Rewriting the Code: A Simple Method for Large Language Model Augmented Code Search

Haochen Li Xin Zhou* Zhiqi Shen
Nanyang Technological University, Singapore
{haochen003, xin.zhou, zqshen}@ntu.edu.sg

Abstract

In code search, the Generation-Augmented Retrieval (GAR) framework, which generates exemplar code snippets to augment queries, has emerged as a promising strategy to address the principal challenge of modality misalignment between code snippets and natural language queries, particularly with the demonstrated code generation capabilities of Large Language Models (LLMs). Nevertheless, our preliminary investigations indicate that the improvements conferred by such an LLM-augmented framework are somewhat constrained. This limitation could potentially be ascribed to the fact that the generated codes, albeit functionally accurate, frequently display a pronounced stylistic deviation from the ground truth code in the codebase. In this paper, we extend the foundational GAR framework and propose a simple yet effective method that additionally Rewrites the Code (ReCo) within the codebase for style normalization. Experimental results demonstrate that ReCo significantly boosts retrieval accuracy across sparse (up to 35.7%), zero-shot dense (up to 27.6%), and fine-tuned dense (up to 23.6%) retrieval settings in diverse search scenarios. To further elucidate the advantages of ReCo and stimulate research in code style normalization, we introduce Code Style Similarity, the first metric tailored to quantify stylistic similarities in code. Notably, our empirical findings reveal the inadequacy of existing metrics in capturing stylistic nuances. The source code and data are available at https://github.com/Alex-HaochenLi/ReCo.

1 Introduction

Code search, aimed at retrieving the most semantically relevant code snippets from a codebase according to a specified natural language query, is a common activity that plays an important role in software development (Nie et al., 2016; Shuai et al., 2020). Retrieving and reusing analogous code fragments from large-scale codebases like GitHub can enhance productivity significantly.

Passage Retrieval

Query:
How was the COVID-19 pandemic impacted mental health?

True Passage:
...two studies investigating COVID-19 patients ... significantly higher level of depressive...

Generated Reference:
...depression and anxiety had increased by 20% since the start of the pandemic...

Code Search

Query:
Write a function to get the frequency of the elements in a list.

True Code:
```
import collections
def freq_count(list1):
    freq_count = collections.Counter(list1)
    return freq_count
```

Generated Exemplar Code:
```
def count_frequency(my_list):
    frequency = {}
    for element in my_list:
        if element not in frequency:
            frequency[element] = 0
        frequency[element] += 1
    return frequency
```

Figure 1: Comparison of GAR between passage retrieval and code search. In passage retrieval, the truth (yellow) is included in the generated content. In code search, despite the generated exemplar code satisfies the description of the query, it exhibits noticeable dissimilarity to the true code.

Despite both being sequences of words, matching code queries and natural language queries is challenging as they share few grammatical rules, causing them to fall into two distinct modalities. This grammatical distinction results in limited word overlap, significantly hampering the application of sparse retrieval systems in code search. On the other hand, in dense retrieval systems, the alignment of query and code representations during the training phase assists in alleviating the challenge

*Corresponding author

Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 1371–1389

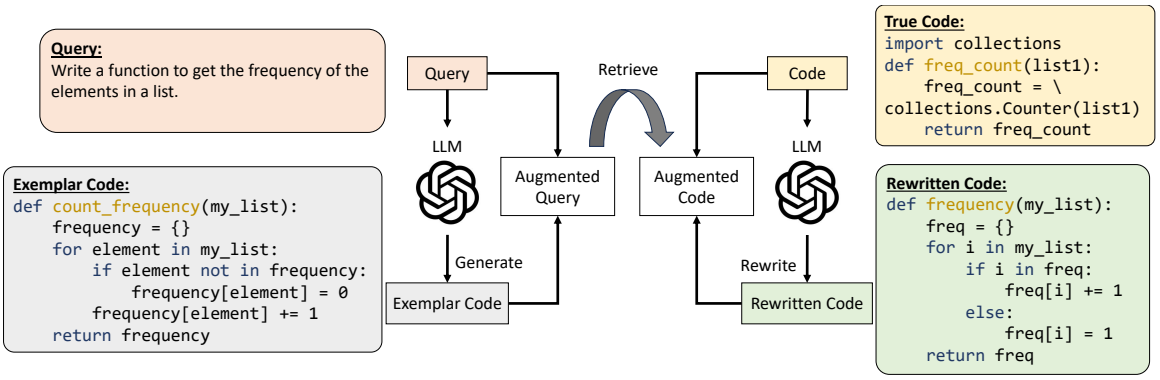

Figure 2: An illustration of the ReCo method. It initially prompts LLMs to generate exemplar codes based on the search query. Subsequently, the original query and these exemplar codes are synthesized to formulate an augmented query. Analogously, the rewritten codes, produced by the LLMs, are merged with the original code, thereby creating a candidate for retrieval. The example delineated in this figure aligns with the one depicted in Fig. 1.

(Li et al., 2023). As a result, these systems are capable of encapsulating potential semantic correlations between terminologies employed in programming languages and those in natural languages. However, this potential association becomes challenging to capture if two terminologies rarely manifest together within a query-code pair.

To bridge this gap, one possible solution is to transform the data from one modality to the other. This could involve either generating exemplar codes based on the query or summarizing the functionality of codes in the codebase. Given that natural language queries in code search are often short and ambiguous (Mao et al., 2023; Rahman and Roy, 2021), our research concentrates on the former solution, referred as Generation-Augmented Retrieval (GAR) (Mao et al., 2020). GAR has demonstrated competitive performance in question answering and passage retrieval. In these NLP tasks, a language model is adopted to generate references based on the query to augment it. Similarly, we could use a language model to generate exemplar code snippets that realize the functionalities described in the query. Then the query and exemplar codes are combined to be fed into the retrieval system. With many LLMs demonstrating great intelligence in precisely writing codes (Touvron et al., 2023a,b; OpenAI, 2023b; Zan et al., 2022), performing GAR with LLMs becomes a promising approach for code search.

However, from our preliminary studies, the improvement in performance brought by GAR using LLMs is limited, especially with the high computational cost of LLMs. We argue that answer format influences the performance of GAR on question answering and code search. In question answering, the correct answer to the question is often unique and can be expressed in limited forms. The generated contents from LLMs, if correct, are usually in the exact same form as the answer. As highlighted in Fig. 1, the matching word "depressive" appears in the reference. On the other hand, code snippets with the same functionality can have diverse formulations, which lowers the chance of matching the code in the codebase, and thus leads to minor improvement of GAR in code search. As shown in Fig. 1, the true code uses Python built-in function `Counter` to count the number of elements in a list, while the exemplar code snippet does it manually.

To address the mismatch of the generated and ground truth code snippets, we build upon GAR and propose a simple yet effective framework that additionally Rewrites and the Code (ReCo) in the codebase. As shown in Fig. 2, after rewriting, the style of codes in the codebase are normalized by LLMs to align with the exemplar code, thereby facilitating the retrieval. We evaluate ReCo on several code search models across various search scenarios, including coding challenge competence, online programming community, and general programming problems in Python and Java. Experimental results show that ReCo could significantly boost the performance of sparse retrieval systems (up to 35.7%) and dense retrieval systems in both zero-shot (up to 27.6%) and fine-tuning (up to 23.6%) settings.

