Contrastive Learning with Keyword-based Data Augmentation for Code Search and Code Question Answering

Shinwoo Park[1], **Youngwook Kim**[2] and **Yo-Sub Han**[2]
[1]Department of Artificial Intelligence, Yonsei University, Seoul, Republic of Korea
[2]Department of Computer Science, Yonsei University, Seoul, Republic of Korea
{pshkhh, youngwook, emmous}@yonsei.ac.kr

Abstract

The semantic code search is to find code snippets from the collection of candidate code snippets with respect to a user query that describes functionality. Recent work on code search proposes data augmentation of queries for contrastive learning. This data augmentation approach modifies random words in queries. When a user web query for searching code snippet is too brief, the important word that represents the search intent of the query could be undesirably modified. A code snippet has informative components such as function name and documentation that describe its functionality. We propose to utilize these code components to identify important words and preserve them in the data augmentation step. We present KeyDAC (Keyword-based Data Augmentation for Contrastive learning) that identifies important words for code search from queries and code components based on term matching. KeyDAC augments query-code pairs while preserving keywords, and then leverages generated training instances for contrastive learning. We use KeyDAC to fine-tune various pre-trained language models and evaluate the performance of code search and code question answering via CoSQA and WebQueryTest. The experimental results confirm that KeyDAC substantially outperforms the current state-of-the-art performance, and achieves the new state-of-the-arts for both tasks.

1 Introduction

Software developers or students who major in computer science often write natural language queries to search for code snippets with desired functionality from the web search engine. The retrieved code snippets are reused or referred to improve productivity of software development. Semantic code search is a well-known code-related downstream task that measures the semantic relevance between a given natural language query and a collection of code snippets to retrieve the most relevant code snippet. CodeXGLUE (Lu et al., 2021), a benchmark for 10 code tasks, provides two test sets—CodeSearchNet AdvTest and WebQueryTest—for an open challenge code search. CodeSearchNet AdvTest from the CodeSearchNet (Husain et al., 2019) corpus is for a retrieval scenario in which one needs to find a Python code function that matches the search intent of the query best. The queries in CodeSearchNet AdvTest are not real user queries. Instead, they are documentations such as comments or summaries written by developers. WebQueryTest is a code question answering task that asks whether a Python code function has functionality described by the real user web query collected from the search logs of a commercial search engine. Recently, Huang et al. (2021) introduce CoSQA that consists of pairs of real user web queries and Python code snippets. Then, they use CoSQA as a benchmark for code search to resemble the real world scenario where developers write natural language queries to find code snippets.

Huang et al. (2021) propose CoCLR, a contrastive learning method using a query-rewritten data augmentation, to improve the code search performance. CoCLR modifies the random words in queries to augment training query-code pair instances. However, besides the user queries, there are useful components that describe the functionality of code snippets. Since a function name and documentation describe functionality of the corresponding code function, we propose to use these components as well for data augmentation. Furthermore, we suggest considering the relative importance of words in query and code function. Note that the data augmentation used in the previous code search method treats all words equally. Xie et al. (2020) use TF-IDF to calculate the relative importance of words in a sentence, then replace unimportant words to augment training sentences. We adapt this idea and define the relative importance in code search based on term matching between a

Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics, pages 3609–3619
May 2-6, 2023

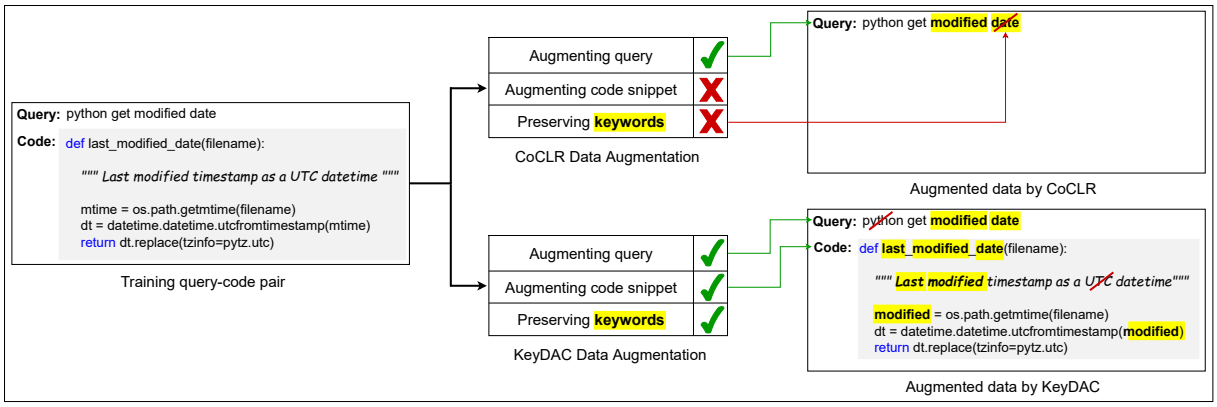

Figure 1: Comparison of data augmentation approaches for semantic code search. Example query-code pair is from CoSQA (Huang et al., 2021) training dataset. Keywords are marked in yellow and red slash line denotes the deletion of a word. Both CoCLR (Huang et al., 2021) and KeyDAC apply data augmentation to query. In contrast to CoCLR, KeyDAC also augments code snippet (deleting `UTC` in documentation and renaming variable `mtime` to *modified*, which is one of keywords). While CoCLR can delete the keyword *date*, KeyDAC preserves keywords.

paired query and code snippet.

