Bias and Debias in Code Search: Improve Search Results beyond Overall Ranking Performance

Anonymous EMNLP submission

Abstract

Code search engine is an essential tool in software development. Many code search methods have sprung up, focusing on the overall ranking performance of code search. In this paper, we study code search from another perspective by analyzing the bias of code search models. Biased code search engines provide poor user experience, even though they show promising overall performance. Due to different development conventions (e.g., prefer long queries or abbreviations), some programmers will find the engine useful, while others may find it hard to get desirable search results. To mitigate biases, we develop a general debiasing framework that employs reranking to calibrate search results. It can be easily plugged into existing engines and handle new code search biases discovered in the future. Experiments show that our framework can effectively reduce biases. Meanwhile, the overall ranking performance of code search gets improved after debiasing. Our implementation is available at: `https://anonymous.4open.science/r/CodeSearchBias-B2AD`.

1 Introduction

Software development is a repetitive task as programmers usually reuse or get inspiration from existing implementations. Studies show programmers spent 19% of their programming time on searching source code (Brandt et al., 2009). Therefore, code search, which refers to the retrieval of relevant code snippets from a codebase according to programmer's intent that has been expressed as a query (Liu et al., 2022), has become increasing important (Grazia and Pradel, 2022).

Although much effort has been devoted to improving code search, existing works mostly emphasize the ranking performance of code search w.r.t. metrics like Mean Reciprocal Rank (MRR) and Hit Ratio@K (HR@K) (Liu et al., 2022; Grazia and Pradel, 2022). In this paper, we study code search from another perspective. We find that state-of-the-art code search methods prevalently have discriminatory behaviors (i.e., different performance) toward queries or code snippets with certain properties (e.g., length). The observation shows, even though the overall ranking performance is good, programmers may still be dissatisfied with search results when their input queries or desired code snippets fall into those categories that code search models cannot handle well. We name our observation as *Code Search Bias*, inspired by the AI bias that attracts great attention recently (Mehrabi et al., 2021). Code search bias hurts user experience. Due to different development conventions (e.g., prefer long queries or abbreviations), users (programmers) of code search engines with biases will have different user experience, i.e., some users will find the engine useful, while others may find it hard to get desirable search results.

Note that most studies of bias in NLP focus on societal bias (Blodgett et al., 2020). For example, the gender bias of NLP algorithms may pose the danger of giving preference to male applicants in automatic resume filtering systems (Sun et al., 2019). However, in applications like search engines (Ovaisi et al., 2020) and recommender systems (Chen et al., 2023), some biases without societal factors are widely studied as they make the system biased toward certain search results and harm the performance. For instance, position bias exists in learning-to-rank systems where top search results are more likely to be clicked even if they are not the most relevant results (Agarwal et al., 2019). But it does not mean any discriminatory behaviors toward certain groups of people. Similarly, code search bias does not involve societal factors.

Considering that our observation has revealed the widespread code search bias in existing models, we aim at designing a general debiasing framework that can be easily plugged into existing code search engines. In the context of code search bias,

debiasing indicates removing the correlations between code search quality and certain properties of queries and code snippets. Our proposed debiasing framework adopts the idea of reranking to calibrate search results. It helps state-of-the-art code search models overcome code search bias and their overall performance can be improved at the meantime. In summary, our contributions are:

1. To our best knowledge, we are the first to study code search bias. We reveal the widespread existence of seven code search biases.
2. To mitigate code search bias, we propose a general debiasing framework using reranking. It can be easily plugged into existing engines.
3. Extensive experiments show that our debiasing framework not only helps alleviate code search bias but also improves the overall ranking performance of state-of-the-art code search models.

2 Related Work

Code Search. Early code search methods adopt traditional information retrieval methods to estimate the relevance between the query and a code snippet (Lv et al., 2015; Bajracharya et al., 2010). Recent works adopt deep neural networks to embed query and code into vectors. Then, the code search task is performed by measuring the similarity between vectors. Along this direction, various deep learning based methods have been proposed, including but not limited to recurrent neural network (RNN) based approaches (Gu et al., 2018), convolutional neural network (CNN) based approaches (Li et al., 2020), graph neural network (GNN) based approaches (Wan et al., 2019) and pre-training approaches (Feng et al., 2020; Guo et al., 2021, 2022).

Bias and Debias. Many AI systems exhibit certain biases that bring unfairness and degrade the performance (Mehrabi et al., 2021). Various debiasing methods have been proposed and they can be roughly divided into three types:

1. **Pre-processing methods** remove biases in training data. Calmon et al. (2017) design a framework for discrimination-preventing preprocessing to enhance data with multi goals. Biswas and Rajan (2021) analyze bias prompts in data preprocessing pipelines and identify data transformers that can mitigate the pipeline bias.
2. **In-processing methods** mitigate biases in the model training step. Garimella et al. (2021) propose a debiasing method that requires pretraining on an extra small corpus with bias mitigation objectives for mitigating social biases in language models. Lin et al. (2021) propose a debiasing framework with three strategies that be used as regularizers in the training objective of review-based recommender systems.
3. **Post-processing methods** handle biases after model training. Petersen et al. (2021) translate debiasing into a graph smoothing problem and propose a post-processing coordinate descent algorithm. Kim et al. (2019) design Multiaccuracy Boost, which uses an auditor to identify subpopulation biases and further uses it for debiasing in the post-processing.

Although many debiasing methods exist, they cannot be directly used for code search biases. Our method belongs to the post-processing category and it is tailored for removing code search biases.

3 Analysis of Code Search Biases

In this section, we reveal the widespread existence of code search biases and provide the analysis.

