CoSQA: 20,000+ Web Queries for Code Search and Question Answering

Junjie Huang[1*], **Duyu Tang**[2], **Linjun Shou**[3], **Ming Gong**[3],
Ke Xu[1], **Daxin Jiang**[3], **Ming Zhou**[2], **Nan Duan**[2]
[1]Beihang University [2]Microsoft Research Asia [3]Microsoft STC Asia
[1]huangjunjie@buaa.edu.cn, kexu@nlsde.buaa.edu.cn
[2,3]{dutang,lisho,migon,djiang,nanduan}@microsoft.com

Abstract

Finding codes given natural language query is beneficial to the productivity of software developers. Future progress towards better semantic matching between query and code requires richer supervised training resources. To remedy this, we introduce the CoSQA dataset. It includes 20,604 labels for pairs of natural language queries and codes, each annotated by at least 3 human annotators. We further introduce a contrastive learning method dubbed CoCLR to enhance query-code matching, which works as a data augmenter to bring more artificially generated training instances. We show that evaluated on CodeXGLUE with the same CodeBERT model, training on CoSQA improves the accuracy of code question answering by 5.1%, and incorporating CoCLR brings a further improvement of 10.5%. [1].

Example 1:
Query: python check if path is absolute path or relative path
Code:

```python
def is_relative_url(url):
    """simple method to determine if a url is relative or absolute"""
    if url.startswith("#"):
        return None
    if url.find("://") > 0 or url.startswith("//"):
        # either 'http(s)://...' or '//cdn...' and therefore absolute
        return False
    return True
```

Label: 1

Example 2:
Query: capitalize letters in string python
Code:

```python
def snake_to_camel(s: str) -> str:
    """ Convert string from snake case to camel case. """
    fragments = s.split('_')
    return fragments[0] + ''.join(x.title() for x in fragments[1:])
```

Label: 0

Figure 1: Two examples in CoSQA. A pair of a web query and a Python function with documentation is annotated with "1" or "0", representing whether the code answers the query or not.

1 Introduction

With the growing population of software developers, natural language code search, which improves the productivity of the development process via retrieving semantically relevant code given natural language queries, is increasingly important in both communities of software engineering and natural language processing (Allamanis et al., 2018; Liu et al., 2020a). The key challenge is how to effectively measure the semantic similarity between a natural language query and a code.

There are recent attempts to utilize deep neural networks (Gu et al., 2018; Wan et al., 2019; Feng et al., 2020), which embed query and code as dense vectors to perform semantic matching in a unified vector space. However, these models are mostly trained on pseudo datasets in which a natural language query is either the documentation of a function or a tedious question from Stack Overflow. Such pseudo queries do not reflect the distribution of real user queries that are frequently issued in search engines. To the best of our knowledge, datasets that contain real user web queries include Lv et al. (2015), CodeSearchNet Challenge (Husain et al., 2019), and CodeXGLUE [2] (Lu et al., 2021). These three datasets only have 34, 99, and 1,046 queries, respectively, for model testing. The area lacks a dataset with a large amount of real user queries to support the learning of statistical models like deep neural networks for matching the semantics between natural language web query and code.

To address the aforementioned problems, we introduce CoSQA, a dataset with 20,604 pairs of web queries and code for **Co**de **S**earch and **Q**uestion **A**nswering, each with a label indicating whether

[*]Work done during internship at Microsoft Research Asia.
[1]The CoSQA data and leaderboard are available at https://github.com/microsoft/CodeXGLUE/tree/main/Text-Code/NL-code-search-WebQuery. The code is available at https://github.com/Jun-jie-Huang/CoCLR

[2]https://github.com/microsoft/CodeXGLUE

Dataset	Size	Natural Language	Code	human-annotated ?
CodeSearchNet (Husain et al., 2019)	2.3M	Documentation	Function	No
Gu et al. (2018)	18.2M	Documentation	Function	No
Miceli Barone and Sennrich (2017)	150.4K	Documentation	Function	No
StaQC (manual) (Yao et al., 2018)	8.5K	Stack Overflow question	Code block	Yes
StaQC (auto) (Yao et al., 2018)	268K	Stack Overflow question	Code block	No
CoNaLa (manual) (Yin et al., 2018)	2.9K	Stack Overflow question	Statements	Yes
CoNaLa (auto) (Yin et al., 2018)	598.2K	Stack Overflow question	Statements	No
SO-DS (Heyman and Cutsem, 2020)	12.1K	Stack Overflow question	Code block	No
Nie et al. (2016)	312.9K	Stack Overflow question	Code block	No
Li et al. (2019)	287	Stack Overflow question	Function	Yes
Yan et al. (2020)	52	Stack Overflow question	Function	Yes
Lv et al. (2015)	34	Web query	Function	Yes
CodeSearchNet (Husain et al., 2019)	99	Web query	Function	Yes
CodeXGLUE WebQueryTest [2]	1K	Web query	Function	Yes
CoSQA (ours)	20.6K	Web query	Function	Yes

Table 1: Overview of existing datasets on code search and code question answering. Some datasets containing both unlabelled data and labelled data are listed in separate lines.

the code can answer the query or not. The queries come from the search logs of the Microsoft Bing search engine, and the code is a function from GitHub[3]. To scale up the annotation process on such a professional task, we elaborately curate potential positive candidate pairs and perform large scale annotation where each pair is annotated by at least three crowd-sourcing workers. Furthermore, to better leverage the CoSQA dataset for query-code matching, we propose a code contrastive learning method (CoCLR) to produce more artificially generated instances for training.

