code) [34] performs code augmentation by rewriting code based on six semantic-preserving transformations: (1) Loop Transformation, (2) DeadCode Injection, (3) Operand Swap, (4) Block Swap, (5) Variable Renaming, (6) Confusing Code Insertion. We use the first five transformations as we find the last transformation is ineffective for the codes in the training set. Similar to code augmentation mentioned in Section III-A3, we use each transformation rule three times and generate 15 augmented codes in total.

C. Experimental Settings

During the data augmentation stage, we use the ChatGPT API (GPT-3.5-Turbo-0301) [43] provided by OpenAI to generate data with default parameter settings. In the subsequent stages, we employed the state-of-the-art model UniXcoder [27] (Table V) as the backbone model in our experiments. It is a Transformer-based architecture with 12 layers, 768 dimensional hidden states, and 12 attention heads. During the data filtering stage, we take cross-encoder-based Unixcoder as the filtering model and train it using the AdamW optimizer [44] with a learning rate of 8e-5 and weight decay of 0.01. We empirically use a filtering score threshold of 0.75 for code and 0.95 for queries during the filtering process. During the model training stage, we treat UniXcoder as a bi-encoder model and use AdamW to optimize it with a learning rate of 3e-5 and weight decay of 0.001. We conduct experiments three times with different random seeds and report the mean values. All experiments are conducted on a machine with 216 GB main memory and Tesla A100 80GB GPU.

TABLE V: Performances of different models on CoSQA.

	Model	**MRR**
Encoder-Only	RoBERTa	60.3
	CodeBERT	65.7
	GraphCodeBERT	68.4
Encoder-Decoder	PLBART	65.0
	CodeT5-base	67.8
Unified	UniXcoder	70.1

D. Evaluation Metrics

Following previous studies [9]–[31], [33], we choose two commonly used metrics, namely Mean Reciprocal Rank (MRR) and R@1, to evaluate the performance of the code search models. MRR is the average of reciprocal ranks of the ground truth code snippets for the given queries Q. $R@1$ calculates the proportion of queries for which the correct code snippets are ranked first in the returned ranked lists. MRR and R@1 are defined as:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i} \tag{6}$$

$$R@1 = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \delta(rank_i = 1) \tag{7}$$

where $|Q|$ is the number of queries, $rank_i$ is the rank of the correct code for query i and δ is an indicator function that returns 1 if the input condition is true and 0 otherwise.

V. Evaluation

A. RQ1: What Is the Effectiveness of Our Approach?

Overall results. We evaluate the effectiveness of ChatDANCE by comparing it with baselines on the CoSQA [33] dataset with two common metrics MRR and R@1. The experiment results are shown in Table VI. From Table VI, we can see that our model outperforms all baselines. Specifically, our approach improved the base model (UniXcoder) by 7.0% in MRR and 13.2% in R@1. Furthermore, our approach has smaller standard deviations in both MRR and R@1 metrics compared to the baselines, indicating better stability. Overall, our approach performs the best among all models.

TABLE VI: Performance of different data augmentation approaches. Standard deviations are shown in parentheses.

Model	**MRR**	**R@1**
UniXcoder	70.2 (±0.35)	56.7 (±0.61)
QRA	71.3 (±0.84)	58.6 (±1.04)
NatGen	72.3 (±1.60)	61.1 (±2.08)
ChatDANCE	**75.1 (±0.62)** (↑7.0%)	**64.2 (±0.87)** (↑13.2%)

Case study. Figure 4 presents the top-1 code snippets returned by QRA, NatGen, and ChatDANCE for the query "how to remove blank lines from a text file in python". We can see that ChatDANCE returns the correct code snippet, which reads the file content and removes blank lines. In contrast, the code snippet returned by QRA and NatGen is incorrect, which can remove blank lines but fails to operate on a file.

Listing (1) The top-1 code returned by NatGen and QRA

```python
def lines(input):
    for raw_line in input:
        line = raw_line.strip()
        if line and not line.startswith('#'):
           yield strip_comments(line)
```

Listing (2) The top-1 code returned by ChatDANCE

```python
def get_stripped_file_lines(filename):
    try:
        lines = open(filename).readlines()
    except FileNotFoundError:
        fatal("Could not open file: {!r}".format
            (filename))
    return [line.strip() for line in lines]
```

Fig. 4: The top-1 code returned by QRA, NatGen, and ChatDANCE for the query "how to remove blank lines from a text file in python".