3.1 Analysis Settings

Data: We use CoSQA dataset[1] (Huang et al., 2021) with 20,604 query-code pairs. Each query is written in English while each code snippet is a Python code snippet. The data is annotated by at least 3 human annotators. We randomly split the dataset by 70%/30% for training and test. We adopt byte-pair encoding tokenization, a standard tokenization method used in preprocessing code search data, to tokenize queries and code snippets. As queries are typically short, stop words in queries are not removed. Note that there are other public code search datasets, e.g., CodeSearchNet dataset (Husain et al., 2019), DeepCS dataset (Gu et al., 2018), and CodeXGLUE dataset (Lu et al., 2021). We choose CoSQA dataset as it includes real code search queries, while other datasets use code documents (e.g., the first sentence in the function comments) to mimic queries. Using CoSQA helps us better discover biases in a real code search scenario.

Code Search Models: We select six representative code search approaches in the literature for our bias analysis, including DeepCS[2] (Gu

[1] https://github.com/microsoft/CodeXGLUE/tree/main/Text-Code/NL-code-search-WebQuery

[2] https://github.com/guxd/deep-code-search

et al., 2018), CQIL[3] (Li et al., 2020), CodeBERT[4] (Feng et al., 2020), CoCLR[5] (Huang et al., 2021), GraphCodeBERT[4] (Guo et al., 2021) and UniXcoder[4] (Guo et al., 2022). They are all under the MIT license, allowing us to adopt them in this study. We have observed similar biases in all the six methods. Due to space limitation, we only show analysis results of CQIL, CodeBert and GraphCodeBERT, and other methods are reported in our debiasing experiments in Sec. 5. We follow authors' descriptions to set hyper-parameters whenever possible in order to tune the performance of each method towards its best.

Evaluation Metrics: We use Mean Reciprocal Rank (MRR), the most widely used measure for code search, to illustrate our bias analysis. It is defined as MRR $= \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i}$, where $|Q|$ is the number of queries and $rank_i$ indicates the rank of the ground-truth code snippet w.r.t. the i-th query. We also adopt another prevalent metric Hit Ratio@K (HR@K, the percentage of ground-truth code snippets that are in the top-K ranking lists from code search models) and results are discussed in Sec. 5. Note that most current code search studies assume that there exists only one good result for each query and public code search datasets are designed this way. Hence, the popular ranking metric Normalized Discounted Cumulative Gain (NDCG) will be consistent with MRR. Our reported results are averaged over several runs.

3.2 Analysis Results

Based on the characteristics of code search and the data involved in the search process, we have found and verified seven code search biases. A general motivation to consider these seven factors is that they are commonly adopted as parameters in the experiments of existing papers as they affect the results of code-related tasks (Hu et al., 2022; Wan et al., 2018; McBurney and McMillan, 2016). The performance of CQIL, CodeBERT and GraphCodeBERT w.r.t. the seven biases are presented in Fig. 1. We first group queries (in the test set) or ground-truth code snippets in intervals with equal lengths w.r.t. certain statistics. Then, we investigate whether code search models show different behaviors towards different intervals. The x-axis illustrates the intervals. To better visualize the result of bias analysis, data in Fig. 1 (a) and (c) is grouped in an interval with a length of 4, data in Fig. 1 (f) is grouped in an interval with a length of 0.15, and data in other subfigures is grouped in an interval with a length of 1. The left y-axis denotes the number of queries or ground-truth code snippets in each interval while the right y-axis shows the average MRR score for data in each interval. We provide our analysis as follows:

Bias 1 w.r.t. Lengths of Ground-Truth Code

Length bias (i.e., model makes decisions based on or affected by the length of texts) has been verified in various information retrieval and natural language processing tasks such as textual matching (Jiang et al., 2022) and machine translation (Murray and Chiang, 2018). This inspires us to investigate the effect of the length of ground-truth code snippets on code search models.

Fig. 1 (a) shows the performance of three models w.r.t. code lengths. From Fig. 1 (a), we can see that lengths of most code snippets are between 20 and 50. Furthermore, we can observe that: (1) In general, the longer the ground-truth code snippet is, the better the MRR score is. There are some sharp drops in MRR when code length gets much longer. The reason may be the number of ground-truth code snippets in intervals with longer lengths (e.g., > 70) is quite small and a few hard cases affect the average performance in those intervals. (2) Code search models show a clear bias towards intervals with longer lengths of ground-truth code snippets, i.e., longer ground-truth code snippets are more easily to match. For instance, the MRR scores of GraphCodeBERT are 0.57 and 0.83 for the interval with average code length 36 and the interval with average code length 68, respectively. Intuitively, longer ground-truth code snippets provide more semantic information, making it more easy to be modeled and matched. From a software engineering perspective, long code snippets are more distinctive than short ones: it is more likely for two short code snippets to be similar, making it hard to distinguish the correct one from other candidates.

Bias 2 w.r.t. Lengths of Queries

Similar to Bias 1, we have identified the bias w.r.t. lengths of input queries. As shown in Fig. 1 (b), as query length increases, MRR decreases, indicating that longer queries have worse search results.

Bias 3 w.r.t. Numbers of AST Nodes

One major difference between natural languages

[3] https://github.com/flyboss/CQIL
[4] https://github.com/microsoft/CodeBERT
[5] https://github.com/Jun-jie-Huang/CoCLR