We present KeyDAC, **Key**word-based **D**ata **A**ugmentation for **C**ontrastive learning that identifies keywords from a given query-code pair based on term matching. KeyDAC applies data augmentation both on natural language (NL) sequences (*i.e.*, query, function name, and documentation) and programming language (PL) sequences (*i.e.*, code statements), and generates more training query-code pair instances while preserving keywords. Figure 1 shows an example training query-code pair instance from CoSQA training dataset and compares the results of two data augmentation approaches, CoCLR and KeyDAC. The query asks how to get modified date. In this case, important words closely related to the search intent are *modified* and *date*. Since CoCLR modifies random words in query, *date* could be modified in data augmentation process. The augmented query by CoCLR is *python get modified* (keyword *date* is deleted) which does not represent clear search intent. On the other hand, KeyDAC preserves keywords in the data augmentation step. In addition, KeyDAC augments code snippet by deleting unimportant word `UTC` in documentation, and renaming the variable in the code statements using keywords (`mtime` to *modified*). We evaluate the effectiveness of KeyDAC on CoSQA and open challenge WebQueryTest.[1] We use KeyDAC to fine-tune various pre-trained language models. KeyDAC brings substantial performance gain, resulting in new state-of-the-art performance both on code search and code question answering tasks. Our main contributions are as follows:

- We propose KeyDAC—data augmentation mechanism for contrastive learning, which identifies important words from training query-code pairs and augments them while preserving identified keywords.
- We demonstrate that KeyDAC outperforms the current SOTA for the code search task on CoSQA benchmark.
- We achieve a new record, with a substantial improvement, for the open challenge code QA, WebQueryTest.

2 Related Work

2.1 Code Search Methods

Some researchers proposed information retrieval-based approaches using term matching between queries and code snippets (Lu et al., 2015; Lv et al., 2015). However, since these traditional approaches focused on lexical information, they often fail to understand the semantic relationship between the query and the code snippet. Recently, many researchers started to address the problem through deep learning-based approaches (Cambronero et al., 2019; Li et al., 2020) that learn the semantic representation of the query and code. Later, pre-trained models for programming languages were proposed and showed an improvement on code search tasks. CodeBERT (Feng et al., 2020) is a bimodal pre-trained model for NL and PL, which was pre-trained with masked language modeling

[1]The leaderboard of WebQueryTest is available at https://microsoft.github.io/CodeXGLUE/

(MLM) and replaced token detection (RTD). Feng et al. (2020) showed that CodeBERT outperformed both RoBERTa and RoBERTa pre-trained on code datasets. GraphCodeBERT (Guo et al., 2020) further improved the performance by pre-training using data flow as a semantic-level structure of code.

2.2 Contrastive Learning for Code Search

Contrastive learning encourages the distance between similar instances to be minimized and the distance between dissimilar instances to be maximized in the representation space. Recently, contrastive learning showed its effectiveness and became popular in self-supervised learning (Chen et al., 2020; Meng et al., 2021; Gao et al., 2021). Such effectiveness of contrastive learning in various fields promoted adapting contrastive learning for code search. As a pre-training approach for source code, Corder (Bui et al., 2021) transformed a code snippet into different versions and minimized the distance between them in the representation space. Corder used program transformation operators, including dead code insertion and permutation of code statements. The experiment results confirmed that pre-training with contrastive learning was effective in improving performance on several code-related tasks such as code-to-code search, text-to-code search, and code summarization. As a fine-tuning approach for code search, CoCLR (Huang et al., 2021) used contrastive learning with query-rewritten data augmentation and in-batch negative samples. For query-rewriting, CoCLR modified random words in a query as follows: deleting random words, switching the position of two random words, or copying random words. They showed that CoCLR improved code search performance of CodeBERT. However, these contrastive learning approaches utilize either query or code in the data augmentation process and do not consider the relative importance of words. We propose to consider the importance of each word differently and preserve keywords for data augmentation.

3 Approach

KeyDAC identifies keywords from positive query-code pairs (*i.e.*, a code snippet meets the demand of a query) in a training dataset based on term matching. Then KeyDAC applies keyword-based data augmentation to both query and code snippet. Using augmented training data, KeyDAC deploys contrastive learning to fine-tune pre-trained encoders for the code search task.