Summary: ChatDANCE significantly outperforms baselines on code search and the case study intuitively demonstrates the superiority of ChatDANCE.

B. RQ2: How Much Do Different Components Contribute?

As described in Section III, ChatDANCE contains 3 main components: query augmentation, code augmentation, and data filtering. We conduct an ablation study by removing each component at a time. When examining the impact of data filtering component, we explore three experimental settings: (1) no filtering on queries, (2) no filtering on code, and (3) no filtering on both. The results are shown in Table VII.

TABLE VII: Ablation study results of ChatDANCE.

Model	MRR	R@1
ChatDANCE	**75.1 (±0.62)**	**64.2 (±0.87)**
w/o query aug	74.6 (±0.66)	63.3 (±1.41)
w/o code aug	72.2 (±0.65)	59.5 (±2.00)
w/o query filtering	73.4 (±0.40)	61.9 (±0.50)
w/o code filtering	73.5 (±0.30)	61.5 (±0.75)
w/o any filtering	72.9 (±2.15)	61.3 (±4.02)

From Table VII, we can see that the performance of ChatDANCE decreases when any individual component is removed. This indicates that each component of ChatDANCE plays an important role in overall performance improvement. Furthermore, we observe that data filtering is critical to the model's performance and stability. Without data filtering, they would be significantly affected. The filtering mechanism can effectively eliminate noise within the data, resulting in high-quality data that is essential for improving model performance.

Summary: The ablation experiments demonstrate the effectiveness of each component of ChatDANCE.

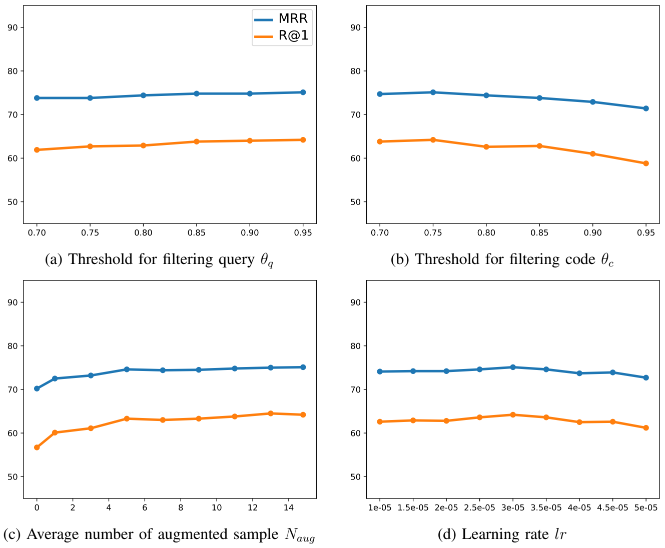
C. RQ3: What Is the Impact of Different Hyperparameters?

We investigate the impact of hyper-parameters including query filtering threshold θ_q, code filtering threshold θ_c, the average number of augmentations per sample N_{aug}, and learning rate lr. We conduct the experiments within ranges surrounding the default values, and the results are shown in Figure 5. The results show that the performance preserves stable as θ_q varies from 0.7 to 0.95. Similarly, the results show that the performance remains stable when θ_c varies from 0.7 to 0.9. However, we observe a sharp decline in performance when θ_c changes from 0.9 to 0.95. This is because when θ_c is set to 0.95, a large number of code augmentation samples are filtered out, leading to a significant decrease in the total number of samples and thus a decline in performance. In Figure 5c, we find that the performance improves significantly as the average number of augmented samples per original sample increases from 0 to 5. Beyond this point, the performance improvement gradually stabilizes. Finally, from Figure 5d, we observe that ChatDANCE is stable when learning rate varies from 1e-5 to 5e-5.

Summary: Overall, ChatDANCE performs stably across a range of hyper-parameter values ($0.7 \leq \theta_c \leq 0.9$, $0.7 \leq \theta_q \leq 0.95$, $N_{aug} \geq 5$, $1e^{-5} \leq lr \leq 5e^{-5}$).

