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 from Python database. 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 perform better than the baseline.

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 bimodal data - which refers to the pair of natural language and code, and unimodal 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 HuggingFace library. 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. CodeSearchNet 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 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 CodeBERT Architecture

Since our project uses CodeBERT as the model for our experiments, it would be only reasonable to understand the architecture of this model. This model is entirely built on the principle of ELECTRA (Clark et al., 2020) which tries to improve "effective contextual learning", one of the disadvantages of MLM pre-training methods such as BERT. The way ELECTRA solves this is by instead of predicting only 15% inputs that are masked out, adds a new RTD (Replaced Token detection) pre-training task. This pre-training task will require the model to determine which tokens from origina input have been replaced or kept the same. This classification task is applied to every input token, making it more efficient than MLM. CodeBert is exactly using the same technique to make the model effectively learn the representation of code and NL. CodeBERT uses unimodal generators individually for Code and Natural language which will generate softmax distributed masked language model. The output from the generators are fed to NL-Code discriminator which will identify if each of the token is replaced or original. After pre-training, the discriminator model is fine-tuned for NL CodeSearch task. The architecture from the CodeBERT paper is shown in figure 1:

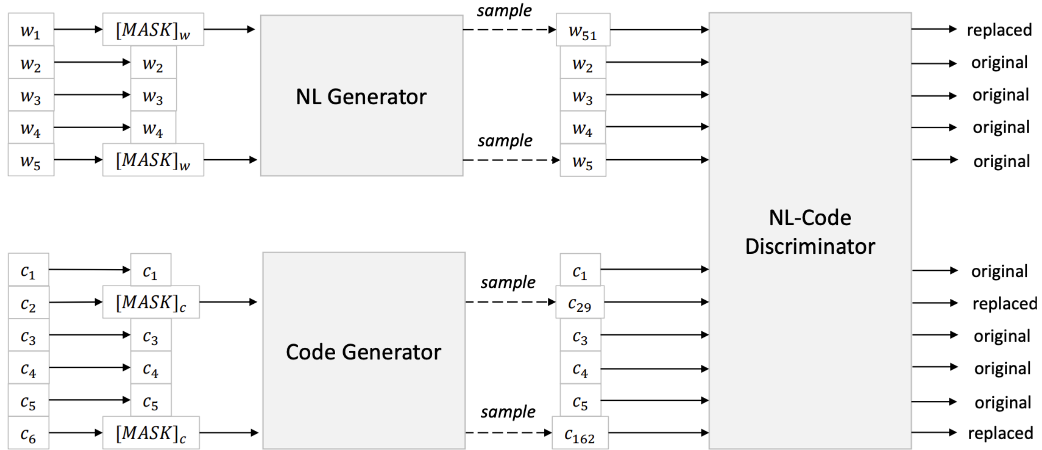


Figure 1: Architecture of CodeBERT, courtesy of (Feng et al., 2020)

5 Approach

Figure 3a 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 3b (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.

Figure 2 shows a sample data point. The original code consists of a docstring input provided by the code developer at the time of developing the code, while User input is the Natural language query for the database. Table 2 shows the various approaches that we have used for deciding the query token and code token of Figure 3a .

```
===============================================
=========== Example # 1 =======================
Original Code:
    def clone_with_new_elements(
            self,
            new_elements,
            drop_keywords=set([]),
            rename_dict={},
            extra_kwargs={}):
        """
        Create another Collection of the same class and with same state but
        possibly different entries. Extra parameters to control which keyword
        arguments get passed to the initializer are necessary since derived
        classes have different constructors than the base class.
        """
        kwargs = dict(
            elements=new_elements,
            distinct=self.distinct,
            sort_key=self.sort_key,
            sources=self.sources)
        for name in drop_keywords:
            kwargs.pop(name)
        for old_name, new_name in rename_dict.items():
            kwargs[new_name] = kwargs.pop(old_name)
        kwargs.update(extra_kwargs)
        return self.__class__(**kwargs)

User Query:
hash set for counting distinct elements
===============================================
```