We perform experiments on the task of query-code matching on two tasks: code question answering and code search. On code question answering, we find that the performance of the same CodeBERT model improves 5.1% after training on the CoSQA dataset, and further boosts 10.5% after incorporating our CoCLR method. Moreover, experiments on code search also demonstrate similar results.

2 Related Work

In this part, we describe existing datasets and methods on code search and code question answering.

2.1 Datasets

A number of open-sourced datasets with a large amount of text-code pairs have been proposed for the purposes of code search (Husain et al., 2019; Gu et al., 2018; Nie et al., 2016) and code question answering (Yao et al., 2018; Yin et al., 2018; Heyman and Cutsem, 2020). There are also high-quality but small scale testing sets curated for code search evaluation (Li et al., 2019; Yan et al., 2020; Lv et al., 2015). Husain et al. (2019), Gu et al. (2018) and Miceli Barone and Sennrich (2017) collect large-scale unlabelled text-code pairs by leveraging human-leaved comments in code functions from GitHub. Yao et al. (2018) and Yin et al. (2018) automatically mine massive code answers for Stack Overflow questions with a model trained on a human-annotated dataset. Nie et al. (2016) extract the Stack Overflow questions and answers with most likes to form text-code pairs. Among all text-code datasets, only those in Lv et al. (2015), CodeSearchNet Challenge (Husain et al., 2019) and CodeXGLUE[2] contain real user web queries, but they only have 34, 99 and 1,046 queries for testing and do not support training data-driven models. Table 1 illustrates an overview of these datasets.

2.2 Code Search Models

Models for code search mainly can be divided into two categories: information retrieval based models and deep learning based models. Information retrieval based models match keywords in the query with code sequence (Bajracharya et al., 2006; Liu et al., 2020b). Keyword extension by query expansion and reformulation is an effective way to enhance the performance (Lv et al., 2015; Lu et al., 2015; Nie et al., 2016; Rahman et al., 2019; Rahman, 2019). deep learning based models encode query and code into vectors and utilize vector similarities as the metric to retrieve code (Sachdev et al., 2018; Ye et al., 2016; Gu et al., 2018; Cambronero

[3]We study on Python in this work, and we plan to extend to more programming languages in the future.

et al., 2019; Yao et al., 2019; Liu et al., 2019a; Feng et al., 2020; Zhao and Sun, 2020). There are also ways to exploit code structures to learn better representations for code search (Wan et al., 2019; Haldar et al., 2020; Guo et al., 2020).

3 CoSQA Dataset

In this section, we introduce the construction of the CoSQA dataset. We study Python in this work, and we plan to extend to more programming languages in the future. Each instance in CoSQA is a pair of natural language query and code, which is annotated with "1" or "0" to indicate whether the code can answer the query. We first describe how to curate web queries, obtain code functions, and get candidate query-code pairs. After that, we present the annotation guidelines and statistics.

3.1 Data Collection

Query Curation We use the search logs from the Microsoft Bing search engine as the source of queries. Queries without the keyword "*python*" are removed. Based on our observation and previous work (Yao et al., 2018; Yan et al., 2020), there are seven basic categories of code-related web queries, including: (1) code searching, (2) debugging, (3) conceptual queries, (4) tools usage, (5) programming knowledge, (6) vague queries and (7) others. Basically, queries in (2)-(7) categories are not likely to be answered only by a code function, since they may need abstract and general explanations in natural language. Therefore, we only target the first category of web queries that have code searching intent, i.e., queries that can be answered by a piece of code.

To filter out queries without code searching intent, we manually design heuristic rules based on exact keyword matching. For example, queries with the word of *benefit* or *difference* are likely to seek a conceptual comparison rather than a code function, so we remove all queries with such keywords. Based on the observations, we manually collect more than 100 keywords in total. Table 2 displays a part of selected keywords used for removing unqualified queries and more details can be found in Appendix A. To evaluate the query filtering algorithm, we construct a human-annotated testset. We invite three experienced python programmers to label 250 randomly sampled web queries with a binary label of having/not having searching intent. Then we evaluate the accuracy of intent predictions given keyword-based rules and those given by humans. We find the F1 score achieves 67.65, and the accuracy is up to 82.40. This demonstrates the remarkable effectiveness of our rule-based query filtering algorithm.