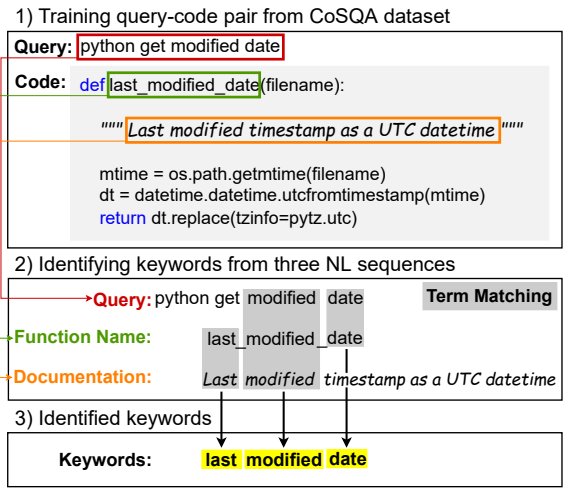

Figure 2: An example of identifying keywords from a pair of real user query and code function. Given a training query-code pair, we identify keywords based on term matching from three NL sequences (*i.e.*, query, function name, and documentation). The identified keywords are related to the functionality of the code function.

3.1 Data Augmentation with Keywords

A code function c_i has the following three main components:

- function name in the function header (NL)
- function-level documentation (NL)
- code statements in the function body (PL)

The previous approaches consider these three components as PL sequences. On the other hand, we consider the function name and the documentation as NL sequences, since 1) those two code components describe the functionality of the code snippet; 2) modifying the function name or documentation does not produce any syntax errors.

Identifying Keywords Since a user query demands certain functionality of the code snippet, KeyDAC utilizes two NL descriptions of code function, such as the function name and documentation, to identify keywords. Specifically, KeyDAC identifies common words from three NL sequences (*i.e.*, query, function name, and documentation) based on term matching. Figure 2 gives an example of how KeyDAC identifies keywords from the paired query and code function. The identified keywords from the example are: *last*, *modified* and *date*. These

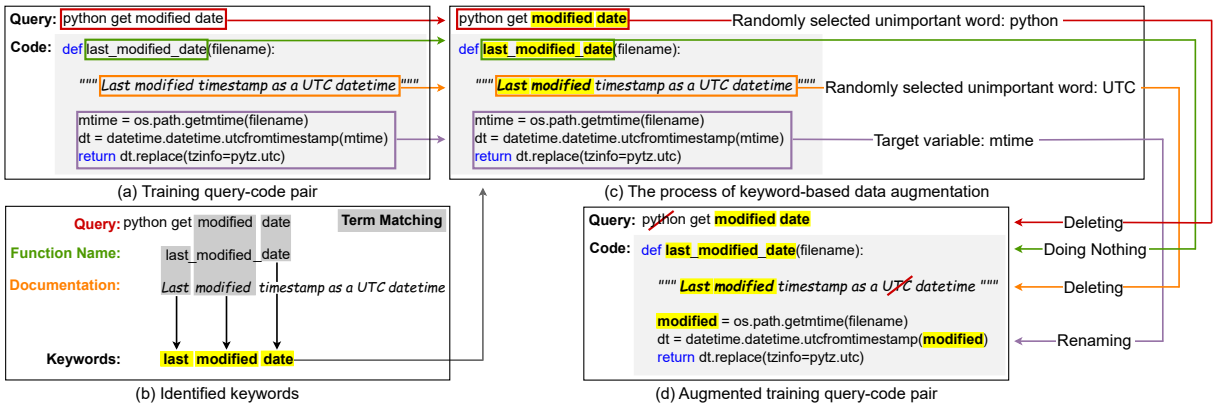

Figure 3: An illustration of keyword-based data augmentation. For NL sequences, keywords (*i.e.*, *last*, *modified*, and *date*) are preserved (highlighted in bold) while unimportant words (*i.e.*, `python` in query and `UTC` in documentation) are deleted (red slash lines) (**NL: Rewriting**). For PL sequences, variable `mtime` is renamed using a keyword *modified* (**PL: Variable Renaming**).

identified keywords are related to the functionality of code function.

Using the identified keywords, KeyDAC applies data augmentation to NL and PL sequences in different ways. In the following, we demonstrate keyword-based data augmentation in detail.