D. RQ4: Why Does Our Approach Work?

In general, the primary advantage of our approach is generating a substantial volume of high-quality training data applicable to both queries and code snippets, which enables the model to achieve superior performance. By augmenting queries, we can introduce richer syntactic structures and more expressive forms, enhancing the model's robustness and generalization ability. Additionally, by augmenting codes, we can enable the model to learn more complex syntactic structures and semantic information in code snippets, improving the model's understanding of code snippets. This method not only amplifies the training data quantity but also elevates its quality, leading to a more effective and robust model. Next, we conduct quantitative and qualitative analysis to investigate why ChatDANCE works well in detail.



(a) Threshold for filtering query θ_q

(b) Threshold for filtering code θ_c

(c) Average number of augmented sample N_{aug}

(d) Learning rate lr

Fig. 5: The impact of different hyperparameters.

1) Quantitative Analysis: We investigate why ChatDANCE works by studying the distribution of data representations learned by models. We use ℓ_{align} and $\ell_{uniformity}$ [45] metrics to evaluate the quality of the representations learned by models, which are widely used in contrastive learning [45]–[48]. The ℓ_{align} and $\ell_{uniformity}$ metrics are defined as follows in Equation 8:

$$\begin{aligned} \ell_{align} &= \underset{(x,y)\sim D_{pair}}{\mathbb{E}} [\|f(x) - f(y)\|_2^{\alpha}], \quad \alpha{>}0 \\ \ell_{uniformity} &= log \underset{(x,y)\sim D}{\mathbb{E}} e^{-t\|f(x)-f(y)\|_2^2}, \quad t{>}0 \end{aligned} \tag{8}$$

where $(x, y) \sim D_{pair}$ means that x and y (such as query and code) are paired, and $(x, y) \sim D$ means that x and y are independently and identically distributed. $f(x)$ and $f(y)$ represent the representations learned by the model, and $\|f(x) - f(y)\|_2$ represents the 2-norm distance between the representations. The hyperparameters α and t are set to 2 in our experiments. The $\ell_{alignment}$ is defined as the expected distance between paired representations, which measures the degree of matching between paired representations. From Equation 8, we know that the smaller the distance between paired representations, the closer the alignment loss is to 0. In the extreme case, if the distance between all paired representations is 0, the alignment loss is 0. The $\ell_{uniformity}$ measures the uniformity of the distribution of representations. From Equation 8, we know that the closer the distribution of representations is to uniformity, the closer the value of uniformity loss is to negative infinity. According to [45], [46], better alignment and

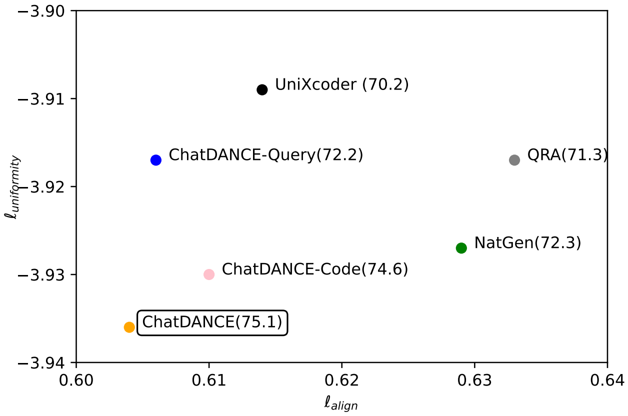
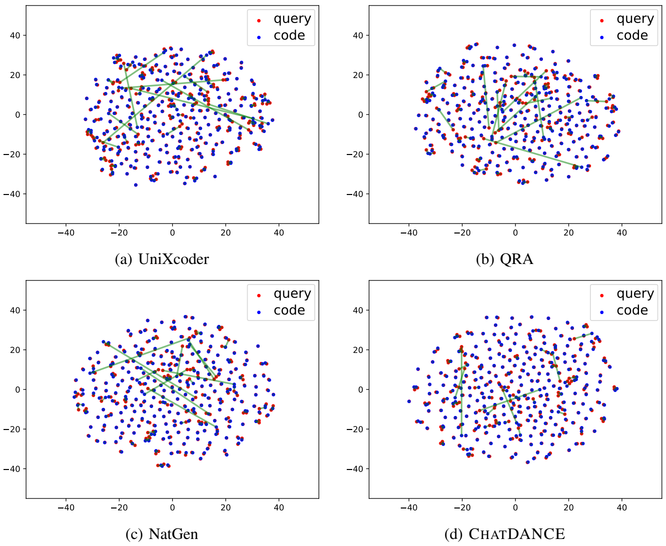


Fig. 6: ℓ_{align}-$\ell_{uniformity}$ plot of different models. ChatDANCE-Query and ChatDANCE-Code represent performing data augmentation on queries or codes, respectively.

uniformity can enable the model to have better performance and generalization. In our experiments, we use $\ell_{alignment}$ and $\ell_{uniformity}$ to measure these two properties. The lower the values of two losses, the better the alignment and uniformity of the learned representations.