Categories	Some Keywords
Debugging	exception, index out of, ignore, stderr, ...
Conceptual Queries	vs, versus, difference, advantage, benefit, drawback, how many, what if, why, ...
Programming Knowledge	tutorial, advice, argument, suggestion, statement, declaration, operator, ...
Tools Usage	console, terminal, open python, studio, ide, ipython, jupyter, vscode, vim, ...
Others	unicode, python command, "@", "()", ...

Table 2: Selected keywords for our heuristic rules to filter out web queries without code search intent in five categories. Vague queries are morphologically variable, so we ignore this category.

Code Collection The selection of code format is another important issue in constructing query-code matching dataset, which includes a statement (Yin et al., 2018), a code snippet/block (Yao et al., 2018), a function (Husain et al., 2019), etc. In CoSQA, we simplify the task and adopt a compete Python function with paired documentation to be the answer to the query for the following reasons. First, it is complete and independent in functionality which may be more prone to answering a query. Second, it is syntactically correct and formally consistent which enables parsing syntax structures for advanced query-code matching. Additionally, a complete code function is often accompanied with documentation wrote by programmers to help understand its functionality and usage, which is beneficial for query-code matching (see Section 6.4 for more details).

We take the CodeSearchNet Corpus (Husain et al., 2019) as the source for code functions, which is a large-scale open-sourced code corpus allowing modification and redistribution. The corpus contains 2.3 million functions with documentation and 4.1 million functions without documentation from public GitHub repositories spanning six programming languages (Go, Java, JavaScript, PHP, Python, and Ruby). In CoSQA, we only keep complete Python functions with documentation and remove those with non-English documentation or special tokens (e.g. "$\langle img... \rangle$" or "$http://$").

Query	Code		Explanations
(1) boolean function to check if variable is a string python	`def is_string(val):` `""" Determines whether the passed value is a string, safe for 2/3. """` `try:` `basestring` `except NameError:` `return isinstance(val, str)` `return isinstance(val, basestring)`	☺	Code can fully satisfy the demand of the query. Therefore the code is a correct answer.
(2) python check if argument is list	`def is_listish(obj):` `""" Check if something quacks like a list. """` `if isinstance(obj, (list, tuple, set)):` `return True` `return is_sequence(obj)`	☺	Code meets the demand of checking *list* type, and the *tuple* and *set* types, which exceeds query's demand. It is a correct answer.
(3). python measure distance between 2 points	`def vector_distance(a, b):` `""" The Euclidean distance between two vectors. """` `a = np.array(a)` `b = np.array(b)` `return np.linalg.norm(a - b)`	☺	Code computes Euclidean distance, which is one category of vector distances. So it is correct.
(4) python measure distance between 2 points	`def dist_sq(self, other):` `""" Distance squared to some other point. """` `dx = self.x - other.x` `dy = self.y - other.y` `return dx**2 + dy**2`	☺	Code computes square distance, which is another category of vector distances.
(5) read write in the same file python	`def file_read(filename):` `""" Read a file and close it. Returns the file source. """` `fobj = open(filename,'r');` `source = fobj.read();` `fobj.close()` `return source`	☹	Query asks for reading and writing, but code only implements reading. The code satisfies 50% of the demands and is not a correct answer.
(6) python get the value in the list starting with the str	`def get_list_index(lst, index_or_name):` `""" Return the index of an element in the list.` `Args:` `lst (list): The list.` `index_or_name (int or str): Value of the reference element, or directly its numeric index.` `Returns: (int) Index of the element in the list. """` `if isinstance(index_or_name, six.integer_types):` `return index_or_name` `return lst.index(index_or_name)`	☹	The query is looking for an element in the list that starts with a specific *str*, but the code does not have the function of starting with the *str*, and it returns index instead of value. There are two unsatisfied areas, which is less than 50%.
(7) python check if something is an array	`def is_number(obj):` `""" Check if obj is number. """` `return isinstance(obj,(int,float,np.int_,np.float_))`	☹	A small part of code is relevent to the query but is can not answer.

Table 3: Examples and explanations of query-code pairs for correct and incorrect answers.

Candidate Query-code Pairs Obviously, it is not possible to annotate all query-code pairs. To improve efficiency, we wipe off low-confidence instances before annotation. Specifically, we employ a CodeBERT-based matching model (Feng et al., 2020) to retrieve high-confidence codes for every query. The CodeBERT encoder is fine-tuned on 148K automated-minded Python Stack Overflow question-code pairs (StaQC) (Yao et al., 2018) with the default parameters. A cosine similarity score on the pooled $[CLS]$ embeddings of query and code is computed to measure the relatedness. To guarantee the quality of candidates, we automatically remove low-quality query-code pairs according to the following evaluation metrics.