NL: Rewriting KeyDAC rewrites three NL sequences by modifying unimportant words while preserving keywords by choosing one of the following four ways: 1) deleting one randomly selected unimportant word (Delete); 2) switching the position of two randomly selected unimportant words (Switch); 3) copying one randomly selected unimportant word (Copy); 4) doing nothing (None). In Figure 3 (d), KeyDAC deletes an unimportant word `python` in the query and `UTC` in the documentation, while preserving keywords *last*, *modified* and *date*. **PL: Variable Renaming** Software developers sometimes name a variable in abbreviation form; in other words, there can be a lexical gap between query and variable name (*e.g.*, in Figure 3 (a), the variable `mtime` represents the meaning of *modified time*). We propose to rename variables using keywords to bridge the lexical gap between query and code snippet. We first parse a code function c_i into an abstract syntax tree (AST) that represents the syntactic structure of c_i. Then we identify variables from terminal nodes of the resulting AST. Then KeyDAC renames the variables that appear most frequently in code statements using keywords. If there is a keyword that matches the target variable name (`mtime`) with the first letter, KeyDAC uses that keyword (*modified*). If not, KeyDAC randomly chooses one from keywords (*i.e.*, *last*, *modified*, and *date*). In Figure 3 (d), KeyDAC renames `mtime` to *modified*.

3.2 Siamese Network for Code Search Task

We adopt siamese network architecture, which has a shared encoder to map a query and code snippet to fixed-sized embeddings. Each query q_i and code function c_i are encoded by a shared encoder $Encoder$ (*e.g.*, CodeBERT). We take the representation of $[CLS]$ token from the last hidden layer of $Encoder$. Then we compute the cosine similarity $sim^{(q_i,c_i)}$ between a query-code pair (q_i, c_i) as:

$$sim^{(q_i,c_i)} = \langle Encoder(q_i), Encoder(c_i) \rangle, \quad (1)$$

where $\langle \cdot \rangle$ indicates cosine similarity operation. We use binary cross-entropy as the training objective:

$$\mathcal{L}_{bce} = -[y_i \cdot \log sim^{(q_i,c_i)} + (1 - y_i) \log(1 - sim^{(q_i,c_i)})], \quad (2)$$

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

3.3 Contrastive Learning

KeyDAC uses contrastive learning to optimize the parameters of $Encoder$. This contrastive learning aims to maximize the similarity of the query q_i and code function c_i with label of $y_i = 1$ while minimizing the similarity of the query with unrelated code snippets.

Given the query-code pair (q_i, c_i) in a batch of size N, we consider the other $N - 1$ code snippets as unrelated. The contrastive loss with in-batch negative samples is defined as:

$$\mathcal{L}_{ib} = -\frac{1}{N-1} \sum_{\substack{j=1 \\ j \neq i}}^{N} \log \left(1 - sim^{(q_i,c_j)}\right). \quad (3)$$

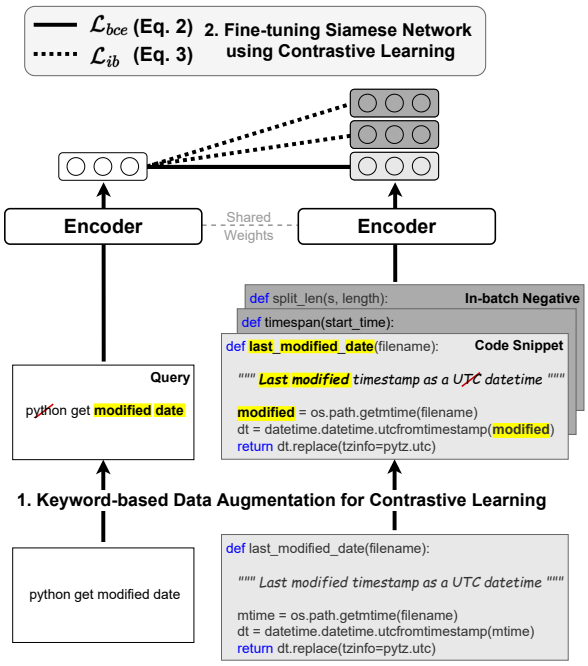

Figure 4: The fine-tuning process of siamese network using KeyDAC. The boxes, colored in dark gray, indicate the in-batch negative samples. The black solid line denotes the binary cross-entropy loss and black dashed lines denote the contrastive loss with in-batch negative.

The overall training objective is:

$$\mathcal{L} = \mathcal{L}_{bce} + \mathcal{L}_{ib}. \quad (4)$$

Figure 4 illustrates the fine-tuning process of siamese network with shared encoder using KeyDAC. A black solid line denotes the binary cross-entropy loss (Equation 2), and black dashed lines denote the contrastive loss with in-batch negative samples (Equation 3).

4 Experiments

4.1 Tasks

We evaluate the performance for two tasks, code search and code question answering.

Code Search aims to retrieve the most relevant code function c^* in a collection of H code snippets $C = c_1, \ldots, c_H$ according to a real user web query q_i. Following CoSQA, we use Mean Reciprocal Rank (MRR) as evaluation metric.