The ℓ_{align}-$\ell_{uniformity}$ plot shown in Figure 6 reveals several observations. Firstly, our approach reduces both $\ell_{alignment}$ and $\ell_{uniformity}$ compared to the two baselines. This implies that our approach can enhance both the alignment and uniformity of the learned representations, resulting in superior performance and generalization. In contrast, the baselines only reduce the uniformity loss but increase the alignment loss, indicating a deterioration in the alignment of the learned representations. Therefore, we believe that better alignment and uniformity are crucial factors in achieving superior performance and generalization in our approach. Secondly, our approach can simultaneously improve the alignment and uniformity of the learned representations, even when data augmentation is applied only to either code or query. In contrast, the baselines fail to achieve this. We attribute this success to our approach's ability to generate a large amount of high-quality and diverse data, thereby improving uniformity while ensuring the quality and diversity of the generated data. This allows the model to better comprehend the semantic meaning of both query and code, ultimately improving the alignment of the learned representations.

2) Qualitative Analysis: We visualize the distribution of representations learned by four models shown in Figure 7 to intuitively explore why our approach works. First, we sample 300 query-code pairs from the test set of CosQA and obtain their representations in the high-dimensional vector space by feeding them into the model. Then, we use t-SNE [49] to perform dimensionality reduction on the representations and visualize their distribution. The experimental results are shown in Figure 7. We visualized the distribution of representations learned by four models: UniXcoder, QRA, NatGen, and ChatDANCE, which are shown in Figure 7a, 7b, 7c, and 7d, respectively. The red dots represent the code, and the blue dots represent the query. The lines between the red and blue dots indicate the distance between the code and the query.



(a) UniXcoder

(b) QRA

(c) NatGen

(d) ChatDANCE

Fig. 7: t-SNE visualization of representations of queries and code snippets. The red and blue dots represent queries and code, respectively. The green line connecting red and blue dot shows the distance between query-code pairs.

From the experimental results, we can observe that: (1) In Figure 7a, 7b, 7c, and 7d, most of the green lines are very short, indicating that for most query-code pairs, the model is able to align their representations in the high-dimensional vector space to close locations. (2) In Figure 7, compared to the other three figures, Figure 7d has the fewest long green lines. This suggests that our approach can enable the model to have better alignment compared to the other methods, as there is the fewest number of representations with long distances. Overall, the visualization results intuitively demonstrate that our approach can enable the model to learn better representations compared to the baselines by improving the alignment and uniformity of the learned representations.

Summary: ChatDANCE learns a more uniform distribution of representations and effectively aligns the learned representations of paired code snippets and queries.

VI. Discussion

A. Discussion on Using LLMs

Generally, LLMs such as ChatGPT [50] are commonly used for generation tasks. However, when used for code search, a smaller code search model is more efficient. In our pre-study, we find that ChatGPT takes an average of 5 seconds to generate a single code snippet, while a code search model retrieves code with an average time of 0.023 seconds. As a result, code search models demonstrate higher efficiency compared to directly using LLMs. Additionally, when the codebase is evolving, training ChatGPT to update its knowledge is costly, whereas training dedicated code search models on the updated codebase is relatively inexpensive. In general, using smaller models for code search is more efficient and cost-effective compared to using LLMs directly.

B. Metric Choice

In our experiments, we choose MRR and R@1 as the evaluation metrics instead of R@5 and R@10, even though MRR and R@k (where k=1, 5, 10) are commonly used metrics in code search. This is because our experiments employed a powerful baseline, UniXcoder, which achieved MRR scores greater than 0.6. This suggests that for the majority of queries, the correct code is ranked within the top-3 results. Consequently, R@5 and R@10 metrics are not very informative in this case to reflect performance improvements. Therefore, we focus on using MRR and R@1 as the primary evaluation metrics. However, for completeness and comparison purposes, we also provide the evaluation results under R@5 and R@10 in Appendix of replication package [42].