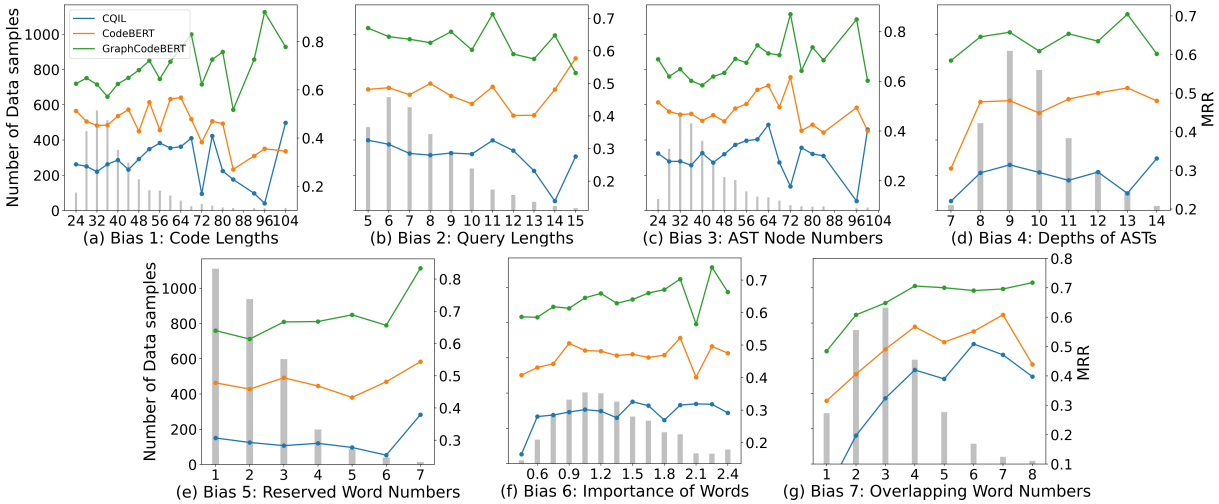

Figure 1: Examples of code search biases.

(NLs) and programming languages (PLs) is that PLs have strict syntax rules that are enforced by language grammars. Abstract Syntax Tree (AST), used in compilers, represents the abstract syntactic structure of the source code. Each node of ASTs denotes a construct or symbol occurring in the source code. Compared to plain source code, ASTs are abstract and some details (e.g., punctuation and delimiters) are not included. ASTs are used in various code-related tasks like code summarization (Alon et al., 2019) and code completion (Wang and Li, 2021) for capturing syntactic information.

Considering the importance of ASTs for modeling PL syntax, we investigate the influence of ASTs on code search models. Usually, longer code snippets correspond to deep ASTs. However, some complex yet short code snippets such as list parsing in Python may also have deep ASTs. Hence, Bias 3 is not equivalent to Bias 1. Fig. 1 (c) demonstrates the impacts of AST node numbers on the performance of code search models. We can observe the bias: code search models show diverse performance towards different intervals. For example, the MRR scores of GraphCodeBERT are 0.6 and 0.87 for the interval with average AST node number 40 and the interval with average AST node number 72, respectively. The performance gap is significant in code search.

Bias 4 w.r.t. Depths of ASTs

Similar to Bias 3, we further identify the bias w.r.t. AST depths which also depict the complexity of ASTs. Note a deep AST may not have many AST nodes. Hence, Bias 3 and Bias 4 are different. Fig. 1 (d) shows the impact of AST depths. In Fig. 1 (d), code snippets are grouped by the depth of their ASTs and the interval length is 1. We can observe the existence of bias: code search models have diverse performance towards different intervals containing ASTs with different depths.

Bias 5 w.r.t. Numbers of Reserved Words

If we do not consider identifiers and constants, the vocabulary of code tokens containing reserved words of a PL is small. We investigate the impact of reserved words on the behaviors of code search models. Specially, we consider Python reserved words `if`, `for`, `while`, `with`, `try` and `except`. They are related to control structures and demonstrate the programming logic of designing a function. Fig. 1 (e) demonstrates the performance towards ground-truth code snippets containing different numbers of reserved keywords. We can see the existence of a bias: performance of code search models varies when the number of code keywords changes. We can observe that the considerable growth of the MRR score when the number of keywords in ground-truth code snippets increases. One possible reason is that logic-related reserved words in ground-truth code snippets help code search models better capture the logic of the code. Therefore, it is easier for code search models to match the ground-truth code snippet and the user intent that manifests in the queries when code contains more logic-related reserved words.

Bias 6 w.r.t. Importance of Words

Queries are typically concise, containing only a few words. For each query, we calculate the max TF-IDF values for the words contained in the query to estimate how important words contained in a query are. We have also calculated the average and the minimum TF-IDF values and similar results can

be observed. TF-IDF helps avoid amplifying the importance of words that appear more frequently in general (e.g., the word "an" in a query "sort an array"). When calculating TF-IDF, we treat each query in CoSQA as a document. Results are presented in Figs. 1 (f), and we can observe the existence of a bias, i.e., code search models show different performance for queries containing words with varying importance. Intuitively, the important words (e.g., "sort") contained in a query help code search models better understand user intent and match the ground-truth code snippet.

Bias 7 w.r.t. Numbers of Overlapping Words

Early code search methods rely on the overlapping words of queries and code snippets to estimate query-code relevance scores. However, overlapping words received less attention in deep learning based code search models (Zhu et al., 2020). We investigate the influence of overlaps on the behaviors of the three code search models which all leverage deep learning. Fig. 1 (g) illustrates the performance on test query-code pairs that have different numbers of overlapping words. From the figure, we can observe a bias: models produce better MRR towards query-code pairs with more overlapping words. In other words, deep learning-based code search models also capture overlapping words and treat them as a strong signal of a matching result, confirming the standard hypothesis that overlapping words affect code search.

In summary, we have identified seven distinct biases, meaning that code search models show different performance when facing input queries or ground-truth code snippets with different characteristics. In practice, code search biases result in the inconsistence of user experience: depending on the characteristics of queries and/or ground-truth code snippets, the quality of search results varies.

4 Mitigate Code Search Biases

In this section, we illustrate our debiasing framework shown in Fig. 2. Our goal is to design a *general* framework: (1) it can be easily plugged into existing code search models without much additional effort, and (2) it can handle new code search biases that are not discovered at the moment.

We opt to adopt *reranking*, a post-processing method, to calibrate code search results. The idea is to rerank the ranking results provided by code search models. Even though code search biases are prevalent in many cases as we have seen in

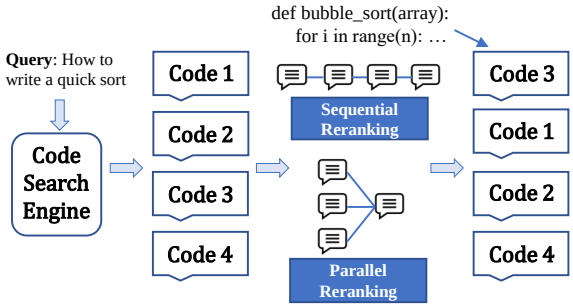

Figure 2: Overview of the debiasing framework.