- To ensure the code may answer the query, we only keep the code with the highest similarity to the query and remove the pairs with a similarity below 0.5.
- To increase the code diversity and control the code frequency, we restrict the maximum occurrence of each code to be 10.

3.2 Data Annotation

Annotating such a domain-specific dataset is difficult since it requires the knowledge of Python. Even experienced programmers do not necessarily understand all code snippets. To ensure the feasibility and control annotation quality, we design comprehensive annotation guidelines and take a two-step annotation procedure.

Annotation Guidelines Our annotation guideline is developed through several pilots and further updated with hard cases as the annotation progresses. Annotation participants are asked to make a two-step judgment for each instance: intent annotation and answer annotation.

In the first step of *intent annotation*, annotators are asked to judge whether the query has the intent to search for a code. They will skip the second step if the query is without code search intent. As shown in Section 3.1, *vague queries* are hard to be filtered out by our heuristic intent filtering algorithm. Therefore, it is necessary to take this step to remove such queries so that we can focus more on the matching between query and code rather than query discrimination.

In the second step of *answer annotation*, annotators are asked to judge whether the code can answer the query. They should label the instance with "1" if the code is a *correct answer*; otherwise, it is labeled "0". In this step, judgment should be made after comprehensively considering the relevance between query with documentation, query with function header, and query with function body.

During annotation, it is often the case that a code function can completely answer the query, which means that the code can satisfy all the demands in the query and it is a correct answer. (Case (1) in Table 3.) But more often, the code can not completely answer the query. It may exceed, partially meet or even totally dissatisfy the demands of the query. Therefore we divide such situations into four categories and give explanations and examples (Table 3) for each category:

- If code can answer the query and even exceed the demand of the query, it is a correct answer. (Case (2) in Table 3.)
- If code can meet a certain category of the query demands, it is also a correct answer. (Case (3) and Case (4) in Table 3.)
- If code satisfies no more than 50% of the query demands, the code can not correctly answer the query. (Case (5) and Case (6) in Table 3.)
- If a small part of the code is relevant to the query, the code can not be a correct answer. (Case (7) in Table 3.)

Annotation We ask more than 100 participants, who all have a good grasp of programming knowledge, to judge the instances according to the annotation guideline. Participants are provided with the full guidelines and allowed to discuss and search on the internet during annotation. When annotation is finished, each query-code pair has been annotated by at least three participants. We remove the pairs whose inter-annotator agreement (IAA) is poor, where Krippendorff's alpha coefficient (Krippendorff, 1980) is used to measure IAA. We also remove pairs with no-code-search-intent queries. Finally, 20,604 labels for pairs of web query and code are retained, and their average Krippendorff's alpha coefficient is 0.63. Table 4 shows the statistics of CoSQA.

	# of query	avg. length	# of tokens
query	20,604	6.60	6,784
code	6,267	71.51	28,254

Table 4: Statistics of our CoSQA dataset.

4 Tasks

Based on our CoSQA dataset, we explore two tasks to study the problem of query-code matching: code search and code question answering.

The first task is natural language code search, where we formulate it as a text retrieval problem. Given a query q_i and a collection of codes $C = \{c_1, \dots, c_H\}$ as the input, the task is to find the most possible code answer c^*. The task is evaluated by Mean Reciprocal Rank (MRR).

The second task is code question answering, where we formulate it as a binary classification problem. Given a natural language query q and a code sequence c as the input, the task of code question answering predicts a label of "1" or "0" to indicate whether code c answers query q or not. The task is evaluated by accuracy score.

5 Methodology

In this section, we first describe the model for query-code matching and then present our code contrastive learning method (CoCLR) to augment more training instances.

5.1 Siamese Network with CodeBERT

The base model we use in this work is a siamese network, which is a kind of neural network with two or more identical subnetworks that have the same architecture and share the same parameters and weights (Bromley et al., 1994). By deriving fixed-sized embeddings and computing similarities, siamese network systems have proven effective in modeling the relationship between two text sequences (Conneau et al., 2017; Yang et al., 2018; Reimers and Gurevych, 2019).

We use a pretrained CodeBERT (Feng et al., 2020) as the encoder to map any text sequence to a d-dimensional real-valued vectors. CodeBERT is a bimodal model for natural language and programming language which enables high-quality text and