C. Threats To Validity

We identify the following threats to our approach:

LLM Choice. In our experiments, we only use the GPT-3.5-Turbo-0301 model [43] to perform data augmentation. This is because OpenAI only made this API available during the experiment. Additionally, other large open-source models such as Llama [39] were not open-sourced at the time, so we did not explore the effectiveness of our method using more LLMs. In the future, we will combine our method with more LLMs to comprehensively explore its effectiveness.

Dataset Choice. As shown in Section IV-A, we use the CoSQA [33] dataset instead of other datasets such as CodeSearchNet [51] in our experiments. This is because CoSQA is collected from real-world queries and with manually checked data quality, which is advantageous for us to explore the effectiveness of our method on both query and code enhancement. Additionally, CodeSearchNet has a much larger dataset size compared to CoSQA, and considering that our method is time-consuming, performing data augmentation on CodeSearchNet would require a significant amount of time. In the future, we will explore the effectiveness of our method on more datasets.

VII. Conclusion

In this paper, we explore the effectiveness of ChatGPT-based data augmentation in the code search task and demonstrate its effectiveness through extensive experiments. We propose a novel and efficient data augmentation method called ChatDANCE, which generates a large amount of high-quality data by rewriting both code and queries using ChatGPT with carefully designed guidance. Additionally, we introduced a filtering mechanism that removes low-quality data from the augmented data, further enhancing the quality of the augmented data. Our experimental results demonstrate that our method can effectively improve the performance of code search models and significantly outperform the baselines. We believe that our augmentation method could be adapted to other code intelligence tasks such as code summarization and different programming languages. Furthermore, our exploration on prompt engineering for code search may inspire researchers to effectively leverage LLMs in solving various software engineering tasks. Replication package is anonymously available at https://anonymous.4open.science/r/ChatDANCE.

Acknowledgement

The work described in this paper was supported by the National Natural Science Foundation of China (No.62002352) and the National Natural Science Foundation of China (No.62032025).

References

[1] “Github,” https://github.com/.

[2] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, “A survey of machine learning for big code and naturalness,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[3] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, “An examination of software engineering work practices,” in *CASCON First Decade High Impact Papers*, 2010, pp. 174–188.

[4] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li, “Query expansion based on crowd knowledge for code search,” *IEEE Transactions on Services Computing*, vol. 9, no. 5, pp. 771–783, 2016.

[5] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, “Portfolio: finding relevant functions and their usage,” in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 111–120.

[6] M. Lu, X. Sun, S. Wang, D. Lo, and Y. Duan, “Query expansion via wordnet for effective code search,” in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 545–549.

[7] F. Lv, H. Zhang, J.-g. Lou, S. Wang, D. Zhang, and J. Zhao, “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, 2015, pp. 260–270.

[8] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi, “Sourcerer: mining and searching internet-scale software repositories,” *Data Mining and Knowledge Discovery*, vol. 18, pp. 300–336, 2009.

[9] X. Gu, H. Zhang, and S. Kim, “Deep code search,” in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 933–944.

[10] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, “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*, 2019, pp. 964–974.

[11] X. Ling, L. Wu, S. Wang, G. Pan, T. Ma, F. Xu, A. X. Liu, C. Wu, and S. Ji, “Deep graph matching and searching for semantic code retrieval,” *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 15, no. 5, pp. 1–21, 2021.

[12] J. Shuai, L. Xu, C. Liu, M. Yan, X. Xia, and Y. Lei, “Improving code search with co-attentive representation learning,” in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 196–207.

[13] L. Du, X. Shi, Y. Wang, E. Shi, S. Han, and D. Zhang, “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*, 2021, pp. 2994–2998.

[14] W. Li, H. Qin, S. Yan, B. Shen, and Y. Chen, “Learning code-query interaction for enhancing code searches,” in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 115–126.

[15] W. Ye, R. Xie, J. Zhang, T. Hu, X. Wang, and S. Zhang, “Leveraging code generation to improve code retrieval and summarization via dual learning,” in *Proceedings of The Web Conference 2020*, 2020, pp. 2309–2319.

[16] Y. Ma, Y. Yu, S. Li, Z. Jia, J. Ma, R. Xu, W. Dong, and X. Liao, "Mulcs: Towards a unified deep representation for multilingual code search."

[17] Q. Zhu, Z. Sun, X. Liang, Y. Xiong, and L. Zhang, "Ocor: an overlapping-aware code retriever," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 883–894.