Fig. 1, many code search models show promising overall performance (i.e., high MRR or HR@K). Therefore, for biased cases, the ground-truth code snippets are not too far away from the top of search results. Otherwise, the overall MRR scores will be quite low according to its definition. Similarly, we believe that any new code search biases also meet the above condition (i.e., biases exist but overall search performance is high). For biased cases, a successful reranking method can help ground-truth code snippets emerge on top. Post-processing search results also avoid modifying existing code search models. This way, the designed debiasing framework is orthogonal to a specific code search method and it can be easily used as a reinforcement.

Next, we first demonstrate how our framework mitigates one bias. Then, the way that our framework mitigates multiple biases is presented.

4.1 Mitigate A Single Bias via Reranking

Our idea is to use the prior knowledge of biased search from the training data to determinate whether a similar search in the test set will face a bias issue and require reranking. The detailed steps of mitigating a single bias via a single reranker are illustrated as follows:

1. Firstly, we embed all queries in the training set into vectors using a pre-trained CodeBERT model. For a test query (i.e., the current search), after it is embedded by the CodeBERT model, we retrieve its top-M most similar queries in the training set based on cosine similarity between vectors. These retrieved queries and their corresponding ground-truth code snippets in the training set will provide some hints on whether the current search may face a certain bias.

2. Then, we identify intervals in training data where code search models show *very high* performance. It is likely that search results are not severely biased within these intervals. Otherwise the MRR scores for these intervals should

be low by its definition. For such intervals, it is unnecessary to rerank for debiasing. We sort the search cases in training set by their MRR scores and retrieve cases with top $N\%$ maximum MRR scores. We adopt k-means to cluster the retrieved training search cases into S clusters. Then, the maximum and minimum MRR scores in each cluster are used as the boundaries of the cluster.

3. For a test search t, if its top-M most similar training query-code pairs have an average MRR score that falls in the range of any cluster, then it is likely that code search models provide reasonable relevance prediction scores for the candidate code snippets contained in these query-code pairs and our method will not rerank these candidate code snippets. For other candidate code snippets, reranking is required.

4. For a candidate code snippet c that requires reranking, the reranking score is calculated as:

$$R = \text{Score}_c^{\text{original}} + P(T_e < T_m), \qquad (1)$$

where $\text{Score}_c^{\text{original}}$ denotes the original ranking score of c, T_e represents the MRR value of the code search model on a training query-code pair, T_m represents the overall MRR value of the code search model on the training data, and $P(T_e < T_m)$ indicates the percentage of training query-code pairs that the code search model shows a lower MRR score than its overall MRR score over all the training pairs.

5. For the test search t, our method will use reranking scores R instead of $\text{Score}^{\text{original}}$ as relevance scores for all candidate code snippets that are identified to require reranking in Step 3. Then, the ranking list is reranked according to new relevance scores.

We discuss the impact of the choices of M, N and S in Analysis 5 of Sec. 5.

4.2 Mitigate Multiple Biases

To mitigate multiple code biases together, we adopt two simple yet effective strategies to assemble rerankers for different code search biases:

1. **Sequential Reranking**: Adopt each reranker sequentially. The relevance scores from a previous reranker will be used as the base relevance scores (i.e., $\text{Score}^{\text{original}}$) in the next reranker.

2. **Parallel Reranking**: Adopt each reranker parallel and use the average of the relevance scores from all rerankers between a candidate code snippet and the current search as the prediction.

Table 1: Examples for two reranking methods for assembling rerankers. "S" and "P" indicate sequential reranking and parallel reranking, respectively. R_1 and R_2 are reranking scores from reranker 1 and reranker 2, respectively.

Method	Code	Reranker 1	Reranker 2	Relevance Score
S	c1	$\text{Score}_{c1}^{\text{original}} + R_1$	$\text{Score}_{c1}^{\text{original}} + R_1 + R_2$	$\text{Score}_{c1}^{\text{original}} + R_1 + R_2$
	c2	$\text{Score}_{c2}^{\text{original}}$	$\text{Score}_{c2}^{\text{original}}$	$\text{Score}_{c2}^{\text{original}}$
P	c1	$\text{Score}_{c1}^{\text{original}} + R_1$	$\text{Score}_{c1}^{\text{original}} + R_2$	$\text{Score}_{c1}^{\text{original}} + (R_1 + R_2)/2$
	c2	$\text{Score}_{c2}^{\text{original}}$	$\text{Score}_{c2}^{\text{original}}$	$\text{Score}_{c2}^{\text{original}}$

Tab. 1 provides examples to illustrate relevance scores between a query and two candidate code snippets c_1 and c_2. From final relevance scores of the code snippet c_1, we can see that sequential reranking emphasizes the adjustment of reranking as it aggregates reranking terms from different rerankers. Differently, parallel reranking averages reranking terms from different rerankers, avoiding a sharp reranking. If none of the rerankers adjust the relevance score, then the final relevance scores are the same for both methods, as shown in the case of the code snippet c_2. Empirically, different ordering shows only slight performance difference, as we will show in Analysis 4 of Sec. 5.

Note the above two strategies in our debiasing framework looks similar to Boosting and Bagging methods used in Ensemble Learning (Zhou, 2009), but they are not the same: (1) Compared to Boosting methods like AdaBoost (Freund and Schapire, 1997), sequential reranking does not increase the weights for wrongly labeled training samples (biased/unbiased cases) in previous reranker since each reranker is designed for different targets (mitigate different biases) and wrongly labeled samples in the previous reranker may be correct samples for the next reranker. Differently, Boosting methods will increase weights of incorrectly predicted sampled for training the next learner. (2) Compared to Bagging methods (Breiman, 1996), parallel reranking does not adopt sampling to prepare different datasets (from the complete training set) for use in each reranker. The reason is that, to make our debiasing method simple and general, our reranking method is designed as a similarity-based adjuster with simple rules instead of a learning-based approach. In a large training set, most similar queries that are used to judge whether current search is facing bias may not be selected in sampling, which negatively affects debiasing.

