Semantic Code Search

Stanford CS224N {Custom} Project

Suhit Anand Pathak
Department of Computer Science
Stanford University
suhitp@stanford.edu

Dinesh Rathinasamy Thangavel
Department of Computer Science
Stanford University
rtdinesh@stanford.edu

Abstract

Semantic Code Search is the task of finding the relevant code as the output for a given natural language query. Code search is done very frequently by programmers, where the target is to reduce code duplication and reduce effort in implementing some functionalities by re-using code from the relevant libraries. These code searches are generally on web browser and in natural language. So, the exact function name and details, which could otherwise facilitate in better result, are not known. Hence, we need a mechanism to provide the required function details based on the natural language query. In this project, we have fine-tuned CodeBERT based semantic code search model to search for a function in function documents and output the most relevant function, for a given query. CodeBERT is a pre-trained model from Huggingface's transformer library designed specifically for Natural language query for a code search. With this implementation, we are able to obtain an MRR of 0.3881 for our pre-processed data.

1 Key Information to include

- Mentor: Heidi Zhang
- External Collaborators (if you have any): -
- Sharing project: -

2 Introduction

Code Search is a very common, yet important task performed by most of the programmers, who aim to re-use some of the functions already implemented in another library. This search is typically done on web browser and not in the individual function documents and are typically based on key-word matching, and not based on the context. For example, an "Array" used in the context of C is equivalent to a "List" in the context of Python programming language. However, a search for "List" for C programming on web browser provides more hit for a "Linked List" (and rightfully so) than an array. In such a case, deriving from the context becomes important.

Another difficulty in code search is that the actual code varies significantly from the natural language query. The code includes the language constructs, keywords and comments. So, an efficient algorithm should be able to derive only the required information from these programming language and code constructs. It should also be capable of projecting the query and the code to a similar vector space for a better similarity detection. This is where Natural language processing can help.

Semantic code search is a research field that has gained significant traction recently. More so with the Code search challenge(Husain et al., 2020), which encouraged computer scientists to design a more efficient semantic code search algorithm. They also provide a code Search Corpus, which is a database that can be used for training and evaluation. Some of the initial work included using Neural

Bag of word, 1D convolutional Neural network and Self attention as the text and code encoder.

More recently Feng et al. (2020) proposed CodeBERT which is a pre-trained bidirectional transformer based model for programming and natural language. CodeBERT is trained by bimodel data - which refers to the pair of natural language and code-, and unimodel data - code and natural language data individually and not as a pair. CodeBERT uses the same model architecture as RoBERTa-base (another transformer based neural architecture designed for general purpose natural language models), which is supported by Hugging Face. CodeBERT has been used by researchers from Microsoft Research Asia, Developer Division, and Bing (Lu et al., 2021) to develop a codeBERT-pipeline for benchmarking the dataset, called CodeXGlue. CodeXGlue has provided baseline for CodeSearchNet Corpus. CodeSearch Corpus, however, has the disadvantage that it uses comments/docstring provided in the code as the query input. These docstring are very detailed and do not resemble actual human query.

In our project, we have used dataset from CodeSearchNet challenge where queries are human generated and compared it to the CodeXGlue baseline. We have developed an algorithm to pre-process the query and text dataset for Python programming language and fine-tune the pre-trained pipeline provided by CodeXGlue. The baseline used is CodeXGlue's measurement for CodeSearchNet Corpus.

3 Related Work

The earliest work on semantic code search was based on lexical token of code. DeepCS (X. Gu, 2018) and CARLCS (J. Shuai, 2020) were two such state of the art models. The disadvantage of these models is that they do not look at the structure of the code. An improvement on this was suggested with PSCS (Sun et al., 2020), where the code and query are pre-processed and encoded on a similar vector space. The code is represented by AST paths. Table 1 1 provides a comparison of DeepCS, CARLCS and PSCS for the some of the metrics.

	DeepCS	CARLCS	PSCS
SuccessRate@1	14.6	17.8	22.9
SuccessRate@10	40.3	43.7	47.6
MRR	22.4	25.5	30.4

Table 1: Comparison of DeepCS, CARLCS and PSCS for JAVA dataset.