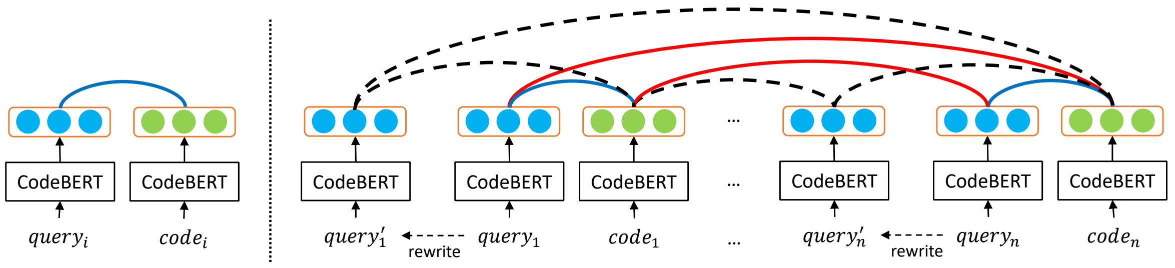

Figure 2: The frameworks of the siamese network with CodeBERT (left) and our CoCLR method (right). The blue line denotes the original training example. The red lines and dashed lines denote the augmented examples with in-batch augmentation and query-rewritten augmentation, respectively.

code embeddings to be derived. Specifically, it shares exactly the same architecture as RoBERTa (Liu et al., 2019b), which is a bidirectional Transformer with 12 layers, 768 dimensional hidden states, and 12 attention heads, and is repretrained by masked language modeling and replaced token detection objectives on CodeSearchNet corpus (Husain et al., 2019).

For each query q_i and code c_i, we concatenate a $[CLS]$ token in front of the sequence and a $[SEP]$ token at the end. Then we feed the query and code sequences into the CodeBERT encoder to obtain contextualized embeddings, respectively. Here we use the pooled output of $[CLS]$ token as the representations:

$$\mathbf{q}_i = \mathbf{CodeBERT}(q_i), \quad \mathbf{c}_i = \mathbf{CodeBERT}(c_i). \tag{1}$$

Next we perform query-code matching through a multi-layer perceptron. Following Chen et al. (2017) and Mou et al. (2016), we concatenate the query embedding $\mathbf{q}_i$ and code embedding $\mathbf{c}_i$ with the element-wise difference $\mathbf{q}_i - \mathbf{c}_i$ and element-wise product $\mathbf{q}_i \bigodot \mathbf{c}_i$, followed by a 1-layer feed-forward neural network, to obtain a relation embedding:

$$\mathbf{r}^{(i,i)} = \tanh(\mathbf{W}_1 \cdot [\mathbf{q}_i, \mathbf{c}_i, \mathbf{q}_i - \mathbf{c}_i, \mathbf{q}_i \bigodot \mathbf{c}_i]). \tag{2}$$

We expect such an operation can help sharpen the cross information between query and code to capture better matching relationships such as contradiction.

Then we put the relation embedding $\mathbf{r}^{(i,i)}$ into a final 1-layer perceptron classifier with a sigmoid output layer: $s^{(i,i)} = sigmoid(\mathbf{W}_2 \cdot \mathbf{r}^{(i,i)})$. Score $s^{(i,i)}$ can be viewed as the similarity of query q_i and code c_i.

To train the base siamese network, we use a binary cross entropy loss as the objective function:

$$\mathcal{L}_b = -[y_i \cdot \log s^{(i,i)} + (1 - y_i)\log(1 - s^{(i,i)})], \tag{3}$$

where y_i is the label of (q_i, c_i).

5.2 Code Contrastive Learning

Now we incorporate code contrastive learning into the siamese network with CodeBERT. Contrastive learning aims to learn representations by enforcing similar objects to be closer while keeping dissimilar objects further apart. It is often accompanied with leveraging task-specific inductive bias to augment similar and dissimilar examples. In this work, given an example of query and code (q_i, c_i), we define our contrastive learning task on example itself, in-batch augmented examples (q_i, c_j), and augmented example with rewritten query (q'_i, c_i). Hence, the overall training objective can be formulated as:

$$\mathcal{L} = \mathcal{L}_b + \mathcal{L}_{ib} + \mathcal{L}_{qr}. \tag{4}$$

In-Batch Augmentation (IBA) A straightforward augmentation method is to use in-batch data, where a query and a randomly sampled code are considered as dissimilar and forced away by the models. Specifically, we randomly sample n examples $\{(q_1, c_1), (q_2, c_2), \ldots, (q_n, c_n)\}$ from a mini-batch. For (q_i, c_i), we pair query q_i with the other $N - 1$ codes within the mini-batch and treat the $N - 1$ pairs as dissimilar. Let $s^{(i,j)}$ denote the similarity of query q_i and code c_j, the loss function of the example with IBA is defined as:

$$\mathcal{L}_{ib} = -\frac{1}{n-1} \sum_{\substack{j=1 \\ j \neq i}}^{n} \log(1 - s^{(i,j)}), \tag{5}$$

Query-Rewritten Augmentation (QRA) The in-batch augmentation only creates dissimilar pairs from the mini-batch, which ignores to augment similar pairs for learning positive relations. To remedy this, we propose to augment positive examples by rewriting queries. Inspired by the feature that web queries are often brief and not necessarily grammatically correct, we assume that the rewritten query with minor modifications shares the same semantics as the original one. Therefore, an augmented pair with a rewritten query from a positive pair can also be treated as positive.