Docstring

Original Code

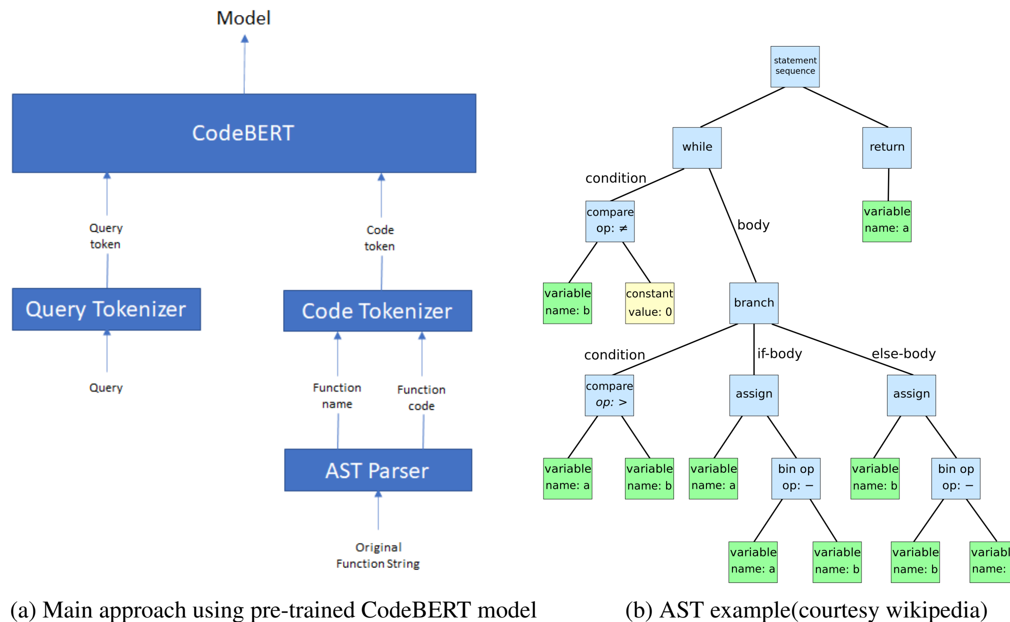
User Input

Figure 2: Sample code

Approach	Code token	Query Token
Approach 1	Original Code token	User Input token
Approach 2	Original Code + Docstring token	User Input token
Approach 3	Original Code token	Docstring token

Table 2: Approaches taken for preprocessing data

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 taken reference of the implementation by Ramesh.



(a) Main approach using pre-trained CodeBERT model

(b) AST example(courtesy wikipedia)

Figure 3: Main approach and AST example

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).

6 Experiments

6.1 Data

There are three different datasets that we used for this project, out of which two of them are from CodeSearchNet(Husain et al., 2020). We will explain in detail about each of the dataset.

1. CODESEARCHNETCORPUS which consists of 6 million functions from open-source code in six programming languages (Go, Java, JAVASCRIPT, PHP, Python and Ruby). For

simplicity, we only experimented on Python data, which amounted to 45K functions and docstrings. This dataset is expected to be self-supervised, so they have parsed the code (functions) and docstrings (NL queries) in the repos and created JSONL files. They follow a specific format, which will be the common format for all our datasets. We will explaing the format in detail below.

2. CODESEARCHNETCHALLENGE consists of 99 natural language queries along with likely results for each of the languages mentioned above. Each query/result pair is labeled by human expert. The main distinction between this and CODESEARCHNETCORPUS is that the queries resemble the human language used while searching than the language used by programmers during documentation of a function. Illustrating the same with an example from the dataset
 In the Figure 4, we have three different examples, which explains the problem with

```
============================================


Example 1: Code has no docstrings
------------C   O   D    E (1) -------------
['def pack_unsigned_int(number, size, le):\n',
 '  if not isinstance(number, int):\n',
 '    raise StructError("argument for i,I,l,L,q,Q,h,H must be integer")\n',
 '  if number < 0:\n',
 '    raise TypeError("can\'t convert negative long to unsigned")\n',
 '  if number > (1 << (8 * size)) - 1:\n',
 '    raise OverflowError("Number:%i too large to convert" % number)\n',
 '  return pack_int(number, size, le)\n']
------------Q    U    E    R    Y  (1) -------
'convert string to number'
============================================


Example 2: Code has matching docstrings
------------C   O   D    E   (2) -----------
['def convert_string_to_number(value):\n',
 '    """\n',
 '    Convert strings to numbers\n',
 '    """\n',
 '    if value is None:\n',
 '        return 1\n',
 '    if isinstance(value, int):\n',
 '        return value\n',
 '    if value.isdigit():\n',
 '        return int(value)\n',
 "    num_list = map(lambda s: NUMBERS[s], re.findall(numbers + '+', "
 'value.lower()))\n',
 '    return sum(num_list)\n']
------------Q    U    E    R    Y  (2) -------
'convert string to number'
============================================


Example 3: Code has irrelevant docstrings
------------C   O   D    E  (3) ------------
['    def compute_bridge_similarity(self, vec1, vec2):\n',
 '        EWP = 1 - np.multiply(vec1, vec2)\n',
 '\n',
 '        # not sure exactly how to sort the EWP vector\n',
 '        #EWP = sorted(EWP, reverse=True)\n',
 '        EWP = sorted(EWP, reverse=True)\n',
 '\n',
 '        k = 10\n',
 '        EWP = EWP[:k]\n',
 '\n',
 '        # The paper does not mention using logs but start to get into '
 'underflow\n',
 '        # issues multiplying so many decimal values\n',
 '        lEWP = -1 * np.log(EWP)\n',
 '        return 1/np.sum(lEWP)\n']
------------Q    U    E    R    Y  (3)--------
'how to reverse a string'
============================================
```

Figure 4: Dataset from SEMANTICCODESEARCH

CODESEARCHNETCORPUS. Example 1, doesn't have any docstrings in the function. Example 2, has a docstring which translates to the meaning of the function. Example 3, has docstrings, but its not at all relevant to the NL Query that humans would use to search for this function. We identified this as a problem, and worked on generating the SEMANTICCODESEARCHCORPUS.