With the advent of transformer model for NLP, codeBERT (Feng et al., 2020) was introduced, which is a modification on RoBERT designed specifically for code search. It achieves an MRR in the range of $71\% - 75\%$ depending on other parameters.

4 Approach

Figure 1 1 describes the main approach of our project. The natural language query is parsed through a tokenizer to generate the query token. The original function code, in the form of a string is passed through Abstract syntax tree(AST) parser. With AST, the source code is represented in the form of a tree for better visualization of the code. Figure 2 2 (courtesy wikipedia) is an example of AST. When original function code is passed through AST parser, following information is easily abstracted - function name, function code, docstring, classes(if any), sub-functions(if any). These abstracted information is then passed through the relevant tokenizer.
Based on this abstracted information, we have tried the following 3 approaches:

1. User query tokenizer is provided as Query Input and function code tokenizer is provided as the code tokenizer.

2. User query tokenizer is provided as Query Input and function code tokenizer + docstring tokenizer is provided as the code tokenizer.

3. Docstring tokenizer is provided as Query Input and function code tokenizer is provided as the code tokenizer.

The first approach above is what we expect in the actual real world scenario, where the query is evaluated with only the code. With the second approach, we want to evaluate how the addition of docstring to code impacts the model. The third approach is more of a reference to understand how well the model works when developers comments (which should have higher similarity with code, compared to User query) are provided as the query. This pre-processing logic has been implemented by us. For generating code and query tokenizer from code and query input, we have the taken reference of the implementation by (Ramesh).

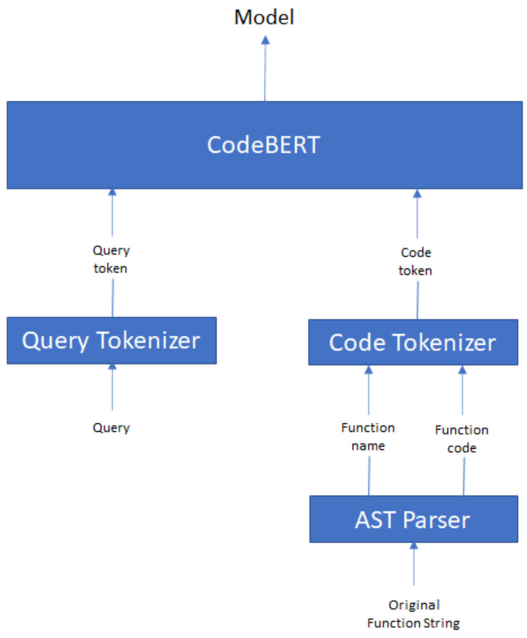


Figure 1: Main approach using pre-trained CodeBERT model

The Query token and code token described above is used to fine-tune a pre-trained CodeBERT model. CodeBERT model architecture is very similar to basic transformer architecture by Vaswani et al. (2017) - 6 identical layers for encoder and 6 identical layers for decoder. For CodeBERT, we have used the Pipeline-CodeBERT by Lu et al. (2021).

5 Experiments

5.1 Data

We have used the dataset by CodeSearchNet(Husain et al., 2020). This dataset contains two csv files - queries.csv and annotations.csv. Each row in these two files form one pair of data-point. Table 2 **??** shows the format of this data. We have combined the data from the two csv file into one table for representation simplicity. The highlighted yellow portion in Figure 3 3 shows the code pointed by one of the link from this code database.

In our pre-processing work, we have implemented the code for parsing each of the URL pointed by annotations.csv into a local structures.