Furthermore, we propose a novel evaluation metric, dubbed Code Style Similarity, to quantitatively measure the disparity in code style. Our metric validates ReCo's capability in aligning the style of code

within the codebase with that of code generated by LLMs. Conventional metrics like BLEU (Papineni et al., 2002) and CodeBLEU (Ren et al., 2020) are deemed less appropriate as they calculate similarity based on exact-matched tokens of the given two code snippets. In contrast, Code Style Similarity evaluates style from three distinct perspectives: variable naming, API invocation, and code structure, based on edit distance (Ristad and Yianilos, 1998). Our experiments show that Code Style Similarity exhibits superior explanatory power than existing metrics in measuring the style deviation of code from the dataset and that generated from LLM.

2 Related Works

Code Search Models The development of code search models could be split into three stages. Traditional methods, also denoted as sparse retrieval, employ information retrieval techniques to match words between queries and codes (Hill et al., 2011; Yang and Huang, 2017; Satter and Sakib, 2016). As we mentioned before, since programming languages and natural languages share few grammatical rules, these methods often suffer from vocabulary mismatch problems (McMillan et al., 2011). Then, neural models became popular (Gu et al., 2021; Cambronero et al., 2019; Gu et al., 2018; Husain et al., 2019). They all employ a framework where queries and codes are encoded by neural encoders separately into a joint representation space.

Recently, transformer-based pre-trained models significantly outperformed previous methods, since they can be trained on large-scale unlabelled corpus with self-supervised pre-training tasks. Many novel pre-training tasks are proposed to let models have a better general understanding of codes (Guo et al., 2021; Li et al., 2022b,c; Shi et al., 2022). For instance, CodeBERT (Feng et al., 2020) utilizes masked language modeling and replaced token detection. CodeT5 (Wang et al., 2021) focuses on generative tasks through bimodal dual generation. UniXcoder (Guo et al., 2022) integrates the aforementioned generative and understanding pre-training tasks. CodeT5+ (Wang et al., 2023b) employs the same architecture as CodeT5 and pretrains it with span denoising, causal language modeling, contrastive learning, and text-code matching from both unimodal code data and bimodal code-text data.

Large Language Models As the model parameters and size of training corpora of those transformer-based pre-trained models scale up to billions, they appear to demonstrate remarkable intelligence in understanding and generating codes. As a milestone, Codex (Chen et al., 2021) with 12 billion parameters indicates the beginning of the Code LLM era. Meanwhile, there are a number of powerful Code LLMs proposed (Zan et al., 2022), though most of them are not publicly accessible. Recently, ignited by OpenAI's ChatGPT (OpenAI, 2023a), a bunch of excellent open-sourced models also contribute to the thriving of Code LLMs (Roziere et al., 2023; Nijkamp et al., 2022; Luo et al., 2023). Among them, Code LLaMA (Roziere et al., 2023) has attracted significant attention because it is a collection of efficient Code LLMs ranging from 7B to 34B parameters. At the same time, some LLMs that are not specifically trained for code exhibit surprising abilities in code intelligence as well, such as GPT3.5 (OpenAI, 2023a) and LLaMA (Touvron et al., 2023a,b). This can be attributed to the inclusion of code snippets in the unlabeled training corpus.

LLMs for Retrieval While LLMs are designed for token generation, their direct application to retrieval tasks such as code search is not suitable. Indeed, there have been attempts to amalgamate the search query and all the candidates together as input, subsequently requesting the LLMs to rank the candidates within the input (Qin et al., 2023). However, the constraint on input sequence length impedes its applicability to large-scale retrieval tasks.

One indirect way is to ask LLMs to generate some references and expand the search query with them. This framework, denoted as Generation-Augmented Retrieval, has been proven effective in both question answering and passage retrieval (Mao et al., 2020; Gao et al., 2022; Wang et al., 2023a). Mao et al. (2020) is the first work to propose GAR in question answering. HyDE (Gao et al., 2022) evaluates GAR in passage retrieval under zero-shot setting. query2doc (Wang et al., 2023a) extends GAR to fine-tuning. Our research findings suggest that the Generation-Augmented Retrieval (GAR) method does not substantially enhance the efficiency of code search, primarily due to the significant stylistic difference between exemplar code and true code.

Code Generation Evaluation Metrics A suitable automatic evaluation metric is vital to the growth of code generation. It is used to measure the lexical similarity between the generated hypothetical code and the true reference code. Initially, metrics such as BLEU (Papineni et al., 2002) and ROUGE (Lin, 2004), originally designed for machine translation, were utilized in the realm of code generation. However, subsequent scholarly discourse posits that these metrics overlook the syntactic and semantic nuances inherent to code. Hence, to consider those features, CodeBLEU (Ren et al., 2020) adds terms that calculate Abstract Syntax Tree similarity and data-flow similarity. CrystalBLEU (Eghbali and Pradel, 2022) sets weights for tokens according to their frequency. They find that high-frequency tokens are often meaningless hence assigning lower weights. These metrics are widely adopted in code generation evaluation, yet they are not suitable for measuring the style difference between two codes due to shared syntactic verbosity.

3 Methodology

3.1 Preliminaries

Code search aims to retrieve code snippets that are semantically most pertinent to a specified query. Given a search query q and a code snippet c in the fixed codebase, an encoder G is used to map the query and the code to a shared representation space. We calculate the similarity between query and code by dot product, which could be formulated as:

$$sim(q, c) = \langle G(q), G(c) \rangle = \langle \mathbf{v}_q, \mathbf{v}_c \rangle, \quad (1)$$

where $\mathbf{v}_q$ and $\mathbf{v}_c$ are representation vectors of q and c, respectively. Finally, codes in the codebase are ranked according to the similarity score. Note that in a code search system, code representations $\mathbf{v}_c$ can be calculated and stored in advance.

3.2 ReCo

Building on GAR, ReCo not only generates exemplar codes based on the query but also rewrites the codes in the codebase.

Generating and Rewriting Code First, we elucidate the process of generating exemplar codes. Given a query q, we employ few-shot prompting (a.k.a in-context learning) (Brown et al., 2020) to generate an exemplar code snippet. The prompt consists of an instruction *"Please generate a Java/Python code snippet according to the given description."* and K randomly sampled query-code pairs from the training set. In this paper, we set $K = 4$. The instruction and in-context samples are denoted as GEN, enabling us to derive the exemplary code c_q as follows:

$$c_q = \text{LLM}(q, \text{GEN}). \quad (2)$$

In the procedure of rewriting the code c, we initially summarize the code into a natural language description, represented as q_{sum}. This can be achieved by changing the instruction in GEN to *"What is the main purpose of the Java/Python code snippet?"*. We denote it as SUM. Subsequently, similar to generating exemplar codes, we consider q_{sum} as the query q[1]. The entire process leading to the acquisition of the rewritten code, denoted as c_c, is as follows:

$$q_{sum} = \text{LLM}(c, \text{SUM}), \quad (3)$$

$$c_c = \text{LLM}(q_{sum}, \text{GEN}). \quad (4)$$

Detailed examples are provided in Appendix A to further elucidate the prompt.

Sparse Retrieval Query q and code c are appended with exemplar code c_q and rewritten code c_c in a sparse retrieval system, respectively. Since we could generate multiple code snippets as augmentation, to retain the original semantics of q and c, we simply repeat them for N times which is equal to the number of augmented codes. Take the query as an example, the augmented search query q^+ could be expressed as:

$$q^+ = \text{concat}(\{q\} \times N, \{c_{q1}, c_{q2}, \ldots, c_{qN}\}). \quad (5)$$

Similarly, we could get the augmented code c^+. In application, q^+ is fed to the sparse retrieval system as the search query and c^+ are candidates in the codebase.