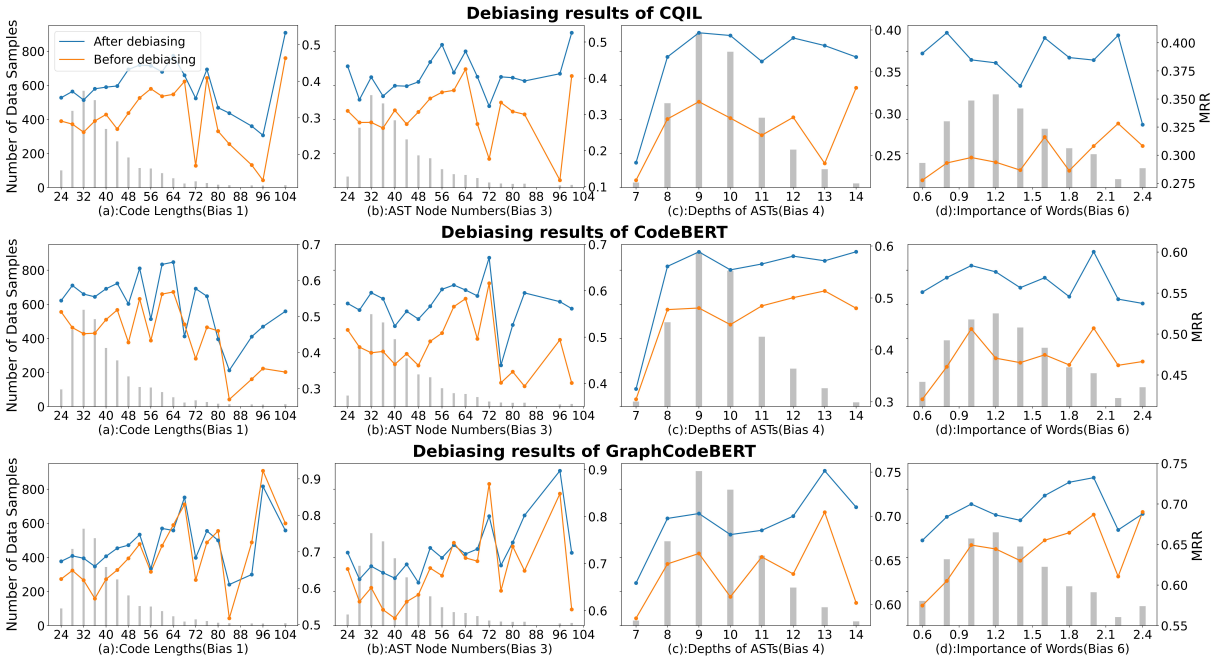

Figure 3: Mitigate biases using sequential reranking.

Table 2: Overall performance changes of code search models using sequential reranking.

Method Name	MRR		HR@1		HR@5		HR@10	
	Before	After	Before	After	Before	After	Before	After
DeepCS	0.295	0.428 (+45%)	0.219	0.366 (+67%)	0.375	0.489 (+30%)	0.462	0.553 (+20%)
CQIL	0.296	0.384 (+30%)	0.216	0.299 (+38%)	0.377	0.478 (+27%)	0.469	0.557 (+19%)
CodeBERT	0.474	0.569 (+20%)	0.363	0.471 (+30%)	0.598	0.685 (+15%)	0.712	0.782 (+9.8%)
CoCLR	0.756	0.770 (+1.9%)	0.641	0.661 (+3.1%)	0.909	0.917 (+0.88%)	0.967	0.971 (+0.41%)
GraphCodeBERT	0.641	0.695 (+8.4%)	0.524	0.587 (+12%)	0.790	0.831 (+5.2%)	0.882	0.911 (+3.3%)
UniXcoder	0.702	0.737 (+5.0%)	0.584	0.630 (+7.9%)	0.862	0.880 (+2.1%)	0.935	0.940 (+0.53%)

5 Debiasing Experiment

In this section, we will illustrate the effectiveness of our debiasing framework on mitigating code search biases. Results are reported using our framework to mitigate the seven biases for the six code search methods on the CoSQA dataset. By default, the order of rerankers in sequential reranking is Biases 7, 6, 3, 4, 2, 5 and 1. We also analyze the impact of reranker order in Analysis 4 of our experiments. Our method requires three hyper-parameters: M, N and S, as illustrated in Sec. 4.1. We search M, N and S in $\{1, 3, 5\}$, $\{10, 15, 20\}$ and $\{1, 3, 5\}$, respectively. Best results ($M = 1$, $N = 10$ and $S = 1$) are reported.

Analysis 1: Debiasing Results. We first analyze the results after debiasing. Due to space limitation, we only visualize results of Bias 1 (Lengths of Code), Bias 3 (Numbers of AST nodes), Bias 4 (Depths of ASTs) and Bias 6 (Importance of Words) for CQIL, CodeBERT and GraphCodeBERT. For other code search methods and biases, we observe similar results. Fig. 3 shows the performance before and after mitigating biases using sequential reranking. The result using parallel reranking is presented in Fig. 6 of Appendix B. From visualization results, we can clearly see that, for all the four biases, MRR scores of most intervals increase after deploying our debiasing framework, showing the effectiveness of our debiasing framework. Sequential reranking shows a slightly better debiasing result than parallel reranking (e.g., see CQIL(b) and GraphCodeBERT(b) in Fig. 3 and Fig. 6). However, sequential reranking is not as efficient as parallel reranking as it processes each reranker one by one.