[18] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, and P. Yu, "Multi-modal attention network learning for semantic source code retrieval," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 13–25.

[19] R. Haldar, L. Wu, J. Xiong, and J. Hockenmaier, "A multi-perspective architecture for semantic code search," in *Annual Meeting of the Association for Computational Linguistics*, 2020.

[20] J. Gu, Z. Chen, and M. Monperrus, "Multimodal representation for neural code search," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 483–494.

[21] C. Ling, Z. Lin, Y. Zou, and B. Xie, "Adaptive deep code search," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 48–59.

[22] W. Sun, C. Fang, Y. Chen, G. Tao, T. Han, and Q. Zhang, "Code search based on context-aware code translation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 388–400.

[23] Y. Shi, Y. Yin, Z. Wang, D. Lo, T. Zhang, X. Xia, Y. Zhao, and B. Xu, "How to better utilize code graphs in semantic code search?" in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 722–733.

[24] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708.

[25] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[26] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, L. Shujie, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," in *International Conference on Learning Representations*.

[27] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "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)*, 2022, pp. 7212–7225.

[28] E. Shi, Y. Wang, W. Gu, L. Du, H. Zhang, S. Han, D. Zhang, and H. Sun, "Cocosoda: Effective contrastive learning for code search," 2023.

[29] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "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*, 2021, pp. 2655–2668.

[30] N. D. Bui, Y. Yu, and L. Jiang, "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*, 2021, pp. 511–521.

[31] P. Jain and A. Jain, "Contrastive code representation learning," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021.

[32] Z. Dong, Q. Hu, Y. Guo, Z. Zhang, M. Cordy, M. Papadakis, Y. L. Traon, and J. Zhao, "Boosting source code learning with data augmentation: An empirical study," *arXiv preprint arXiv:2303.06808*, 2023.

[33] J. Huang, D. Tang, L. Shou, M. Gong, K. Xu, D. Jiang, M. Zhou, and N. Duan, "Cosqa: 20,000+ web queries for code search and question answering," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 2021, pp. 5690–5700.

[34] S. Chakraborty, T. Ahmed, Y. Ding, P. T. Devanbu, and B. Ray, "Natgen: generative pre-training by "naturalizing" source code," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 18–30.

[35] D. A. Van Dyk and X.-L. Meng, "The art of data augmentation," *Journal of Computational and Graphical Statistics*, vol. 10, no. 1, pp. 1–50, 2001.

[36] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, "A survey of large language models," *arXiv preprint arXiv:2303.18223*, 2023.

[37] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[38] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311*, 2022.

[39] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[40] S. Mishra, D. Khashabi, C. Baral, and H. Hajishirzi, "Cross-task generalization via natural language crowdsourcing instructions," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2022, pp. 3470–3487.

[41] F. Hu, Y. Wang, L. Du, X. Li, H. Zhang, S. Han, and D. Zhang, "Revisiting code search in a two-stage paradigm," in *Proceedings of the Sixteenth ACM International Conference on Web Search and Data Mining*, 2023, pp. 994–1002.

[42] Anonym, "Anonymous replication package," https://anonymous.4open.science/r/ChatDANCE/README.md, 2023.

[43] "Openai platform - chat api documentation," https://platform.openai.com/docs/guides/chat, 2023.

[44] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.

[45] T. Wang and P. Isola, "Understanding contrastive representation learning through alignment and uniformity on the hypersphere," in *International Conference on Machine Learning*. PMLR, 2020, pp. 9929–9939.

[46] T. Gao, X. Yao, and D. Chen, "Simcse: Simple contrastive learning of sentence embeddings," in *2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*. Association for Computational Linguistics (ACL), 2021, pp. 6894–6910.

[47] Y. Meng, C. Xiong, P. Bajaj, S. Tiwary, P. Bennett, J. Han, and X. Song, "Coco-lm: Correcting and contrasting text sequences for language model pretraining," in *35th Conference on Neural Information Processing Systems, NeurIPS 2021*. Neural information processing systems foundation, 2021, pp. 23 102–23 114.

[48] F. Wang and H. Liu, "Understanding the behaviour of contrastive loss," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2021, pp. 2495–2504.

[49] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne." *Journal of machine learning research*, vol. 9, no. 11, 2008.

[50] Chatgpt. OpenAI. [Online]. Available: https://chat.openai.com/

[51] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.