Dense Retrieval InfoNCE loss (Van den Oord et al., 2018) is widely adopted in fine-tuning because it can pull together the representations between the query and its corresponding code while pushing away the representation of negative codes (Li et al., 2023, 2022a). During training, we take

[1]The process of rewriting is implemented via a summarize-then-generate approach, as we have observed that merely instructing LLMs to rewrite the original codes does not result in significant alterations.

other in-batch codes as negative samples for a query (Huang et al., 2021). With augmented query q^+ and augmented code c^+, InfoNCE loss $\mathcal{L}$ can be described as:

$$\mathcal{L} = -\mathbb{E}\left[\log \frac{\exp(\mathbf{v}_{qi}^+ \cdot \mathbf{v}_{ci}^+)}{\exp(\mathbf{v}_{qi}^+ \cdot \mathbf{v}_{ci}^+) + \sum_{j \neq i}^{n} \exp(\mathbf{v}_{qi}^+ \cdot \mathbf{v}_{cj}^+)}\right], \tag{6}$$

where n is the batch size, $\mathbf{v}_q^+$ and $\mathbf{v}_c^+$ are augmented representations of q^+ and c^+, respectively.

For augmented representations, we calculate the expectation of all the generated content according to the chain rule. Take the exemplar code as an example, we have:

$$\mathbb{E}[\mathbf{v}_{cq}] = \mathbb{E}[G(c_q)] = \mathbb{E}[G(\mathrm{LLM}(q, \textsc{gen}))]. \tag{7}$$

Here we assume the distribution of $\mathbf{v}_{cq}$ is unimodal since the preferred style of LLM is consistent when generating codes. Then, we employ average pooling between the representation of $\mathbf{v}_{cq}$ and $\mathbf{v}_q$ to get the augmented representation $\mathbf{v}_q^+$. The total process can be described as:

$$\mathbf{v}_q^+ = \frac{1}{2N}\left(N \cdot G(q) + \sum_{c_q \sim \mathrm{LLM}(q, \textsc{gen})} G(c_q)\right), \tag{8}$$

where N is the number of exemplar codes. Similarly, we can get the augmented representation $\mathbf{v}_c^+$ of each code. During evaluation, $\mathbf{v}_q$ and $\mathbf{v}_c$ in Eq.(1) are replaced by $\mathbf{v}_q^+$ and $\mathbf{v}_c^+$.

Theoretical Insights We offer theoretical insights to differentiate GAR and our proposed ReCo. Each code in the codebase is an implementation of a specific query, which we denote as $c \sim P(q)$. Here P denotes a real-world distribution between queries and codes. LLM also defines a probability distribution over queries. Thus, exemplar codes can be considered to follow $c_q \sim \mathrm{LLM}(q)$. We could find that c and c_q are sampled from two different distributions given q. This accounts for the occasional divergence between the true code and the exemplar code for the same query, as illustrated in Fig. 1.

The rewritten code follows $c_c \sim \mathrm{LLM}(q_{sum})$. If the query q and code c are identical in semantics and q_{sum} correctly reflect the functionality of code c, we could approximate the distribution of $\mathrm{LLM}(q_{sum})$ as $\mathrm{LLM}(q)$. Once the exemplar code and the rewritten code are both sampled from $\mathrm{LLM}(q)$, the expectation of LLM-generated content becomes more similar, which is reflected in the style of generated codes.

4 Code Style Similarity

To quantitatively measure the style difference among codes, we propose a novel evaluation metric, dubbed Code Style Similarity (CSSim). To the best of our knowledge, this is the first metric addressing the similarity between two codes from a stylistic perspective. Indeed, there are several evaluation metrics widely adopted in code generation or translation to measure semantic similarity like BLEU and CodeBLEU. Yet they are not suitable for measuring the style similarity.

The basic idea of these metrics is to compare the predicted code snippet against the ground truth by calculating the intersection of contiguous sequences of code tokens (i.e., n-grams). It is recognized that due to the syntactic verbosity and coding conventions inherent to programming languages, two code snippets frequently share numerous n-grams that are incapable of reflecting their stylistic nuances. Besides, the score is calculated based on the exact match of n-grams, which can be deemed excessively rigid. For example, compared with `token_count`, `word_count` is expected to be more stylistically similar to `words_count`. However, both of them will be assigned a score of 0 under 2-gram match.

CSSim addresses style from three perspectives: variable naming, API invocation, and code structure. Variable naming is generally believed as a reflection of the programmer's preference. For API invocation, similar APIs often exist in various libraries or packages, the choice of APIs also indicates the preference. As for code structure, sometimes the swap of two lines does not influence the operation hence the order should also be considered. Besides, CSSim is calculated based on a softer measurement, edit distance (Ristad and Yianilos, 1998).

API invocation and variable name follow the same process. Here we take the variable name as an example. We first extract all the variables from the code snippet to get $\mathbf{V} = \{v_i\}_{i=1}^{N}$. For each variable in the set, we find the most similar variable from the other code and take the edit distance between the two as the similarity of this variable. Then, we take the weighted average value of all the variables

as the style distance in variable naming. The whole process can be described as:

$$\text{Dis}_{\text{V}_1} = \frac{1}{||\lambda||_1} \sum_{v_i \in \mathbf{V}_1} \lambda_i \min_{v_j \in \mathbf{V}_2} \text{ED}(v_i, v_j), \quad (9)$$

where $\mathbf{V}_1$ and $\mathbf{V}_2$ are extracted variables from two codes and ED denotes Edit Distance. λ_i is normalized inverse document frequency (IDF) because we intend to decrease the impact of common words. To ensure symmetry in this metric, we update code distance in variable naming as:

$$\text{Dis}_{\text{Var}} = \frac{\text{Dis}_{\text{V1}} + \text{Dis}_{\text{V2}}}{2}. \quad (10)$$

For the measurement of code structure, we simply apply Tree Edit Distance (TED) (Paaßen, 2018) to the Abstract Syntax Tree transformed from the code. Similar to edit distance, TED quantifies the least amount of basic operations (Insertion, Deletion, and Substitution) required to transform one tree into the other. To calculate CSSim, we first calculate the Code Style Distance CSDis between two codes c_1 and c_2, which is:

$$\text{CSDis}(c_1, c_2) = \frac{\text{Dis}_{\text{Var}} + \text{Dis}_{\text{API}} + \text{TED}}{3}, \quad (11)$$

where Dis_{Var}, Dis_{API}, $\text{TED} \in [0, 1]$ hence $\text{CSDis} \in [0, 1]$. We define $\text{CSSim} = 1 - \text{CSDis}$.

5 Experimental Setups

Datasets We evaluate ReCo across various search scenarios and programming languages: online forum StackOverflow CoNaLa (Yin et al., 2018), coding challenge competence APPS (Hendrycks et al., 2021), general programming problems MBPP (Austin et al., 2021) and MBJP (Athiwaratkun et al., 2022). The first three datasets are written in Python while the last one is written in Java. The statistics of the datasets are shown in Appendix B.1. We take the widely adopted Mean Reciprocal Rank (MRR) as the evaluation metric (Li et al., 2023). MRR is the average of reciprocal ranks of the true code snippets for the given query.