Analysis 2: Changes of Code Search Performance after Debiasing. Tab. 2 illustrates the changes of overall code search performance after debiasing using sequential reranking. Due to page limitation, the results for parallel reranking is provided in Tab. 4 of Appendix C. From results, we can see that, after debiasing, overall code search performance w.r.t. MRR or HR@K significantly increases. The improvements are especially noticeable for DeepCS, CQIL and CodeBERT: MRR

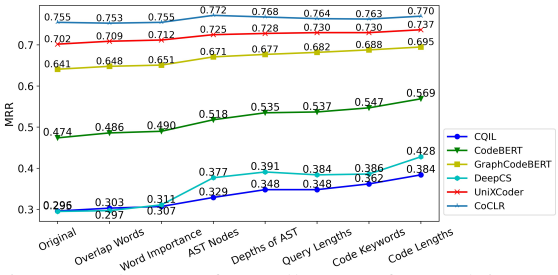

Figure 4: Changes of overall MRR after applying each reranker in sequential reranking.

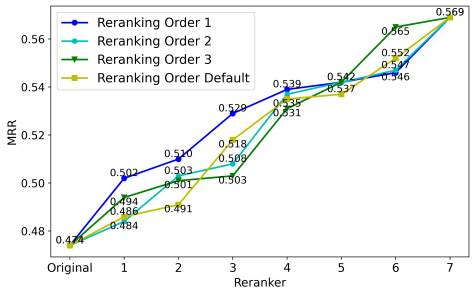

Figure 5: Sequential reranking in different order.

Table 3: MRR for different hyper-parameters.

Method	M			S			N		
	1	3	5	1	3	5	10	15	20
CQIL	0.380	0.329	0.348	0.384	0.380	0.355	0.380	0.380	0.376
CodeBERT	0.569	0.537	0.504	0.572	0.569	0.552	0.569	0.569	0.569
GraphCodeBERT	0.695	0.673	0.660	0.696	0.695	0.690	0.695	0.695	0.695

and HR@K of these methods increase by 9.6%-67%. The reason is that the original search performance of the three methods is not high and there is still large room for improvement. Even for CoCLR, GraphCodeBERT and UniXcoder which show quite high MRR (>0.6) and HR@K (>0.5) before debiasing, our debiasing framework still helps improve the overall code search performance. Thus, we can conclude that *mitigating code search bias has a positive effect on improving the overall code search performance*.

Analysis 3: Impacts of Applying Multiple Rerankers. Next, we investigate whether applying multiple rerankers brings better debiasing results than using a single reranker. Fig. 4 illustrates the changes of overall MRR scores for the six code search models after applying each reranker using sequential reranking in the default order. The horizontal axis labels (from left to right) show the order of rerankers applied. We can observe that MRR scores of CodeBERT, DeepCS and CQIL gradually increase as more rerankers are applied. Eventually, their overall performance after debiasing gets significantly improved compared to their original performance. For CoCLR, UniXCoder and GraphCodeBERT which have achieved high MRR scores before debiasing, applying multiple rerankers slightly enhances or does not negatively affect their overall performance. Overall, after applying seven rerankers, the performance of CoCLR, UniXCoder and GraphCodeBERT gets enhanced. We can observe a similar trend when using parallel reranking. In conclusion, *the more rerankers are applied, the better overall code search performance the code search model can achieve. In other words, each reranker indeed contributes to the improvement of the quality of code search results.*

Analysis 4: Impacts of Reranker Order in Sequential Reranking. Since sequential reranking has various possible order of rerankers, we analyze the impact of reranking order. In addition to the default order, we report the debiasing performance on CodeBERT using sequential reranking with three other orders: order 1 (biases 1, 6, 4, 5, 2, 7, 3), order 2 (bases 6, 2, 4, 7, 3, 5, 1) and order 3 (biases 4, 6, 2, 1, 5, 3, 7). Fig. 5 demonstrates the performance changes after each reranker is applied in the three order. The horizontal axis labels (from left to right) show rerankers in the applied order. Similar to the observation in Analysis 3, we can see that adding more rerankers help improve the MRR score. And the intermediate debiasing results are slightly different using three different order. But the different order does not affect the final debiasing result too much.

Analysis 5: Impacts of Hyper-parameters. We further analyze the impacts of hyper-parameters. Tab. 3 provides the debiasing results of CQIL, CodeBERT and GraphCodeBERT using different hyper-parameters. Each of the MRR score in the table is obtained by changing one hyper-parameter while keeping the other two hyper-parameters the same as the best ones found in hyper-parameter search. From the result, we can conclude that hyper-parameters do not affect results too much. We provide the detailed analysis in Appendix D.

Analysis 6: Human Evaluation. We also conduct human evaluation for assessing the quality of debiasing. The details are described in Appendix E.

6 Conclusion

In this paper, we reveal the existence of code search biases. We design a general debiasing framework that can be easily plugged into existing search models. In the future, we will explore pre-processing and in-processing methods to improve our framework and better mitigate code search biases.

7 Limitations

In this section, we discuss some limitations of this work:

- **Data:** When we submitted this manuscript, only one real code search dataset CoSQA was publicly available. Other datasets in the literature do not have real search queries, and they use code documents to simulate queries. However, code documents and queries have different text styles (i.e., length). Hence, we only study code search bias based on the real data in CoSQA. To overcome this limitation, we are constructing another dataset containing real code search queries and will release it for future study.
- **Language:** Queries and code snippets in CoSQA are written in English and Python, respectively. It is unclear whether our analysis results hold for queries written in other natural languages (e.g., French and Chinese). As the causes of code search biases analyzed in this work should be common across different programming languages (e.g., Java and Go), we expect that code search in other programming languages also suffers from the biases studied in this paper. We leave the study of the impacts of different natural languages and programming languages on code search bias as future work.

References

Aman Agarwal, Ivan Zaitsev, Xuanhui Wang, Cheng Li, Marc Najork, and Thorsten Joachims. 2019. Estimating position bias without intrusive interventions. In *WSDM*, pages 474–482.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. In *ICLR*.

Sushil Krishna Bajracharya, Joel Ossher, and Cristina Videira Lopes. 2010. Leveraging usage similarity for effective retrieval of examples in code repositories. In *SIGSOFT FSE*, pages 157–166.