Specifically, given a pair of query q_i and code c_i with $y_i = 1$, we rewrite q_i into q_i' in one of the three ways: randomly *deleting* a word, randomly *switching* the position of two words, and randomly *copying* a word. As shown in Section 6.3, switching position best helps increase the performance.

For any augmented positive examples, we also apply IBA on them. Therefore the loss function for the example with QRA is:

$$\mathcal{L}_{qr} = \mathcal{L}_b' + \mathcal{L}_{ib}', \tag{6}$$

where $\mathcal{L}_b'$ and $\mathcal{L}_{ib}'$ can be obtained by Eq. 3 and Eq. 5 by only change q_i to q_i'.

6 Experiments

We experiment on two tasks, including code question answering and natural language code search. We report model comparisons and give detailed analyses from different perspectives.

6.1 Experiment Settings

We train the models on the CoSQA dataset and evaluate them on two tasks: code question answering and code search.

On code question answering, we randomly split CoSQA into 20,000 training and 604 validation examples. As for the test set, we directly use the WebQueryTest in CodeXGLUE benchmark, which is a testing set of Python code question answering with 1,046 query-code pairs and their expert annotations.

On code search, we randomly divide the CoSQA into training, validation, and test sets in the number of 19604:500:500, and restrict the instances for validation and testing are all positive. We fix a code database with 6,267 different codes in CoSQA.

Baseline Methods CoSQA is a new dataset, and there are no previous models designed specifically for it. Hence, we simply choose RoBERTa-base (Liu et al., 2019b) and CodeBERT (Feng et al., 2020) as the baseline methods. The baseline methods are trained on CodeSearchNet Python corpus with balanced positive examples. Negative samples consist of a balanced number of instances with randomly replaced code.

Evaluation Metric We use accuracy as the evaluation metric on code question answering and Mean Reciprocal Rank (MRR) on code search.

Implementation Details We initialize CoCLR with *microsoft/codebert-base*[4] repretrained on CodeSearchNet Python Corpus (Husain et al., 2019). We use the AdamW optimizer (Loshchilov and Hutter, 2019) and set the batch size to 32 on the two tasks. On code question answering, we set the learning rate to 1e-5, warm-up rate to 0.1. On code search, we set the learning rate to 1e-6. All hyper-parameters are tuned to the best on the validation set. All experiments are performed on an NVIDIA Tesla V100 GPU with 16GB memory.

6.2 Model Comparisons

Table 5 shows the experimental results on the tasks of code question answering and code search. We can observe that:

(1) By leveraging the CoSQA dataset, siamese network with CodeBERT achieves overall performance enhancement on two tasks, especially for CodeXGLUE WebQueryTest, which is an open challenge but without direct training data. The result demonstrates the high-quality of CoSQA and its potential to be the training set of WebQueryTest.

(2) By integrating the code contrastive learning method, siamese network with CodeBERT further achieves significant performance gain on both tasks. Especially on the task of WebQueryTest, CoCLR achieves the new state-of-the-art result by increasing 15.6%, which shows the effectiveness of our proposed approach.

6.3 Analysis: Effects of CoCLR

To investigate the effects of CoCLR in query-code matching, we perform ablation study to analyze the major components in our contrastive loss that are of importance to help achieve good performance. We conduct experiments on the CoSQA code search task, using the following settings: (i) fine-tuning with vanilla binary cross-entropy loss only, (ii) fine-tuning with additional in-batch augmentation (IBA) loss, (iii) fine-tuning with additional query-rewritten augmentation (QRA) loss, (vi) fine-tuning with both additional IBA and QRA loss. And for QRA loss, we also test the three rewriting methods when applied individually. The results are listed in Table 6. We can find that:

[4] https://github.com/microsoft/CodeBERT

Model	Data	Code Question Answering	Code Search
RoBERTa[2]	CSN	40.34	0.18
CodeBERT[2]	CSN	47.80	51.29
CodeBERT	CSN + CoSQA	52.87	54.41
CodeBERT + CoCLR	CSN + CoSQA	**63.38**	**64.66**

Table 5: Evaluation on code question answering and code search. CSN denotes CodeSearchNet Python corpus. By incorporating CoCLR method, siamese network with CodeBERT outperforms the existing baseline approaches.

Augmentations	MRR
no augmentations	54.41
+ query-rewritten (delete)	55.24
+ query-rewritten (copy)	54.82
+ query-rewritten (switch)	55.66
+ in-batch	63.51
+ in-batch + query-rewritten (delete)	63.41
+ in-batch + query-rewritten (copy)	63.97
+ in-batch + query-rewritten (switch)	**64.66**

Table 6: Performance of CodeBERT with different augmentations in CoCLR on code search.

(1) Both incorporating IBA and QRA individually or together improve models' performance. This indicates the advantage of applying code contrastive learning for code search.