Baselines We apply ReCo on several models: **BM25**, an enhanced version of TF-IDF, is a statistical measure that matches certain keywords in codes with the given query. **CodeBERT** is a bimodal model pre-trained on Masked Language Modeling and Replaced Token Detection. **UniXcoder** unifies both generating and understanding pre-training tasks to further enhance code representation learning by leveraging cross-modal contents. **Contriever** (Izacard et al., 2021) is an unsupervised dense information retrieval model that leverages contrastive learning for its training. **CodeT5+** is pre-trained on both unimodal code data and bimodal code-text data with a diverse set of pre-training tasks including span denoising, causal language modeling, contrastive learning, and text-code matching.

	CoNaLa	MBPP	APPS	MBJP
Unsupervised				
BM25	52.6	12.6	11.6	11.3
+ GAR	71.7	35.1	17.6	33.5
+ ReCo	**75.8**$_{+4.1}$	**70.8**$_{+35.7}$	**22.6**$_{+5.0}$	**65.3**$_{+31.8}$
UniXcoder	77.2	69.3	8.3	73.2
+ GAR	83.9	85.0	13.2	80.0
+ ReCo	**85.1**$_{+1.2}$	**92.4**$_{+7.4}$	**28.8**$_{+15.6}$	**87.6**$_{+7.6}$
Contriever	55.7	55.3	9.6	37.0
+ GAR	75.0	71.3	14.0	62.3
+ ReCo	**77.9**$_{+2.9}$	**87.4**$_{+16.1}$	**41.6**$_{+27.6}$	**76.6**$_{+14.3}$
CodeT5+	73.7	59.4	7.6	67.7
+ GAR	80.3	77.7	10.2	79.2
+ ReCo	**80.8**$_{+0.5}$	**89.4**$_{+11.7}$	**29.9**$_{+19.7}$	**84.0**$_{+4.8}$
Supervised				
CodeBERT	83.6	79.6	25.1	79.6
+ GAR	**88.6**	87.7	29.3	84.1
+ ReCo	85.0$_{-3.6}$	**92.3**$_{+4.6}$	**51.2**$_{+21.9}$	**89.1**$_{+5.0}$
UniXcoder	84.8	81.2	24.3	81.6
+ GAR	85.9	89.0	34.5	85.6
+ ReCo	**87.1**$_{+1.2}$	**94.2**$_{+5.2}$	**58.1**$_{+23.6}$	**90.5**$_{+4.9}$

Table 1: Comparative analysis of various models w.r.t MRR(%) when utilizing GAR or ReCo.

Compared Metrics We compare Code Style Similarity with several metrics used for measuring semantic similarity. **BLEU** measures how many words are shared between the generated and the reference sentence based on the modified n-gram precision. **ROUGE-L** computes the longest common subsequence of words. **CodeBLEU** is tailored for code snippets by setting higher weights for programming language keywords and considering data-flow and AST match as well.

Implementation Details When prompting LLMs, we randomly sample 4 in-context examples from the training sets and set a temperature of 1. And when we prompt LLMs multiple times for the

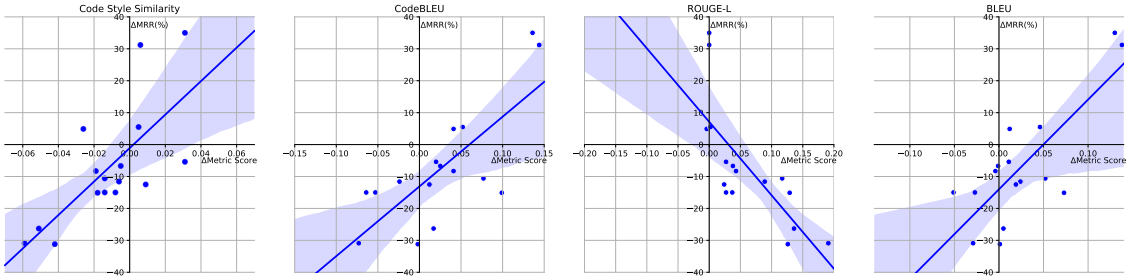

Figure 3: Regression plots between $\Delta\text{MRR}(= \text{MRR}_{\text{ReCo}} - \text{MRR}_{\text{GAR}})$ and $\Delta\text{MetricScore}(= \text{Metric}(c_q, c_c) - \text{Metric}(c_q, c))$ under different evaluation metrics. The data points are from BM25 results on four datasets with four LLMs.

same input, each time we resample the in-context example. The maximum length of output for code summarization and generation is 128 and 256, respectively. For sparse retrieval, we use the default implementation from Pyserini (Lin et al., 2021). For dense retrieval, during training, we adopt the default hyperparameters described in the original paper. They are trained for 10 epochs with a batch size of 32. Experiments are conducted on a Nvidia Tesla A100 GPU. Please refer to Appendix B.3 for more details.

6 Results

Overall Results The results are shown in Table 1. It is worth noting that our experiments encompass the use of various LLMs and multiple instances of both exemplar codes and rewritten codes for conducting ablation studies. Here we report the best performance when equipped with ReCo. Comprehensive results can be found in Appendix D. We can observe that ReCo significantly outperforms GAR on both supervised and unsupervised models across diverse search scenarios. With ReCo, the non-neural model BM25 can have competitive performance compared to neural models under zero-shot setting. And ReCo could boost the performance of zero-shot neural models similar to supervised models. We also evaluate ReCo on Contriever, a passage retrieval model that is not specifically trained for code-related tasks. We argue that ReCo can also benefit general-purpose retrieval models. Note that compared with GAR, ReCo does not bring any additional computation in real-time search because the rewritten code could be pre-processed and stored in the codebase.

Comparison among Evaluation Metrics To demonstrate the superiority of Code Style Similarity, we prove that *existing metrics are not effective in measuring the style similarity between two codes* by contradiction. If existing metrics are effective, they should satisfy two necessary conditions: 1) the variation of metric scores $\Delta\text{MetricScore}$ between $\text{Metric}(c_q, c_c)$ and $\text{Metric}(c_q, c)$ is in the same direction with code search performance gap between ReCo and GAR ($\Delta\text{MRR} = \text{MRR}_{\text{ReCo}} - \text{MRR}_{\text{GAR}}$). This is because once the rewritten code is closer to the exemplar code in style, the code search performance should improve accordingly. 2) If we only choose the best one with the highest $\text{Metric}(c_q, c)$ among multiple exemplar codes, the performance should be significantly better than randomly selecting one exemplar code.

Dataset	Random	w/ the best exemplar code			
		CSSim	CodeBLEU	ROUGE-L	BLEU
CoNaLa	41.3	**43.6**	39.6	41.5	42.3
MBPP	24.0	**26.8**	25.1	23.7	24.0
APPS	14.1	**15.2**	14.8	14.2	14.2
MBJP	26.6	**29.9**	29.1	27.2	28.8

Table 2: Performance comparison of the best exemplar code selection versus random selection for GAR across four datasets, using the Code Llama-7B model.

For the first condition, we analyze the results from BM25 on the four datasets with different LLMs including GPT3.5, Code Llama-7B, 13B, and 34B. Here we do not take the results from dense retrieval systems because neural models can capture the potential relationship among similar tokens. The numerical scores under different evaluation metrics are shown in Appendix D. Regression plots are shown in Fig. 3. According to our first condition, $\Delta\text{MetricScore}$ and ΔMRR should be consistent, which means that points are expected to be scattered on Quadrant I and III. We can see that most of the points in CodeBLEU, ROUGE-L, and BLEU are scattered on Quadrant IV. In other

	CoNaLa	MBPP	APPS	MBJP
UniXcoder	77.2	69.3	8.3	73.2
UniXcoder + ReCo	**85.1**	**86.2**	**27.3**	**83.4**
w/o original query&code	83.1	84.4	26.8	78.1
UniXcoder-ft	84.8	81.2	24.3	81.6
UniXcoder-ft + ReCo	**87.1**	**88.0**	**48.8**	**85.4**
w/o original query&code	85.5	84.8	39.0	80.3

Table 3: Comparative performance analysis of using exclusively LLM-generated codes versus a combination of LLM-generated codes, original queries, and codes. "UniXcoder-ft" represents UniXcoder after fine-tuning.