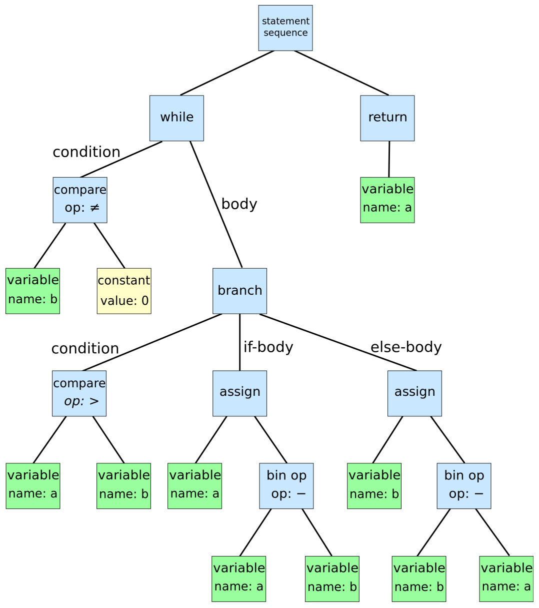
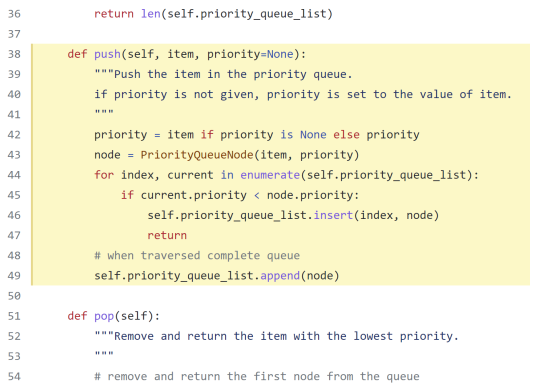


Figure 2: AST example(courtesy wikipedia)

Query	Code
sorting multiple arrays based on another arrays sorted order	Python,sorting multiple arrays based on another arrays sorted order,https://github.com/JoseAntFer/pyny3d/blob/fb81684935a24f7e50c975cb4383c81a63ab56df/pyn L33,2,
priority queue	Python,priority queue,https://github.com/keon/algorithms/blob/4d6569464a62a75c1357acc97e2dd32e $L49, 3,$

Table 2: Few rows from CodeSearNet database



```
        return len(self.priority_queue_list)

    def push(self, item, priority=None):
        """Push the item in the priority queue.
        if priority is not given, priority is set to the value of item.
        """
        priority = item if priority is None else priority
        node = PriorityQueueNode(item, priority)
        for index, current in enumerate(self.priority_queue_list):
            if current.priority < node.priority:
                self.priority_queue_list.insert(index, node)
                return
        # when traversed complete queue
        self.priority_queue_list.append(node)

    def pop(self):
        """Remove and return the item with the lowest priority.
        """
        # remove and return the first node from the queue
```

Figure 3: Code example

5.2 Evaluation method

The evaluation metric used is MRR(Mean Reciprocal Rank) defined as follows:

$$MRR = \frac{1}{Q}\sum_{i=1}^{Q}\frac{1}{rank_i + 1}$$

where Q are the total number of queries which have found a match in the dataset which are also listed in the true answer, $rank_i$ is the rank of this code match. The queries whose search result does not predict a code listed in the true answer list, does not contribute to this sum.

5.3 Experimental details

Following are the model configuration used for this project:

1. Adapted Model - Pipeline CodeBERT (Feng et al., 2020)
2. Model Type - RoBERTa
3. Pre-trained Model - Microsoft CodeBERT-base
4. Learning Rate - 5e-5
5. Maximum Gradient Norm - 1.0
6. Number of Epochs - 50
7. Total training data size - 370
8. Total valid data size - 80
9. Total test data size - 72
10. Training Batch size - 32
11. Evaluation Batch size - 64
12. Similarity Metric - Cosine Similarity
13. Loss function - Cross Entropy Loss

5.4 Results

The baseline numbers used is CodeXGlue's evaluation on CodeSearchNet Corpus. This Corpus uses docstrings as query. However, our implementation uses datsbase of CodeSearch Challenge. This has human provided queries. Since, the two databases are different, our Approach 3 3 can be considered a baseline. Table 3 3 provides a comparison of MRR for the different approaches.

Approach	MRR
CodeXGlue	27.19
Approach 1 1	38.81
Approach 2 2	43.35
Approach 3 3	37.88

Table 3: MRR Comparison of various approaches.