(2) No matter integrating IBA or not, the model with QRA by *switching* method performs better than models with the other two methods. We attribute the phenomenon to the fact that web queries do not necessarily have accurate grammar. So switching the positions of two words in the query better maximizes the agreement between the positive example and the pseudo positive example than the other two augmentations, which augments better examples to learn representations.

(3) Comparing the two augmentations, adding IBA achieves more performance gain than QRA (1.25% versus 9.10%). As the numbers of examples with QRA and examples with IBA are not equal under two settings, we further evaluate the model with only one more example with IBA. The MRR is 55.52%, which is comparable to the performance of adding one more example with QRA. This suggests that there may be no difference between adding examples with IBA or examples with QRA. Instead, the number of high-quality examples is important for training. Similar findings are also reported in Sun et al. (2020), and a theoretical analysis is provided in Arora et al. (2019).

Code Component	MRR
complete code	**64.66**
w/o header	62.01
w/o body	59.11
w/o documentation	58.54
w/o header & body	52.89
w/o header & documentation	43.35
w/o body & documentation	42.71

Table 7: Performance of CoCLR-incorporated CodeBERT trained and tested with different code components on code search.

6.4 Analysis: Effects of Code Components

To explore the effects of different components of code in query-code matching, we evaluate CoCLR on code search and process the codebase by the following operations: (i) removing the function header, (ii) removing the natural language documentation, (iii) removing the code statements in the function body. We also combine two of the above operations to see the performance. From the results exhibited in Table 7, we can find that: by removing code component, the result of removing documentation drops more than those of removing header and removing function body. This demonstrates the importance of natural language documentation in code search. Since documentation shares the same modality with the query and briefly describes the functionality of the code, it may be more semantically related to the query. Besides, it also reveals the importance of using web queries rather than treating documentation as queries in code search datasets, which liberates models from the matching between documentation with code to the matching between query with documentation and code.

7 Conclusion

In this paper, we focus on the matching problem of the web query and code. We develop a large-scale human-annotated query-code matching dataset CoSQA, which contains 20,604 pairs of real-world web queries and Python functions with documentation. We demonstrate that CoSQA is an ideal dataset for code question answering and code search. We also propose a novel code contrastive learning method, named CoCLR, to incorporate artificially generated instances into training. We find that model with CoCLR outperforms the baseline models on code search and code question answering tasks. We perform detailed analysis to investigate the effects of CoCLR components and code components in query-code matching. We believe our annotated CoSQA dataset will be useful for other tasks that involve aligned text and code, such as code summarization and code synthesis.

Acknowledgement

We thank all anonymous reviewers for their useful comments. We also thank Zenan Xu, Daya Guo, Shuai Lu, Wanjun Zhong and Siyuan Wang for valuable discussions and feedback during the paper writing process.

References

Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Survey*.

S. Arora, Hrishikesh Khandeparkar, M. Khodak, Orestis Plevrakis, and Nikunj Saunshi. 2019. A theoretical analysis of contrastive unsupervised representation learning. In *ICML*.

S. Bajracharya, Trung Chi Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, P. Baldi, and C. Lopes. 2006. Sourcerer: a search engine for open source code supporting structure-based search. In *OOPSLA '06*.

Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. 1994. Signature verification using a "siamese" time delay neural network. In *Advances in Neural Information Processing Systems*.

José Cambronero, Hongyu Li, S. Kim, K. Sen, and S. Chandra. 2019. When deep learning met code search. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

Qian Chen, Xiao-Dan Zhu, Zhenhua Ling, Si Wei, Hui Jiang, and D. Inkpen. 2017. Enhanced lstm for natural language inference. In *ACL*.

Alexis Conneau, Douwe Kiela, Holger Schwenk, Loïc Barrault, and Antoine Bordes. 2017. Supervised learning of universal sentence representations from natural language inference data. In *EMNLP*.

Zhangyin Feng, Daya Guo, Duyu Tang, N. Duan, X. Feng, Ming Gong, Linjun Shou, B. Qin, Ting Liu, Daxin Jiang, and M. Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of ICSE*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, L. Zhou, N. Duan, Jian Yin, Daxin Jiang, and M. Zhou. 2020. Graphcodebert: Pre-training code representations with data flow. In *ICLR*.

Rajarshi Haldar, Lingfei Wu, JinJun Xiong, and Julia Hockenmaier. 2020. A multi-perspective architecture for semantic code search. In *Proceedings of ACL*.

Geert Heyman and Tom Van Cutsem. 2020. Neural code search revisited: Enhancing code snippet retrieval through natural language intent. *ArXiv*, abs/2008.12193.

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*.

K. Krippendorff. 1980. Krippendorff, klaus, content analysis: An introduction to its methodology . beverly hills, ca: Sage, 1980.