	CoNaLa	MBPP	APPS	MBJP
BM25	52.6	12.6	11.6	11.3
w/ Code Llama-7B	14.2	14.0	7.8	15.4
w/ Code Llama-13B	29.8	26.3	8.1	28.4
w/ Code Llama-34B	20.4	14.7	4.7	15.8
w/ GPT3.5	**75.8**	**70.8**	**22.6**	**65.3**

Table 4: Performance of ReCo on BM25 when using different LLMs to generate exemplar and rewritten codes.

words, when the rewritten codes are considered to be more similar to the exemplar code by these metrics, the performance of ReCo, on the contrary, drops compared with GAR. Points from Code Style Similarity mostly fall on Quadrant I and III and the regression line nearly passes through the origin.

For the second condition, we analyze the results from BM25 equipped with GAR. For each query, we calculate the metric score between its exemplar codes and the true code in the codebase and then select the one with the highest metric score. The performance on four datasets after selecting the best exemplar code generated by Code Llama-7B is shown in Table 2. We can observe that compared with random selection, the improvement brought by CodeBLEU, ROUGE-L, and BLEU is not significant generally. On the contrary, Code Style Similarity outperforms other settings. To better understand the preference of different evaluation metrics, we also conduct case studies in Appendix C.

In conclusion, our findings indicate that existing metrics for measuring code style similarity fall short when subjected to two contradictory conditions. Conversely, Code Style Similarity (CSSim) demonstrably satisfies these criteria, highlighting its superior effectiveness in quantifying stylistic similarities in code. Furthermore, we observe a clear positive correlation between code style similarity as measured by CSSim and improvement in MRR for code search, thereby validating ReCo's motivation that style normalization is advantageous.

Using only LLM-generated codes To further demonstrate that the exemplar code and rewritten code are similar in style, we conduct experiments to only use these LLM-generated codes in the retrieval system. In other words, we use exemplar code to retrieve rewritten codes. The results of UniXcoder under fine-tuning and zero-shot settings on four datasets are shown in Table 3. We can see from the results that only using LLM-generated codes can reach competitive performance compared with additionally using original queries and codes, and even outperform the setting only using original queries and codes, which indicates that the consistent style in exemplar code and rewritten code have made retrieval easier.

Impact of Different LLMs We explore the effect of using different LLMs in ReCo. The results on BM25 are shown in Table 4. The full results including other retrieval models are in Appendix D. GPT3.5's number of parameters is not released publicly but is estimated at around 175 billion. Generally, we observe that larger LLMs yield greater improvements. However, a decrement in performance is noted with the application of Code Llama-34B. This decrement is attributed to the model's propensity to generate code not only for the prompted fifth example but also for the initial four in-context examples. Consequently, the generated code is often truncated due to output length limitations.

Impact of Number of Generated Codes We also explore the effect of the numbers of generated exemplar codes and rewritten codes in ReCo. The outcomes of BM25 on CoNaLa and MBPP are depicted in Fig. 4, while a comprehensive compilation of results, inclusive of other retrieval models and datasets, can be found in Appendix D. Our observations indicate a marginal enhancement when LLMs are tasked with generating more codes. We discern that the multiple codes generated exhibit similarities, with minor variations, attributable to the self-consistent style of each LLM. To address this, our future work will investigate controlled prompts that steer LLMs towards generating code with controlled stylistic variations, thereby enhancing the diversity of code generation. Fig. 4 also illustrates that the improvement tends to diminish as the quantity of generated codes increases. In practical applications, it is essential to weigh the

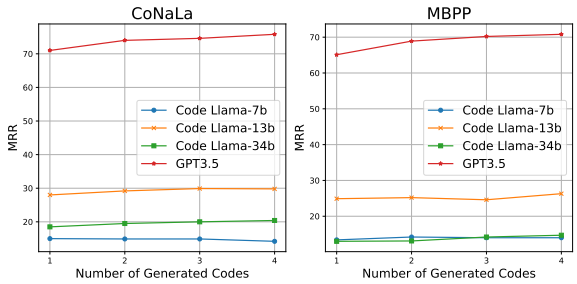

Figure 4: Performance of BM25 + ReCo with different numbers of generated codes.

trade-off between performance enhancement and the incremental costs associated with generation.

7 Broader Impact

As we stated, the key motivation behind ReCo is to normalize the code style between exemplar code and original code. Indeed, there exist code normalization methods, but they only focus on superficial formatting such as the usage of indentation and naming convention (e.g., from camelCase to snake_case). In this paper, we discuss code style normalization from a deeper perspective, implementation logic, and preference for variable naming or API invocation. We believe that this task has great potential as it could not only benefit code search but also many other code-related tasks like code review and code translation.

In this paper, we adopt LLMs to achieve the goal of style normalization by first summarizing the code snippet and then generating code based on the summary. This is because we find directly asking LLMs to rewrite the code results in very similar outputs. In the process of summarize-then-generate, models are expected to have great code intelligence hence there is no loss of information, as described in the theoretical insights of ReCo. Yet we are aware of the huge cost brought by LLMs. To decrease the cost, one promising solution is to train models specifically used for code style normalization. These models are considered to have much fewer parameters since much general knowledge in LLMs is not needed. To push forward the research of code style normalization, we propose a suitable evaluation metric, dubbed Code Style Similarity. In our future work, we plan to train such models to improve the efficiency of ReCo.

8 Conclusion

In this paper, we propose ReCo, an LLM-augmented code search framework built on GAR, that additionally rewrites the code in the code base to normalize the code style between exemplar code and code in the codebase. We evaluate ReCo on several code search models across various search scenarios with different programming languages. Experimental results demonstrate the effectiveness of ReCo by significantly boosting the performance of models. To encourage further research works on code style normalization and explain the effect of ReCo, we propose an evaluation metric Code Style Similarity. In our future work, based on this metric, we may develop new models that can more efficiently normalize the code style.

Acknowledgement

We thank the anonymous reviewers for their helpful comments and suggestions.

Limitations

There are mainly two limitations of this work. First, although ReCo does not require any additional computation in real-time search compared with GAR, both GAR and ReCo rely on the real-time generation of exemplar codes. Therefore, ReCo and GAR may have limitations when applied to tasks that demand low latency. The latency of generating exemplar codes depends on the time cost of LLM inference. As stated in research works focusing on GAR, over the years the cost of hardware has decreased a lot and there are many works proposed to improve the inference efficiency of LLMs (Gao et al., 2022; Wang et al., 2023a). We believe the efficiency problem of GAR and ReCo will be addressed in the future. The second limitation is that we do not evaluate ReCo on some extremely large-scale codebases like CodeSearchNet (Husain et al., 2019). This is due to the time burden of generating exemplar codes and rewriting codes. For example, according to our estimation, there are 1,005,474 queries in total in CodeSearchNet hence generating one exemplar code for them costs more than two months. To address this limitation, we evaluate ReCo on several search scenarios covering coding challenge competence, online programming community, and general programming problems to show the effectiveness of ReCo.