Code Question Answering is a form of binary classification and, thus, predicts a label of 1 or 0; the label indicates whether the code function c_i matches the search intent of the query q_i. The WebQueryTest uses accuracy score as its official evaluation metric.

4.2 Datasets

We use CoSQA (Huang et al., 2021) and WebQueryTest (Lu et al., 2021) to evaluate the effectiveness of KeyDAC. The queries in both datasets are real user queries. Table 1 shows the data statistics. Each number denotes the number of paired queries and code snippets.

Code Search: We follow the same dataset split as CoSQA. The number of candidate code snippets is 6,267 ($H = 6,267$).

Code Question Answering: Each instance of WebQueryTest is a pair of a real user web query and a Python code function. Following CoSQA, we train the models using the CoSQA dataset, then use WebQueryTest as the test dataset. Since WebQueryTest is an open challenge, we submit model predictions to the CodeXGLUE official leaderboard, and report the evaluated results.

4.3 Baseline Approaches

- **In-batch**: Contrastive learning method using in-batch negative (without data augmentation).
- **CoCLR**: Contrastive learning method with query-rewritten data augmentation and in-batch negative.

Huang et al. (2021) report that switching the position of two random words in queries for the data augmentation in CoCLR achieves the best performance. Thus, we use switch as the query-rewriting operation for CoCLR in the experiments (Table 2).

4.4 Experiment Setup

We set the batch size N as 32, the learning rate as 1e-5 and the fine-tuning epoch as 10. We use the Adam optimizer (Kingma and Ba, 2014) to train the models. We conduct all experiments on an NVIDIA RTX3090 GPU with 24GB memory.

We use the following pre-trained models[2]:

- **RoBERTa** (Liu et al., 2019) is pre-trained on a large natural language text corpus with masked language modeling (MLM) objective.
- **CodeBERT** (Feng et al., 2020) is pre-trained on six programming languages with MLM and replaced token detection objectives.

[2]We use HuggingFace's implementation for these models. (https://huggingface.co/)

Dataset	Task	Metric	Train	Valid	Test
CoSQA (Huang et al., 2021)	Code Search	MRR	19,604	500	500
	Code Question Answering	Accuracy	20,000	604	1,046*

Table 1: Dataset statistics. * This is the number of test data in the WebQueryTest open challenge.

Model	Approach	Code Search	Code Question Answering[3]
RoBERTa (Liu et al., 2019)	In-batch	58.37 ± 0.48	58.89
	CoCLR	61.00 ± 0.98	60.70
	KeyDAC	**67.09 ± 0.37**	**60.99**
CodeBERT (Feng et al., 2020)	In-batch	66.72 ± 0.35	57.36
	CoCLR	68.42 ± 0.44	60.03
	KeyDAC	**72.76 ± 0.93**	**62.90**
GraphCodeBERT (Guo et al., 2020)	In-batch	71.34 ± 0.46	63.73±1.28
	CoCLR	71.78 ± 0.69	63.47±1.31
	KeyDAC	**74.93 ± 0.42**	**65.51**±0.77
UniXcoder (Guo et al., 2022)	In-batch	71.87 ± 0.59	62.58±1.06
	CoCLR	71.52 ± 0.85	62.10±0.52
	KeyDAC	**74.71 ± 0.45**	**64.14**±0.78

Table 2: Results on code search and code question answering tasks. The best results for each model are highlighted in bold.

- **GraphCodeBERT** (Guo et al., 2020) leverages data flow of code for two structure-aware pre-training tasks, including edge prediction and node alignment.
- **UniXcoder** (Guo et al., 2022) is a unified pre-trained model which uses code documentation and AST for contrastive pre-training. We use encoder of UniXcoder.

We deploy contrastive learning approaches to fine-tune the aforementioned pre-trained language models.

4.5 Results

Table 2 compares different contrastive learning approaches to code search and code question answering. We report the mean performance with standard deviation in 5 runs for code search task.[3] Remark that the in-batch negative contrastive learning approach (In-batch) does not use data augmentation. We highlight the best results for each pre-trained model in bold.

The results show that KeyDAC consistently outperforms contrastive learning with in-batch negative only (In-batch) and contrastive learning with query-rewritten data augmentation (CoCLR). GraphCodeBERT fine-tuned using KeyDAC achieves the highest performance both on code search and code QA tasks. We notice that CoCLR drops code search performance of UniXcoder and code QA performance of GraphCodeBERT and UniXcoder. On the other hand, KeyDAC shows consistent performance improvement.

5 Analysis

We use GraphCodeBERT as base model for following analyses since among four pre-trained models, GraphCodeBERT achieves the highest performance on code search and code QA when fine-tuned using KeyDAC.