Hongyu Li, S. Kim, and S. Chandra. 2019. Neural code search evaluation dataset. *ArXiv*, abs/1908.09804.

C. Liu, Xin Xia, David Lo, Cuiyun Gao, Xiaohu Yang, and J. Grundy. 2020a. Opportunities and challenges in code search tools. *ArXiv*, abs/2011.02297.

Chao Liu, Xin Xia, David Lo, Zhiwei Liu, A. Hassan, and Shanping Li. 2020b. Simplifying deep-learning-based model for code search. *ArXiv*, abs/2005.14373.

Jason Liu, Seohyun Kim, Vijayaraghavan Murali, Swarat Chaudhuri, and Satish Chandra. 2019a. Neural query expansion for code search. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. ACM.

Y. Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, M. Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019b. Roberta: A robustly optimized bert pretraining approach. *ArXiv*, abs/1907.11692.

I. Loshchilov and F. Hutter. 2019. Decoupled weight decay regularization. In *ICLR*.

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

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. *CoRR*, abs/2102.04664.

Fei Lv, H. Zhang, Jian-Guang Lou, S. Wang, D. Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on api understanding and extended boolean model (e). *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 260–270.

Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. In *Proceedings of IJCNLP*.

Lili Mou, Rui Men, Ge Li, Yan Xu, Lu Zhang, Rui Yan, and Zhi Jin. 2016. Natural language inference by tree-based convolution and heuristic matching. In *ACL*.

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*, 9:771–783.

M. M. Rahman. 2019. Supporting code search with context-aware, analytics-driven, effective query reformulation. *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 226–229.

M. M. Rahman, C. Roy, and D. Lo. 2019. Automatic query reformulation for code search using crowdsourced knowledge. *Empirical Software Engineering*, pages 1–56.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *EMNLP/IJCNLP*.

Saksham Sachdev, H. Li, Sifei Luan, S. Kim, K. Sen, and S. Chandra. 2018. Retrieval on source code: a neural code search. *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*.

S. Sun, Zhe Gan, Y. Cheng, Yuwei Fang, Shuohang Wang, and Jing jing Liu. 2020. Contrastive distillation on intermediate representations for language model compression. In *EMNLP*.

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. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 13–25.

Shuhan Yan, Hang Yu, Yuting Chen, Beijun Shen, and Lingxiao Jiang. 2020. Are the code snippets what we are searching for? a benchmark and an empirical study on code search with natural-language queries. In *Proceedings of SANER*.

Yinfei Yang, Steve Yuan, Daniel Cer, Sheng-yi Kong, Noah Constant, Petr Pilar, Heming Ge, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. 2018. Learning semantic textual similarity from conversations. In *Proceedings of The Third Workshop on Representation Learning for NLP*.

Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. 2019. Coacor: Code annotation for code retrieval with reinforcement learning. In *Proceedings of WWW*.

Ziyu Yao, Daniel S Weld, Wei-Peng Chen, and Huan Sun. 2018. StaQC: A systematically mined question-code dataset from stack overflow. In *Proceedings of WWW*.

Xin Ye, Hui Shen, Xiao Ma, Razvan C. Bunescu, and Chang Liu. 2016. From word embeddings to document similarities for improved information retrieval in software engineering. *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*.

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 *International Conference on Mining Software Repositories*.

Jie Zhao and Huan Sun. 2020. Adversarial training for code retrieval with question-description relevance regularization. In *Findings of the Association for Computational Linguistics: EMNLP 2020*.

A Heuristics for Query Filtering

In this section, we introduce our heuristic rules to filter potential queries without code search intent. Basically, the rules are created from keyword templates and we follow the six categories of queries without code search intent to derive the keywords. Note that vague queries are morphologically variable so we ignore this categories. The keywords are shown in Table 8.

Categories	Keywords
Debugging	exception, index out of, ignore, omit, stderr, try . . . except, debug, no such file or directory, warning,
Conceptual	vs, versus, difference, advantage, benefit, drawback, interpret, understand, cannot, can’t, couldn’t, could not, how many, how much, too much, too many, more, less, what if, what happens, what is, what are, when, where, which, why, reason, how do . . . work, how . . . works, how does . . . work, need, require, wait, turn . . . on/off, turning . . . on/off,
Programming Knowledge	tutorial, advice, course, proposal, discuss, suggestion, parameter, argument, statement, class, import, inherit, operator, override, decorator, descriptor, declare, declaration
Tools Usage	console, terminal, open python, studio, ide, ipython, jupyter, notepad, notebook, vim, pycharm, vscode, eclipse, sublime, emacs, utm, komodo, pyscripter, eric, c#, access control, pip, install, library, module, launch, version, ip address, ipv, get . . . ip, check . . . ip, valid . . . ip,
Others	unicode, python command, “()”, “.”, “_”, “:”, “@”, “=”, “>”, “<”, “-”

Table 8: Keywords of queries without code search intent in five categories.