References

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. 2022. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

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*, pages 964–974.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Aryaz Eghbali and Michael Pradel. 2022. Crystalbleu: precisely and efficiently measuring the similarity of code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics*, pages 1536–1547.

Luyu Gao, Xueguang Ma, Jimmy Lin, and Jamie Callan. 2022. Precise zero-shot dense retrieval without relevance labels. *arXiv preprint arXiv:2212.10496*.

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)*, pages 483–494.

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

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*, pages 7212–7225.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Emily Hill, Lori Pollock, and K Vijay-Shanker. 2011. Improving source code search with natural language phrasal representations of method signatures. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 524–527.

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. Cosqa: 20, 000+ web queries for code search and question answering. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*, pages 5690–5700.

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*.

Gautier Izacard, Mathilde Caron, Lucas Hosseini, Sebastian Riedel, Piotr Bojanowski, Armand Joulin, and Edouard Grave. 2021. Unsupervised dense information retrieval with contrastive learning. *arXiv preprint arXiv:2112.09118*.

Haochen Li, Chunyan Miao, Cyril Leung, Yanxian Huang, Yuan Huang, Hongyu Zhang, and Yanlin Wang. 2022a. Exploring representation-level augmentation for code search. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 4924–4936.

Haochen Li, Xin Zhou, Luu Anh Tuan, and Chunyan Miao. 2023. Rethinking negative pairs in code search. *arXiv preprint arXiv:2310.08069*.

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

Xiaonan Li, Daya Guo, Yeyun Gong, Yun Lin, Yelong Shen, Xipeng Qiu, Daxin Jiang, Weizhu Chen, and Nan Duan. 2022c. Soft-labeled contrastive pre-training for function-level code representation. *arXiv preprint arXiv:2210.09597*.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

Jimmy Lin, Xueguang Ma, Sheng-Chieh Lin, Jheng-Hong Yang, Ronak Pradeep, and Rodrigo Nogueira. 2021. Pyserini: A python toolkit for reproducible information retrieval research with sparse and dense representations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2356–2362.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Yuetian Mao, Chengcheng Wan, Yuze Jiang, and Xiaodong Gu. 2023. Self-supervised query reformulation for code search. *arXiv preprint arXiv:2307.00267*.

Yuning Mao, Pengcheng He, Xiaodong Liu, Yelong Shen, Jianfeng Gao, Jiawei Han, and Weizhu Chen. 2020. Generation-augmented retrieval for open-domain question answering. *arXiv preprint arXiv:2009.08553*.

Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 111–120.

Liming Nie, He Jiang, Zhilei Ren, Zeyi Sun, and Xiaochen Li. 2016. Query expansion based on crowd knowledge for code search. *IEEE Transactions on Services Computing*, 9(5):771–783.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

OpenAI. 2023a. Chatgpt.

OpenAI. 2023b. Gpt-4 technical report. *ArXiv*, abs/2303.08774.

Benjamin Paaßen. 2018. Revisiting the tree edit distance and its backtracing: A tutorial. *arXiv preprint arXiv:1805.06869*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Zhen Qin, Rolf Jagerman, Kai Hui, Honglei Zhuang, Junru Wu, Jiaming Shen, Tianqi Liu, Jialu Liu, Donald Metzler, Xuanhui Wang, et al. 2023. Large language models are effective text rankers with pairwise ranking prompting. *arXiv preprint arXiv:2306.17563*.

Mohammad Masudur Rahman and Chanchal K Roy. 2021. A systematic literature review of automated query reformulations in source code search. *arXiv preprint arXiv:2108.09646*.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Eric Sven Ristad and Peter N Yianilos. 1998. Learning string-edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):522–532.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Abdus Satter and Kazi Sakib. 2016. A search log mining based query expansion technique to improve effectiveness in code search. In *2016 19th International Conference on Computer and Information Technology*, pages 586–591.

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*.

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*, pages 196–207.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023a. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023b. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Aaron Van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation learning with contrastive predictive coding. *arXiv e-prints*, pages arXiv–1807.

Liang Wang, Nan Yang, and Furu Wei. 2023a. Query2doc: Query expansion with large language models. *arXiv preprint arXiv:2303.07678*.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023b. Codet5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1069–1088.

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*.

Yangrui Yang and Qing Huang. 2017. Iecs: Intent-enforced code search via extended boolean model. *Journal of Intelligent & Fuzzy Systems*, 33(4):2565–2576.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th international conference on mining software repositories*, pages 476–486.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022. When neural model meets nl2code: A survey. *arXiv preprint arXiv:2212.09420*.

A Complete Prompt

Table 11 and Table 12 are two complete prompt examples for generating exemplar codes and summarizing original codes, respectively. Note that in the second step of rewriting original codes, we also adopt the prompt structure of generating exemplar codes but replace the description at last with the summary.

B Experiment Settings

B.1 Dataset Statistics

The dataset statistics are shown in Table 5. The numbers here are pairs of queries and their true code snippet. Code search models are asked to distinguish the correct code from the codes from other pairs. Note that in the original APPS dataset, there are 4,284 and 3,515 pairs in the training and test set, respectively. Due to the huge cost of prompting LLMs, we randomly sample a subset for our evaluation.

B.2 MRR Calculation

MRR is the average of reciprocal ranks of the true code snippets for the given query, which could be calculated as:

Dataset	CoNaLa	MBPP	APPS	MBJP
Train	2,379	373	1,000	374
Test	500	500	1,000	500

Table 5: Statistics of the dataset used in our experiment.

True Code

```
def find_first_duplicate(nums):
    num_set = set()
    no_duplicate = -1
    for i in range(len(nums)):
        if nums[i] in num_set:
            return nums[i]
        else:
            num_set.add(nums[i])
    return no_duplicate
```

Exemplar Code 1

```
def find_duplicate(my_list):
    for i in range(len(my_list)):
        if my_list[i] in my_list[i+1:]:
            return my_list[i]
    return None
```

Exemplar Code 2

```
def find_duplicate(my_list):
    seen = set()
    for num in my_list:
        if num in seen:
            return num
        seen.add(num)
    return None
```

Figure 5: Case study between Code Style Similarity and the existing metrics. The first exemplar code is preferred by existing metrics while the second one is preferred by Code Style Similarity. The second exemplar code is more similar to the true code from the perspective of style.

$$\mathrm{MRR} = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{\mathrm{Rank}_i} \tag{12}$$

where Rank_i is the rank of the true code for the i-th given query Q.

B.3 Implementation Details

For neural models, they all use the same set of hyperparameters. The maximum input length of codes and queries are both set to be 256. Models are trained by Adam and learning rate is set to 5e-6. We adopt mean pooling to get the representation of the whole input sentence to make sure the pooling mechanism is consistent with that during pre-training. The representations are normalized by the L2 norm.