3. SEMANTICCODESEARCHCORPUS consists of approx. 500 sets of (c_j, q_j) pairs generated from the CODESEARCHNETCHALLENGE inputs of $query_j, url_j$ with our custom script in the same format as CODESEARCHNETCORPUS. The format for the JSONL files are

documented in the (Lu et al., 2021) GitHub Repo. Since CODESEARCHNETCHALLENGE doesn't have the supervised labels in that format, we worked on a simple python script where for each query in the dataset, we pull the code from GitHub based on URL, and then parsed the functions into code and docstrings, but then replaced the docstrings with the actual query annotated by experts. With this technique, we are able to fine tune the model better.

6.2 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.

Score of individual query is calculated as follows:
$score = \frac{1}{rank_i+1}$, if predicted output is listed as possible answer in the test dataset
= 0 , otherwise

6.3 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. Loss function - Cross Entropy Loss

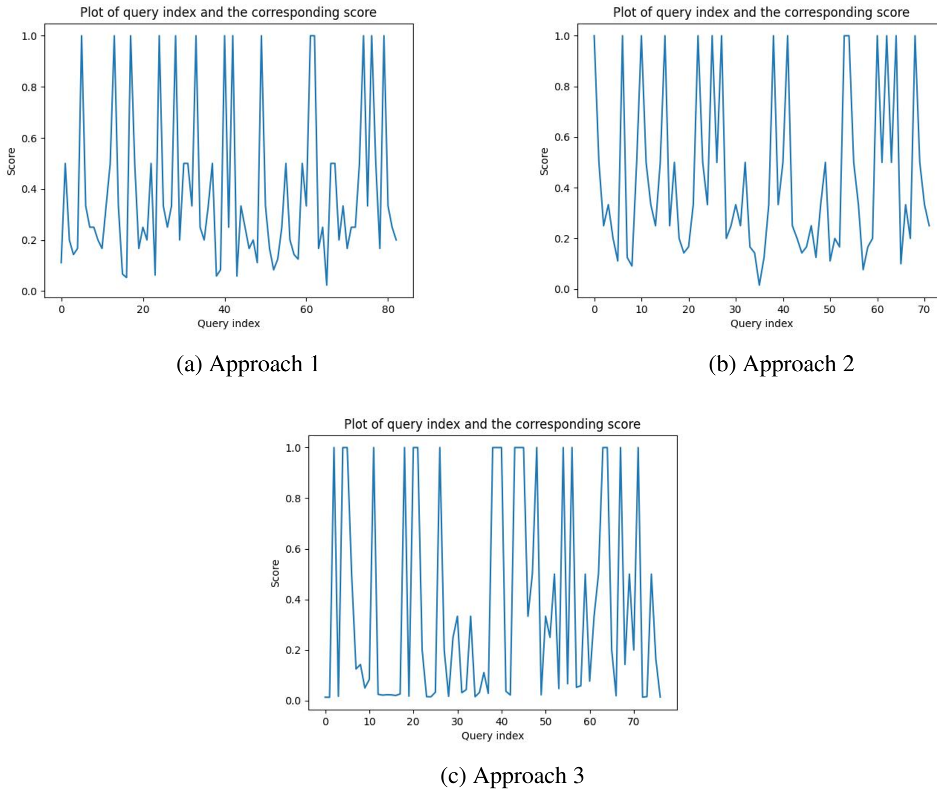
6.4 Results

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 2 can be considered a baseline. Table 3 provides a comparison of MRR for the different approaches.

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

Table 3: MRR Comparison of various approaches.

Figure 5 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.



(a) Approach 1

(b) Approach 2

(c) Approach 3

Figure 5: Plot of score for every query for the three approaches.

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.

7 Analysis

1. The scores indicate that many queries are accurately predicted for all approaches. Few queries result in answer with the best possible rank.
2. Including docstring along with code(Approach 2 2) gives better result as in that case there is a natural language to natural language comparison.
3. Approach 3 does not give better results. So, smaller queries give better result.

8 Conclusion

We were able to get good results with our model. Many queries predicted the best(Rank 0) code. This approach of feeding data with human feedback to CodeBERT model for analysis, has not been explored by anyone so far.

However, CodeBERT model is advanced. The dataset that we used was not big enough, hence we could not use models full potential. Maybe with a larger dataset, we could have seen better MRR values with our approaches.

There is a recent development of using RL on human feedback. In future, we would like to try that with this model.

References

Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. Electra: Pre-training text encoders as discriminators rather than generators. In *International Conference on Learning Representations*.

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*.