5.1 The Effect of Preserving Keywords

From the experimental results, we observe that KeyDAC consistently outperforms CoCLR. We hypothesize that the main reason for the performance gain is preserving keywords in data augmentation. We study the effect of preserving keywords, especially in queries to directly compare KeyDAC and CoCLR. We perform query-rewritten data augmentation in three ways: 1) Preserving keywords (same as applying keyword-based data augmentation to query only); 2) Deleting a random word (same as

[3]For code QA task, we submit the prediction results of 3 random seeds for the two best models and the prediction results of 1 random seed for the others, since the CodeXGLUE guideline discourages excessive submissions to avoid P-hacking.

Component	Rewriting	Code Search
Query	Preserving keywords	**72.03**
	Deleting a random word	71.07
	Deleting a keyword	68.03

Table 3: Effect of preserving keywords in query-rewritten data augmentation.

Model	Data Augmentation	Code Search
GraphCodeBERT	No augmentation	71.46
	(Delete)	**74.81**
	(Switch)	73.74
	(Copy)	72.37

Table 4: Effect of NL rewriting operations. The result in the first row indicates the performance of fine-tuning GraphCodeBERT using in-batch negative only.

CoCLR with delete operation); 3) Deleting a keyword. We analyze this via code search task on CoSQA test set to avoid an excessive submission of code QA prediction results to the WebQueryTest leaderboard. Table 3 shows the results. We can observe that preserving keywords in query-rewritten data augmentation achieves the best performance. The deletion of keywords in queries shows significant performance degradation. The results suggest that keywords determined through our proposed way (Figure 2) are important to the performance of the semantic code search task.

5.2 The Effect of NL Rewriting Operation

We conduct experiments to investigate the effect of different NL rewriting operations on the code search task. Table 4 shows the code search performance evaluated on the CoSQA test set. We observe that KeyDAC with all three NL rewriting operations outperforms GraphCodeBERT fine-tuned using contrastive learning with in-batch negative only (No augmentation).

Among the three rewriting operations, delete shows the best performance (highlighted in bold). We conjecture that deleting unimportant words helps to reduce the noise of data. For example, user web queries often have a typo, and documentation often contains a special character such as >>> to show the execution result.

5.3 Contribution of Each Component

We investigate the contributions of each component to keyword-based data augmentation. Table 5 shows the results for code search task on the CoSQA test set. All five rows determine keywords from pairs of query and code function with the same way. However, the last four rows only modify each component as follows: 1) deleting unimportant words in queries; 2) deleting unimportant words in function names; 3) deleting unimportant words in documentations; 4) renaming variables using keywords.

The results show that KeyDAC leveraging all components (results in the first row) achieves the best result. Among the results of KeyDAC using a single component, documentation contributes the most to performance, and the function name contributes the least. Software developers typically use an abbreviated function name but write documentation in detail to describe the functionality of the code snippet.

5.4 Case Study

We conduct a case study to analyze the consistent code search ability of KeyDAC. More cases for other pre-trained language models can be found in Appendix A.1. In addition, we provide some cases to compare the prediction results of CoCLR and KeyDAC for code question answering task in Appendix A.2

Query: split string into n parts python

Gold Code:

```
def _split_str(s, n):
    """split string into list of strings
       by specified number."""
    length = len(s)
    return [s[i:i + n] for i in range(0, length, n)]
```

(a) A pair of real user web query and gold code function.

Top-1 Code:

```
def _split_str(s, n):
    """split string into list of strings
       by specified number."""
    length = len(s)
    return [s[i:i + n] for i in range(0, length, n)]
```

Top-2 Code:

```
def split_len(s, length):
    """split string *s* into list of strings
       no longer that *length*"""
    return [s[i:i+length] for i in range(0, len(s), length)]
```

Top-3 Code:

```
def _split(string, splitters):
    """Splits a string into parts at
       multiple characters"""
    part = ''
    for character in string:
        if character in splitters:
            yield part
            part = ''
        else:
            part += character
            yield part
```

(b) Code search results with KeyDAC (top-3 results).

Figure 5: Code search results with KeyDAC on CoSQA test set. The code snippets are searched from 6,267 candidates and ranked in the order of semantic relatedness.

Model	Component	Keyword-based Data Augmentation	Code Search
GraphCodeBERT	All Components	NL Rewriting via delete & Variable Renaming	**74.81**
	Query	Deleting unimportant words in queries	72.03
	Function Name	Deleting unimportant words in function names	71.39
	Documentation	Deleting unimportant words in documentations	73.26
	Code Statements	Renaming variables using keywords	72.57

Table 5: The first row indicates the performance of KeyDAC which applied keyword-based data augmentation to all components. The last four rows are the results of applying keyword-based data augmentation to a single component.

Figure 5 shows the code search results with KeyDAC for the query *split string into n parts python*. We only give the top-3 results due to the space limit. KeyDAC returns the gold code function as its top-1 result. The other two code snippets have the same functionality as the gold code, demonstrating the consistent code search ability of KeyDAC.