Metric	LLM	Datasets											
		CoNaLa			MBPP			APPS			MBJP		
		$M(c_q, c_c)$	$M(c_q, c)$	ΔM	$M(c_q, c_c)$	$M(c_q, c)$	ΔM	$M(c_q, c_c)$	$M(c_q, c)$	ΔM	$M(c_q, c_c)$	$M(c_q, c)$	ΔM
CSSim	CodeLlama-7B	0.492	0.543	-0.051	0.504	0.518	-0.014	0.493	0.498	-0.005	0.529	0.52	0.009
	CodeLlama-13B	0.534	0.593	-0.059	0.547	0.565	-0.018	0.481	0.5	-0.019	0.52	0.528	-0.008
	CodeLlama-34B	0.522	0.564	-0.042	0.527	0.533	-0.006	0.526	0.495	0.031	0.506	0.52	-0.014
	GPT3.5	0.553	0.548	0.005	0.553	0.522	0.031	0.508	0.534	-0.026	0.502	0.496	0.006
CodeBLEU	CodeLlama-7B	0.16	0.143	0.017	0.264	0.187	0.077	0.153	0.128	0.025	0.324	0.312	0.012
	CodeLlama-13B	0.116	0.189	-0.073	0.348	0.249	0.099	0.171	0.13	0.041	0.311	0.375	-0.064
	CodeLlama-34B	0.181	0.183	-0.002	0.184	0.208	-0.024	0.151	0.131	0.020	0.266	0.319	-0.053
	GPT3.5	0.28	0.228	0.052	0.385	0.249	0.136	0.204	0.163	0.041	0.462	0.318	0.144
ROUGE-L	CodeLlama-7B	0.138	0.002	0.136	0.118	0.001	0.117	0.041	0.003	0.038	0.025	0.001	0.024
	CodeLlama-13B	0.193	0.002	0.191	0.13	0.001	0.129	0.046	0.003	0.043	0.038	0.001	0.037
	CodeLlama-34B	0.127	0.001	0.126	0.0898	0.001	0.0888	0.03	0.003	0.027	0.027	0	0.027
	GPT3.5	0.007	0.005	0.002	0.001	0.001	0	0.003	0.007	-0.004	0	0	0
BLEU	CodeLlama-7B	0.017	0.012	0.005	0.081	0.029	0.052	0.017	0.018	-0.001	0.115	0.096	0.019
	CodeLlama-13B	0.026	0.055	-0.029	0.143	0.07	0.073	0.019	0.023	-0.004	0.146	0.197	-0.051
	CodeLlama-34B	0.026	0.025	0.001	0.054	0.03	0.024	0.021	0.01	0.011	0.109	0.136	-0.027
	GPT3.5	0.145	0.099	0.046	0.195	0.065	0.13	0.048	0.036	0.012	0.309	0.171	0.138

Table 6: Full results of $\text{MetricScore}(c_q, c_c)$ and $\text{MetricScore}(c_q, c)$ under different metrics. M is short for MetricScore. $\Delta\text{M} = \text{M}(c_q, c_c) - \text{M}(c_q, c)$.

UniXcoder is initialized using the publicly available checkpoint at https://huggingface.co/microsoft/unixcoder-base, Contriever is initialized using https://huggingface.co/facebook/contriever-msmarco, and CodeT5+ is initialized using https://huggingface.co/Salesforce/codet5p-110m-embedding. CodeBERT is initialized using https://huggingface.co/microsoft/codebert-base and then pre-trained on the CodeSearchNet dataset (Husain et al., 2019) for 10 epochs. The pre-training setting is the same as in fine-tuning. All the experiments involving model training are running with 3 random seeds 1234, 12345, and 123456 and they all meet $p < 0.01$ of significance tests.

C Case Study

We also conduct a case study to show the superiority of Code Style Similarity. Fig. 5 shows two exemplar codes generated by Code Llama-7B. The true code aims to find the first duplicate element in a given array of integers. Although both the two exemplar codes satisfy the description, their implementation style is different. The first exemplar code is preferred by CodeBLEU, ROUGE-L, and BLEU while the second one is preferred by Code Style Similarity. The true code uses a set to collect seen elements when traversing the list, which is also the logic in the second exemplar code. The first exemplar code implements the function in a different way by checking whether `my_list[i]` appears in `my_list[i+1:]`. We think the first code is preferred by existing metrics because lines 2-4 in the first exemplar code are very similar to lines 4-6 in the true code, which contributes a lot to the metric score.

D Full Results

In this section, we report the full experimental results. The results of $\text{MetricScore}(c_q, c_c)$ and $\text{MetricScore}(c_q, c)$ under different evaluation metrics are shown in Table 6. The results of BM25 are shown in Table 7. The results of fine-tuned UniXcoder and CodeBERT are shown in Table 8 and Table 9, respectively. The results of UniXcoder, Contriever, and CodeT5+ under zero-shot setting are shown in Table 10.

Model	LLM	#gen	Framework	Datasets			
				CoNaLa	MBPP	APPS	MBJP
BM25	CodeLlama-7B	1	GAR	41.3	24.0	14.1	26.6
			ReCo	15.0	13.4	7.4	14.1
		2	GAR	44.0	25.5	14.9	28.4
			ReCo	14.9	14.2	7.6	14.1
		3	GAR	44.5	26.3	15.4	29.3
			ReCo	14.9	14.0	7.5	14.8
		4	GAR	44.5	26.4	15.2	29.7
			ReCo	14.2	14.0	7.8	15.4
	CodeLlama-13B	1	GAR	58.9	40.0	16.0	41.8
			ReCo	28.0	24.9	7.7	14.1
		2	GAR	62.7	42.1	16.5	4.4
			ReCo	29.2	25.2	8.0	28.1
		3	GAR	64.0	41.4	16.8	45.1
			ReCo	29.9	24.6	8.0	28.6
		4	GAR	63.9	42.2	17.0	45.4
			ReCo	29.8	26.3	8.1	28.4
	CodeLlama-34B	1	GAR	49.7	24.6	10.0	29.6
			ReCo	18.5	13.0	4.6	14.6
		2	GAR	53.0	24.6	10.6	30.2
			ReCo	19.5	13.1	4.8	14.4
		3	GAR	55.5	24.6	10.9	31.5
			ReCo	20.0	14.2	4.8	15.3
		4	GAR	56.8	25.7	10.9	32.3
			ReCo	20.4	14.7	4.7	15.8
	GPT3.5	1	GAR	65.5	30.2	16.3	30.6
			ReCo	71.0	65.1	21.2	61.8
		2	GAR	69.7	34.5	17.0	32.2
			ReCo	74.0	68.9	21.9	65.4
		3	GAR	71.0	35.3	17.3	33.1
			ReCo	74.6	70.2	22.8	65.6
		4	GAR	<u>71.7</u>	<u>35.1</u>	<u>17.6</u>	<u>33.5</u>
			ReCo	**75.8**	**70.8**	**22.6**	**65.3**

Table 7: Full results of BM25. #gen denotes the number of generated and rewritten codes. **Bold** and <u>underlined</u> results are the best performance of ReCo and the performance of GAR under the same setting, which are reported in Table 1.

Model	LLM	#gen	Framework	Datasets			
				CoNaLa	MBPP	APPS	MBJP
UniXcoder	CodeLlama-7B	1	GAR	85.4	74.2	32.2	77.1
			ReCo	72.5	76.7	51.3	81.7
		2	GAR	86.7	74.8	34.0	79.1
			ReCo	78.8	79.3	54.9	83.5
		3	GAR	87.0	75.0	34.5	79.3
			ReCo	81.7	80.2	**58.1**	84.1
	CodeLlama-13B	1	GAR	89.2	87.9	36.3	84.9
			ReCo	81.0	92.9	41.4	90.0
		2	GAR	90.2	89.0	36.9	85.6
			ReCo	85.1	93.2	46.1	89.6
		3	GAR	90.8	89.0	38.3	85.6
			ReCo	85.5	**94.2**	46.8	**90.5**
	CodeLlama-34B	1	GAR	87.0	81.3	29.9	81.2
			ReCo	82.3	65.2	28.4	66.9
		2	GAR	87.6	83.8	31.2	83.6
			ReCo	85.6	75.7	34.5	72.8
		3	GAR	88.4	83.6	32.9	84.6
			ReCo	87.0	78.8	38.2	76.6
	GPT3.5	1	GAR	85.9	79.9	44.5	80.5
			ReCo	**87.1**	88.0	48.8	85.4

Table 8: Full results of UniXcoder after fine-tuning. #gen denotes the number of generated and rewritten codes. **Bold** and underlined results are the best performance of ReCo and the performance of GAR under the same setting, which are reported in Table 1. Note that due to the cost of OpenAI's API for using GPT3.5, we only generate one exemplar code and rewrite the code once for the training set. And due to the GPU memory limit, we can set a maximum number of #gen as 3.