Figure 4 4, Figure 55 and Figure 6 6 provides a plot of score for different queries in the database for the three approaches. A score of 1.0 is the highest while a score of 0.0 is the lowest. It should be noted that query 0 of approach 1 does not correspond to query 0 of approaches 2 and 3. This is because of the random nature in which the queries are grouped into train, valid and test database. Hence, the three plots are seperately provided and should be viewed as such.

The approach 2 performs the best and this is probably expected since approach 2 includes the docstring tokens. These docstring tokens are typically in natural language and intuitively this is expected as it provides a natural language to natural language comparison for the queries. Approach 3 not performing better is surprising as the expectation was that docstring as token would have helped in better matching as it is more detailed compared to a User query.

6 Analysis

Your report should include *qualitative evaluation*. That is, try to understand your system (e.g. how it works, when it succeeds and when it fails) by inspecting key characteristics or outputs of your model.

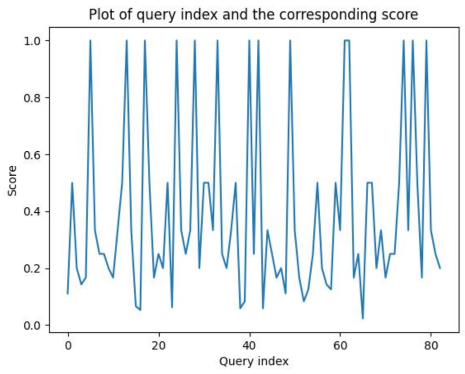


Figure 4: Plot of score for every query when User query tokenizer is provided as Query Input and function code tokenizer is provided as the code tokenizer.

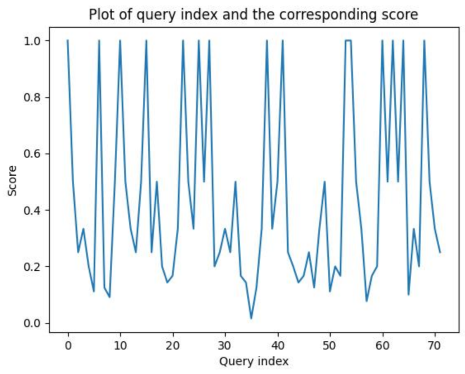


Figure 5: Plot of score for every query when User query tokenizer is provided as Query Input and function code tokenizer + docstring tokenizer is provided as the code tokenizer.

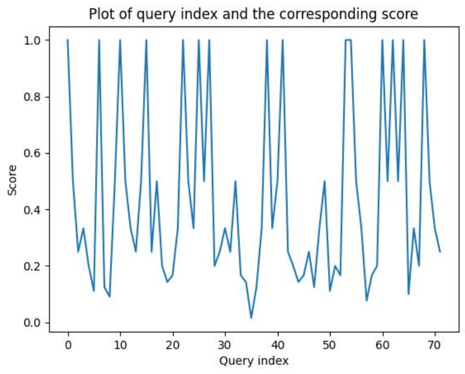


Figure 6: Plot of score for every query when Docstring tokenizer is provided as Query Input and function code tokenizer is provided as the code tokenizer.

7 Conclusion

Summarize the main findings of your project, and what you have learnt. Highlight your achievements, and note the primary limitations of your work. If you like, you can describe avenues for future work.

References

Zhangyin Feng, Daya Guo2, 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 *arXiv*.

Hamel Husain, Miltiadis Allamanis, Ho-Hsiang Wu, Marc Brockschmidt, and Tiferet Gazit. 2020. Codesearchnet challenge evaluating the state of semantic code search. In *arXiv*.

C. Liu M. Yan X. Xia Y. Lei J. Shuai, L. Xu. 2020. Improving code search with co-attentive representation learning. In *28th International Conference on Program Comprehension - ICPC*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin 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.

Shashank Ramesh. Semantic code search using bert and transformer.

Zhensu Sun, Yan Liu, Chen Yang, and Yu Qian. 2020. Pscs: A path-based neural model for semantic code search. In *arXiv*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *arXiv*.

wikipedia. Ast.

S. Kim X. Gu, H. Zhang. 2018. Deep code search. In *40th International Conference on Software Engineering - ICSE*.