Sumon Biswas and Hridesh Rajan. 2021. Fair preprocessing: towards understanding compositional fairness of data transformers in machine learning pipeline. In *ESEC/SIGSOFT FSE*, pages 981–993.

Su Lin Blodgett, Solon Barocas, Hal Daumé III, and Hanna M. Wallach. 2020. Language (technology) is power: A critical survey of "bias" in NLP. In *ACL*, pages 5454–5476.

Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *CHI*, pages 1589–1598.

Leo Breiman. 1996. Bagging predictors. *Mach. Learn.*, 24(2):123–140.

Flávio P. Calmon, Dennis Wei, Bhanukiran Vinzamuri, Karthikeyan Natesan Ramamurthy, and Ku sh R. Varshney. 2017. Optimized pre-processing for discrimination prevention. In *NIPS*, pages 3992–4001.

Jiawei Chen, Hande Dong, Xiang Wang, Fuli Feng, Meng Wang, and Xiangnan He. 2023. Bias and debias in recommender system: A survey and future directions. *ACM Trans. Inf. Syst.*, 41(3):1:1–1:39.

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 *EMNLP (Findings)*, pages 1536–1547.

Yoav Freund and Robert E. Schapire. 1997. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.*, 55(1):119–139.

Aparna Garimella, Akhash Amarnath, Kiran Kumar, Akash Pramod Yalla, Anandhavelu Natarajan, Niyati Chhaya, and Balaji Vasan Srinivasan. 2021. He is very intelligent, she is very beautiful? on mitigating social biases in language modelling and generation. In *ACL/IJCNLP (Findings)*, pages 4534–4545.

Luca Di Grazia and Michael Pradel. 2022. Code search: A survey of techniques for finding code. *arXiv Preprint*. https://arxiv.org/abs/2204.02765.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *ICSE*, 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 *ACL (1)*, 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 *ICLR*.

Fan Hu, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Xirong Li. 2022. Long code for code search. *arXiv Preprint*. https://arxiv.org/abs/2208.11271.

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 *ACL/IJCNLP (1)*, 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*. https://arxiv.org/abs/1909.09436.

Lan Jiang, Tianshu Lyu, Yankai Lin, Chong Meng, Xiaoyong Lyu, and Dawei Yin. 2022. On length divergence bias in textual matching models. In *ACL (Findings)*, pages 4187–4193.

Michael P. Kim, Amirata Ghorbani, and James Y. Zou. 2019. Multiaccuracy: Black-box post-processing for fairness in classification. In *AIES*, pages 247–254.

Wei Li, Haozhe Qin, Shuhan Yan, Beijun Shen, and Yuting Chen. 2020. Learning code-query interaction for enhancing code searches. In *ICSME*, pages 115–126.

Chen Lin, Xinyi Liu, Guipeng Xv, and Hui Li. 2021. Mitigating sentiment bias for recommender systems. In *SIGIR*, pages 31–40.

Chao Liu, Xin Xia, David Lo, Cuiyun Gao, Xiaohu Yang, and John C. Grundy. 2022. Opportunities and challenges in code search tools. *ACM Comput. Surv.*, 54(9):196:1–196:40.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *NeurIPS Datasets and Benchmarks*.

Fei Lv, Hongyu Zhang, Jian-Guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on API understanding and extended boolean model (E). In *ASE*, pages 260–270.

Paul W. McBurney and Collin McMillan. 2016. An empirical study of the textual similarity between source code and source code summaries. *Empir. Softw. Eng.*, 21(1):17–42.

Ninareh Mehrabi, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan. 2021. A survey on bias and fairness in machine learning. *ACM Comput. Surv.*, 54(6):115:1–115:35.

Kenton Murray and David Chiang. 2018. Correcting length bias in neural machine translation. In *WMT*, pages 212–223.

Zohreh Ovaisi, Ragib Ahsan, Yifan Zhang, Kathryn Vasilaky, and Elena Zheleva. 2020. Correcting for selection bias in learning-to-rank systems. In *WWW*, pages 1863–1873.

Felix Petersen, Debarghya Mukherjee, Yuekai Sun, and Mikhail Yurochkin. 2021. Post-processing for individual fairness. In *NeurIPS*, pages 25944–25955.

Tony Sun, Andrew Gaut, Shirlyn Tang, Yuxin Huang, Mai ElSherief, Jieyu Zhao, Diba Mirza, Elizabeth M. Belding, Kai-Wei Chang, and William Yang Wang. 2019. Mitigating gender bias in natural language processing: Literature review. In *ACL (1)*, pages 1630–1640.

Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. 2019. Multi-modal attention network learning for semantic source code retrieval. In *ASE*, pages 13–25.

Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *ASE*, pages 397–407.

Yanlin Wang and Hui Li. 2021. Code completion by modeling flattened abstract syntax trees as graphs. In *AAAI*, pages 14015–14023.

Zhi-Hua Zhou. 2009. Ensemble learning. In *Encyclopedia of Biometrics*, pages 270–273.

Qihao Zhu, Zeyu Sun, Xiran Liang, Yingfei Xiong, and Lu Zhang. 2020. Ocor: An overlapping-aware code retriever. In *ASE*, pages 883–894.

A Hardware Used in Our Experiments

The experiments were run on a machine with two Intel(R) Xeon(R) CPU E5-2678 v3 @ 2.50GHz, 256 GB main memory and 8 GeForce RTX 2080 Ti graphics cards with 11 GB memory per card.

B Additional Results for Analysis 1

As additional experimental results for Analysis 1 in Sec. 5, Fig. 6 illustrates the performance before and after mitigating biases using using parallel reranking.

C Additional Results for Analysis 2

As additional experimental results for Analysis 2 in Sec. 5, Tab. 4 illustrates the changes of overall code search performance after debiasing using parallel reranking.