Model	LLM	#gen	Framework	Datasets			
				CoNaLa	MBPP	APPS	MBJP
CodeBERT	CodeLlama-7B	1	GAR	81.7	72.6	26.8	75.4
			ReCo	65.5	75.6	45.5	79.7
		2	GAR	84.8	73.8	28.4	75.9
			ReCo	73.7	77.4	48.8	81.2
		3	GAR	85.4	74.0	29.3	76.1
			ReCo	77.1	78.6	**51.2**	81.2
	CodeLlama-13B	1	GAR	87.2	87.2	29.5	82.9
			ReCo	77.1	90.5	38.2	88.3
		2	GAR	89.2	87.6	30.8	83.6
			ReCo	81.4	91.3	40.9	88.8
		3	GAR	89.8	87.7	31.6	84.1
			ReCo	83.3	**92.3**	41.6	**89.1**
	CodeLlama-34B	1	GAR	86.1	78.5	23.3	79.7
			ReCo	78.3	59.6	22.2	61.0
		2	GAR	87.4	81.1	25.2	81.8
			ReCo	82.4	68.9	27.0	68.4
		3	GAR	88.6	80.9	26.4	81.5
			ReCo	**85.0**	71.3	30.8	71.7
	GPT3.5	1	GAR	83.3	81.6	38.2	80.9
			ReCo	82.4	83.5	43.1	81.4

Table 9: Full results of CodeBERT after fine-tuning. #gen denotes the number of generated and rewritten codes. **Bold** and underlined results are the best performance of ReCo and the performance of GAR under the same setting, which are reported in Table 1. Note that due to the cost of OpenAI's API for using GPT3.5, we only generate one exemplar code and rewrite the code once for the training set. And due to the GPU memory limit, we can set a maximum number of #gen as 3.

Model	LLM	#gen	Framework	Datasets			
				CoNaLa	MBPP	APPS	MBJP
UniXcoder	CodeLlama-7B	4	GAR	77.2	62.5	13.2	68.4
			ReCo	75.4	70.7	**28.8**	77.4
	CodeLlama-13B	4	GAR	85.1	85.0	16.6	80.0
			ReCo	82.5	**92.4**	25.4	**87.6**
	CodeLlama-34B	4	GAR	81.9	75.9	9.1	78.3
			ReCo	83.2	75.8	14.4	74.5
	GPT3.5	4	GAR	83.9	79.7	19.6	80.0
			ReCo	**85.1**	86.2	27.3	83.4
Contriever	CodeLlama-7B	4	GAR	54.3	50.8	14.0	44.8
			ReCo	55.0	66.2	**41.6**	67.2
	CodeLlama-13B	4	GAR	69.3	71.3	16.4	62.3
			ReCo	72.0	**87.4**	34.4	**76.6**
	CodeLlama-34B	4	GAR	61.3	60.3	10.1	50.1
			ReCo	68.3	63.9	20.3	56.0
	GPT3.5	4	GAR	75.0	65.6	17.9	58.7
			ReCo	**77.9**	79.5	24.1	72.4
CodeT5+	CodeLlama-7B	4	GAR	67.4	61.3	10.2	66.5
			ReCo	57.7	67.2	**29.9**	73.0
	CodeLlama-13B	4	GAR	78.6	77.7	12.8	79.2
			ReCo	72.8	**89.4**	24.7	**84.0**
	CodeLlama-34B	4	GAR	74.5	68.4	8.1	76.5
			ReCo	67.6	57.9	15.8	62.9
	GPT3.5	4	GAR	80.3	72.9	14.6	76.9
			ReCo	**80.8**	82.2	19.1	81.4

Table 10: Full results of UniXcoder, Contriever, and CodeT5+ under zero-shot setting. #gen denotes the number of generated and rewritten codes. **Bold** and underlined results are the best performance of ReCo and the performance of GAR under the same setting, which are reported in Table 1.

Prompt

Please generate a python code snippet according to the last given description. Only output the code snippets. Do not explain the code. I will show you four examples first.

Description: Write a python function to find the index of an extra element present in one sorted array.
Code:

```
def find_Extra(arr1,arr2,n) :
    for i in range(0, n) :
        if (arr1[i] != arr2[i]) :
            return i
    return n
```

Description: Write a function to decode a run-length encoded given list.
Code:

```
def decode_list(alist):
    def aux(g):
        if isinstance(g, list):
            return [(g[1], range(g[0]))]
        else:
            return [(g, [0])]
    return [x for g in alist for x, R in aux(g) for i in R]
```

Description: Write a python function to accept the strings which contains all vowels.
Code:

```
def check(string):
    if len(set(string).intersection("AEIOUaeiou"))>=5:
        return ('accepted')
    else:
        return ("not accepted")
```

Description: Write a function to compute binomial probability for the given number.
Code:

```
def get_inv_count(arr, n):
    inv_count = 0
    for i in range(n):
        for j in range(i + 1, n):
            if (arr[i] > arr[j]):
                inv_count += 1
    return inv_count
```

Description: Write a function to sort a given matrix in ascending order according to the sum of its rows.
Code:

Output

```
def sort_matrix_by_row_sum(matrix):
    row_sums = []
    for row in matrix:
        row_sum = sum(row)
        row_sums.append(row_sum)
    sorted_matrix = [x for _, x in sorted(zip(row_sums, matrix))]
    return sorted_matrix
```

Table 11: A prompt example used for generating exemplar codes for the MBPP dataset. A more detailed prompt may increase the quality of the exemplar code and we leave this as our future work.

What is the main purpose of the fifth python code snippet? Summarize in one sentence and be concise. I will show you four examples first.

Code:

```
def odd_values_string(str):
    result = ""
    for i in range(len(str)):
        if i % 2 == 0:
            result = result + str[i]
    return result
```

Purpose: Write a python function to remove the characters which have odd index values of a given string.

Code:

```
from collections import defaultdict
def max_aggregate(stdata):
    temp = defaultdict(int)
    for name, marks in stdata:
        temp[name] += marks
    return max(temp.items(), key=lambda x: x[1])
```

Purpose: Write a function to calculate the maximum aggregate from the list of tuples.

Prompt

Code:

```
def pos_count(list):
    pos_count= 0
    for num in list:
        if num >= 0:
            pos_count += 1
    return pos_count
```

Purpose: Write a python function to count positive numbers in a list.

Code:

```
import math
def volume_tetrahedron(num):
    volume = (num ** 3 / (6 * math.sqrt(2)))
    return round(volume, 2)
```

Description: Write a function to calculate volume of a tetrahedron.

Code:

```
def sort_matrix(M):
    result = sorted(M, key=sum)
    return result
```

Purpose:

Output | Write a function to sort a matrix (list of lists) based on the sum of each inner list.

Table 12: A prompt example used for summarizing the original codes for the MBPP dataset. A more detailed prompt may increase the quality of the rewritten code and we leave this as our future work.