5.5 Error Analysis

Figure 6 provides two error cases of KeyDAC on code QA task. KeyDAC makes a wrong prediction in the case of Figure 6(a). The purpose of the search for the query is to calculate the l1 norm between vectors. However, the functionality of the code is computing l2 norm of a given array. The model needs mathematical knowledge to understand the difference between norm l1 and norm l2.

Query: python l1 norm between vectors

Code:
```
def l2_norm(arr):
    """The l2 norm of an array is defined as:
       sqrt(||x||), where ||x|| is the dot product
       of the vector """
    arr = np.asarray(arr)
    return np.sqrt(np.dot(arr.ravel().squeeze(), arr.ravel().squeeze()))
```

Label: 0

Prediction: 0.6800 (1)

(a)

Query: python script chmod +x

Code:
```
def chmod_plus_w(path):
    """Equivalent of unix `chmod +w path` """
    path_mode = os.stat(path).st_mode
    path_mode &= int('777', 8)
    path_mode |= stat.S_IWRITE
    os.chmod(path, path_mode)
```

Label: 0

Prediction: 0.6027 (1)

(b)

Figure 6: Two error cases of KeyDAC for code question answering task on CoSQA validation set.

In the case of Figure 6(b), the search intent of the query is to add the execution privilege (*chmod +x*) while the functionality of the code is to add the write privilege (*chmod +w*). KeyDAC fails to understand the difference between *+x* and *+w*, resulting in a wrong prediction. While software developers can easily understand the difference of Unix/Linux command *chmod +x* and *chmod +w*, it is not trivial for the language models and humans without programming knowledge.

The potential research direction is to incorporate domain-specific knowledge, such as mathematical knowledge and programming knowledge, into the pre-training or fine-tuning process of language models to improve code search performance.

6 Conclusion

We have presented KeyDAC— keyword-based data augmentation for contrastive learning, which generates more training query-code pairs while preserving important keywords for the code search task. First, KeyDAC utilizes term matching technique to identify important words from a query and code components (function name and documentation). Then, KeyDAC augments both a query and a code snippet while preserving the identified keywords. Finally, KeyDAC deploys contrastive learning using the augmented data to fine-tune the pre-trained language models. We have demonstrated that KeyDAC outperforms the current state-of-the-art performance on both the code search and an open challenge code question answering task.

Limitations

Given a query-code pair, KeyDAC identifies keywords which share the same surface form by term matching. In other words, KeyDAC identifies keywords at the lexical level. As a future work, KeyDAC can utilize external knowledge for keyword-based data augmentation. For example, KeyDAC can utilize WordNet to identify keywords based on not only surface form, but also synonyms.

Acknowledgements

This research was supported by the NRF grant (RS-2023-00208094) and the AI Graduate School Program (No. 2020-0-01361) funded by the Korea government (MSIT). Han is a corresponding author.

References

Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*.

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*.

Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*.

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

Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. SimCSE: Simple contrastive learning of sentence embeddings. *arXiv preprint arXiv:2104.08821*.

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 Annual Meeting of the Association for Computational Linguistics (ACL)*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. GraphCodeBERT: Pre-training code representations with data flow. In *Proceedings of International Conference on Learning Representations (ICLR)*.

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 (Volume 1: Long Papers)*, pages 5690–5700, Online. Association for Computational Linguistics.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the state of semantic code search. *arXiv preprint*.

Diederik P Kingma and Jimmy Lei Ba. 2014. Adam: A method for stochastic optimization. In *Proceedings of International Conference on Learning Representations (ICLR)*.

Wei Li, Haozhe Qin, Shuhan Yan, Beijun Shen, and Yuting Chen. 2020. Learning code-query interaction for enhancing code searches. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A robustly optimized BERT pretraining approach. *arXiv preprint*.

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

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Proceedings of the Neural Information Processing Systems Track on 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. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Yu Meng, Chenyan Xiong, Payal Bajaj, Paul Bennett, Jiawei Han, Xia Song, et al. 2021. COCO-LM: Correcting and contrasting text sequences for language model pretraining. *Advances in Neural Information Processing Systems*.

Qizhe Xie, Zihang Dai, Eduard Hovy, Thang Luong, and Quoc Le. 2020. Unsupervised data augmentation for consistency training. In *Proceedings of The Conference and Workshop on Neural Information Processing Systems (NeurIPS)*.

A Case Study

We present case studies in the following sections for code search on CoSQA test set and code question answering on CoSQA validation set.

A.1 Code Search

The code snippets, shown in each case study, are retrieved from 6,267 candidate Python code functions.

Query: how to get the parent directory in python

Gold Code:
```
def get_parent_dir(name):
    """Get the parent directory of a filename."""
    parent_dir = os.path.dirname(os.path.dirname(name))
    if parent_dir:
        return parent_dir
    return os.path.abspath('.')
```

(a) A pair of real user web query and gold code function.