D Additional Discussions for Analysis 5

From the result in Tab. 3, we can conclude that hyper-parameters do not affect results too much. We provide the analysis as follows:

- M indicates how many top-M similar queries in the training set are adopted. We believe the top-1 similar query already provides a hint for our method, and including more similar queries do not bring more information. Hence, changing M does not affect the results too much.

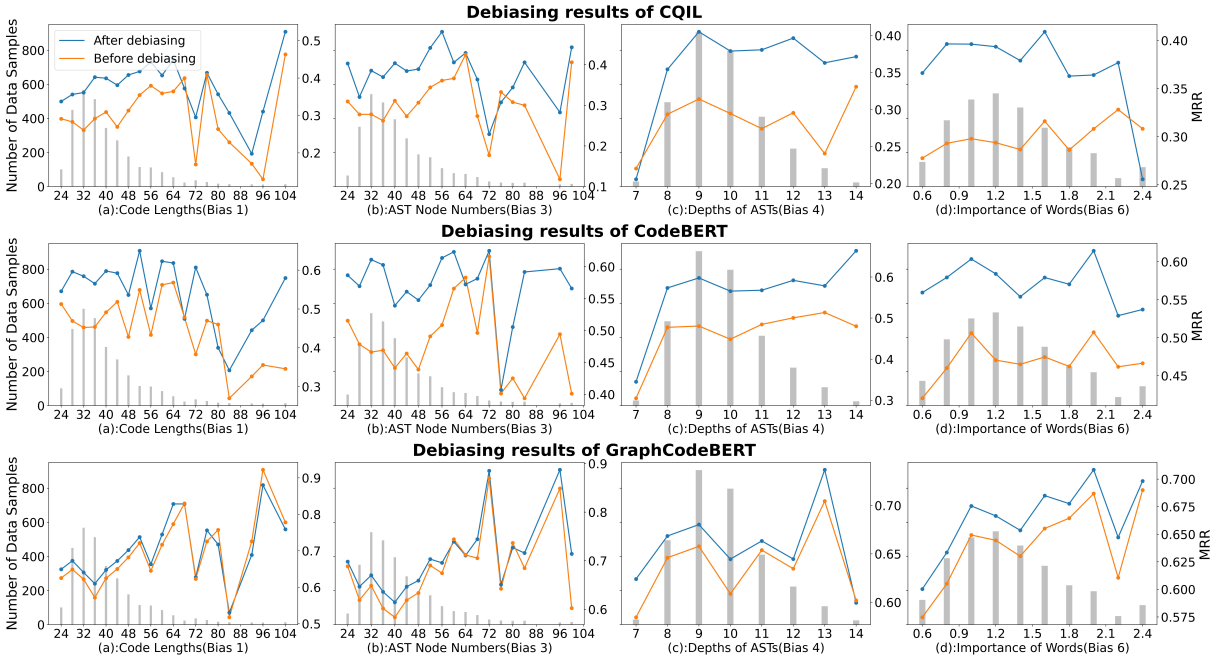

Figure 6: Mitigate biases using parallel reranking.

- $N\%$ represents the percentage of chosen training search cases with the highest MRR scores. Since the training set of CoSQA data contains 14K query-code pairs, changing $N\%$ in $\{10\%, 15\%, 20\%\}$ results in 1,400, 2,100 and 2,800 retrieved cases, respectively. The difference between the numbers of retrieved cases is not large, compared to the total dataset with 21K query-code pairs.

- S indicates the number of clusters after performing kmeans on these $N\%$ cases. We find that small values of S bring relatively robust and good performance of debiasing, as reported in Tab. 3. Therefore, we suggest that users set S to a small value. If we set S to a much larger number (e.g., 100, 500, 1,000), the performance becomes inconsistent, and we suspect that dividing retrieved cases into many small clusters cannot help find case patterns. Instead, many small clusters bring the noise. Hence, we do not suggest that users set S to a large value.

E Details of Human Evaluation

We randomly pick 200 queries from the test set for human evaluation. We choose CQIL as a representation of code search models and use it in human evaluation. We use our debiasing framework to reduce code search biases in the corresponding results of CQIL for the 200 queries. We recruit four master students majoring in computer science to check the quality of debiasing manually. For each query, we provide the students with two lists. One is the original top-10 search results from CQIL, and the other is the top-10 list after debiasing. The lists for each query are shown in random order. Students are asked to choose which top-10 list is better, and they can also indicate that the two lists are roughly of the same quality. From the results of human evaluation, we find that, for 71.5% queries, lists after debiasing are assessed as better ones. For 19.5% queries, the original list and the reranked list are estimated as having similar quality. For the remaining 9% queries, debiasing degrades the quality of the search list. The human evaluation results illustrate that our debiasing method indeed improves the quality of the code search for most queries. The materials of human evaluation are included in our provided repository.

Table 4: Overall performance changes of code search models using parallel reranking.

Method Name	MRR		HR@1	
	Before	After	Before	After
DeepCS	0.295	0.425 (+44%)	0.219	0.363 (+65%)
CQIL	0.296	0.383 (+29%)	0.216	0.300 (+39%)
CodeBERT	0.474	0.579 (+22%)	0.363	0.483 (+33%)
CoCLR	0.756	0.769 (+1.7%)	0.641	0.661 (+3.1%)
GraphCodeBERT	0.641	0.666 (+3.9%)	0.524	0.552 (+5.3%)
UniXcoder	0.702	0.716 (+2.0%)	0.584	0.602 (+3.1%)
Method Name	**HR@5**		**HR@10**	
	Before	After	Before	After
DeepCS	0.375	0.485 (+29%)	0.462	0.551 (+19%)
CQIL	0.377	0.476 (+26%)	0.469	0.551 (+17%)
CodeBERT	0.598	0.694 (+16%)	0.712	0.780 (+9.6%)
CoCLR	0.909	0.915 (0.66%)	0.967	0.971 (+0.41%)
GraphCodeBERT	0.790	0.810 (+2.5%)	0.882	0.895 (+1.5%)
UniXcoder	0.862	0.872 (+1.2%)	0.935	0.939 (+0.43%)