Top-1 Code:
```
def get_parent_dir(name):
    """Get the parent directory of a filename."""
    parent_dir = os.path.dirname(os.path.dirname(name))
    if parent_dir:
        return parent_dir
    return os.path.abspath('.')
```

Top-2 Code:
```
def get_parent_folder_name(file_path):
    """Finds parent folder of file
        :param file_path: path
        :return: Name of folder container"""
    return os.path.split(os.path.split(os.path.abspath(file_path))[0])[-1]
```

(b) Code search results with KeyDAC (top-2 results).

Figure 7: Code search results with KeyDAC on CoSQA test set. Base model is RoBERTa.

Query: get eucliedan distance between two vectors python

Gold Code:
```
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)
```

(a) A pair of real user web query and gold code function.

Top-1 Code:
```
def _euclidean_dist(vector_a, vector_b):
    """param vector_a: A list of numbers.
       param vector_b: A list of numbers.
       :returns: The euclidean distance between the two vectors."""
    dist = 0
    for (x, y) in zip(vector_a, vector_b):
        dist += (x-y)*(x-y)
    return math.sqrt(dist)
```

Top-2 Code:
```
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)
```

(b) Code search results with KeyDAC (top-2 results).

Figure 8: Code search results with KeyDAC on CoSQA test set. Base model is CodeBERT. The query has a typo *eucliedan*.

Query: python urlparse get domain

Gold Code:
```
def get_domain(url):
    """Get domain part of an url.
       For example: https://www.python.org/doc/ -> https://www.python.org"""
    parse_result = urlparse(url)
    domain = "{schema}://{netloc}".format(schema=parse_result.schema, netloc=parse_result.netloc)
    return domain
```

(a) A pair of real user web query and gold code function.

Top-1 Code:
```
def get_domain(url):
    """Get domain part of an url.
       For example: https://www.python.org/doc/ -> https://www.python.org"""
    parse_result = urlparse(url)
    domain = "{schema}://{netloc}".format(schema=parse_result.schema, netloc=parse_result.netloc)
    return domain
```

Top-2 Code:
```
def parse_domain(url):
    """parse the domain from the url"""
    domain_match = lib.DOMAIN_REGEX.match(url)
    if domain_match:
        return domain_match.group()
```

(b) Code search results with KeyDAC (top-2 results).

Figure 9: Code search results with KeyDAC on CoSQA test set. Base model is UniXcoder.

A.2 Code Question Answering

We use the CoSQA validation set, since the ground-truth labels of WebQueryTest are not provided. All cases are negative query-code pairs, such that the code snippet does not match the search intent of the query. The models predict a pair of query and code as a negative pair when the cosine similarity is lower than the threshold 0.5.

Query: python remaining blanks spaces from list

Code:
```
def split_strings_in_list_retain_spaces(orig_list):
    """
        Function to split every line in a list,
        and retain spaces for a rejoin
        :param orig_list: Original list
        :return: A List with split lines
    """
    temp_list = list()
    for line in orig_list:
        line_split = __re.split(r'(\s+)', line)
        temp_list.append(line_split)
    return temp_list
```

Label: 0

Prediction: CoCLR : **0.5123 (1)** / KeyDAC : **0.4402 (0)**

Figure 10: Case study for code question answering task on CoSQA validation set. Base model is RoBERTa.

Query: how to create wrapped socket in python

Code:
```
def connected_socket(address, timeout=3):
    """yields a connected socket"""
    sock = socket.create_connection(address, timeout)
    yield sock
    sock.close()
```

Label: 0

Prediction: CoCLR : **0.6983 (1)** / KeyDAC : **0.4699 (0)**

Figure 11: Case study for code question answering task on CoSQA validation set. Base model is CodeBERT.

Query: remove character type coloumns from dataset using python

Code:
```
def strip_columns(tab):
    """Strip whitespace from string columns."""
    for colname in tab.colnames:
        if tab[colname].dtype.kind in ['S', 'U']:
            tab[colname] = np.core.defchararray.strip(tab[colname])
```

Label: 0

Prediction: CoCLR : **0.6279 (1)** / KeyDAC : **0.3833 (0)**

Figure 12: Case study for code question answering task on CoSQA validation set. Base model is GraphCodeBERT.

Query: python use numpy array as list in code

Code:
```
def shape_list(l,shape,dtype):
    """Shape a list of lists into the
       appropriate shape and data type"""
    return np.array(l, dtype=dtype).reshape(shape)
```

Label: 0

Prediction: CoCLR : **0.5239 (1)** / KeyDAC : **0.4319 (0)**

Figure 13: Case study for code question answering task on CoSQA validation set. Base model is UniXcoder.