CSE538 Project Report - CODEMANTIC (Semantic Code Search)

Sharvil Katariya
112950477
skatariya@cs.stonybrook.edu

Sourav Sarangi
112686475
sosarangi@cs.stonybrook.edu

Rutvik Parekh
112687483
ruparekh@cs.stonybrook.edu

Abstract

Given a natural language query, semantic code search is the task of retrieving relevant code which performs the function that is expressed via the mode of natural language query. This task has been proven to be quite important for day-to-day software development and maintenance activities. However, existing code search mechanisms do not take into account the underlying semantics of the code such as program structure and control flow, which is mainly the reason of why a code works how it works. Here, we implement few models which do the task of semantic code search and present the evaluation results based on widely used metric Mean Reciprocal Rank (MRR).

1 Introduction

Deep Learning has been widely used for tasks such as Question Answering, Relation Extraction, Machine Translation, Text Summarization and myriad of other tasks. This has also been become possible by availability of large datasets, significant improvement in computational power, and explosion of various machine learning models in the scene.

But, deep learning models still struggle on highly structured data. This is evident in tasks like semantic code search. Although, searching on images, and natural language documents have made considerable amount of progress, search for appropriate code is still grappling and often produces incorrect results. Previous works done in this area have been done by the mechanism of information retrieval systems. For example, (Linstead et al., 2009) put forward Sourcerer, that combines textual content of a program with structural information. (Lv et al., 2015) presented CodeHow, which combines text similarity and API matching through an extended Boolean model. (McMillan et al., 2011) proposed Portfolio, which returns a chain of functions through PageRank and keyword matching.

However, a foundational problem of the information-based retrieval system is the mismatch between the high-level meaning, conveyed in natural language queries and low-level implementation details in the source code. Natural language and source code are diverse entities. They may or may not share syntactic structure, lexical tokens, and synonyms.However, they are semantically related. For example, a natural query like ”Download a file over the internet” may have the relevant code as shown in Figure 1.

Already proposed solutions based on Information Retrieval may not be able to return this result as it does not contain the word ’download’ or any of its immediate synonyms. Hence, an effective code search system needs a higher-level semantic mapping between natural language queries and source code. Evaluating methods for IR based tasks is even harder, as there are no substantial datasets that were created for this task; instead, the community tries to cope with small datasets from related contexts (e.g. pairing questions on web forums to code chunks found in answers).

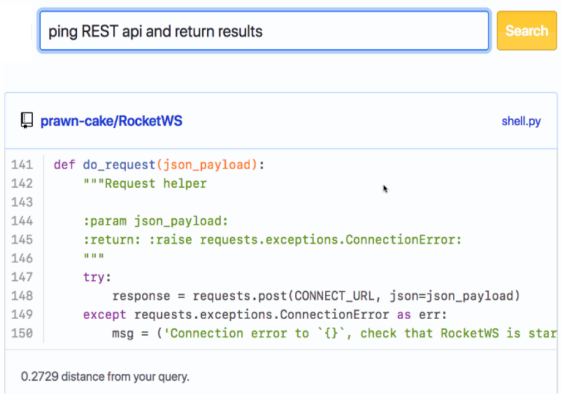

Figure 1: Search Engine for Code Search

Due to the problems mentioned above, an effective method for semantic code search is needed that takes into account the lexical gap between natural language queries & source code. Here,

we have used (Husain et al., 2019) as a motivation & an opportunity to implement various models & evaluate them to find out the best performing among them. Specifically, we implement the models as mentioned in Section 3.

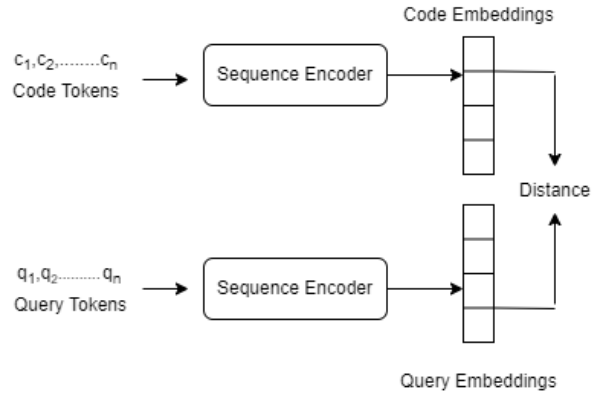

Figure 2: Method used to generate combined code and query representations

2 Approach

We have prepared our dataset after preprocessing the code and documentation on the basis of the steps mentioned in 4.1. Our methodology comprises of the following steps:

1. **Code Encoder**: In order to generate code representations we have tried out various Seq2Seq models like Bidirectional RNN, Self Attention, etc.

2. **Query Encoder**: Build language mode to be a used a general purpose sentence encoder in order to encode the search queries.

3. **Code to Query Encoder**: Fine - tune the code encoder to map the code representation into the same latent space as that of natural language. In order to ensure that the representation of the pair (c_i, d_i) for a corresponding dataset appear closeby, we use inner product and concept similar to negative sampling to generate the training loss. Let E_c and E_q be the code and query encoding, we aim to minimize the following loss $L(\theta)$ for N pairs of (c_i, d_i) (code : (c_i and it 's documentation d_i)):

$$\mathbf{L}(\theta) = \frac{-1}{N} \sum_i \log\left(\frac{exp(E_c(c_i)^T E_q(d_i))}{\sum_j exp(E_c(c_j)^T E_q(d_i))}\right)$$

Here, c_j is the distractor snippets ($i \neq j$), wherein we try to maximize the inner product between the code representation of c_i and the query representation of d_i, while minimizing the inner product between each c_i and the distractor snippet c_j. Inner product has been shown to give better results than cosine similarity for training purpose.

4. **Semantic Search**: During prediction time, we use the Query Encoder as built in Step 2. Then, we look for the closest code matches in the fine-tuned code representation space. This can be done with the help of building indexes to do approximate searches using different Approximate Nearest Neighbour benchmarks during testing. This can be done using ANNOY (Bernhardsson, 2013), FAISS (Johnson et al., 2017) FLANN (Muja and Lowe, 2009). In our implementation, we have relied on Annoy indexes building 10 trees.

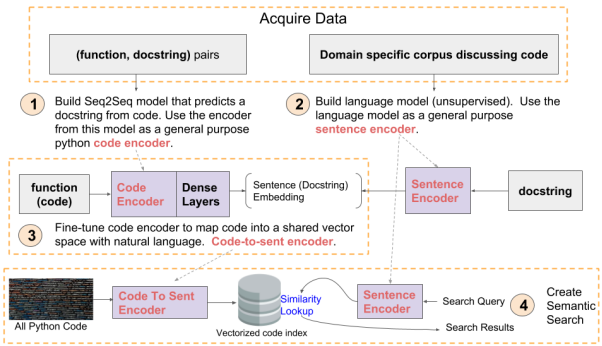

Figure 3: Updated Architecture

As can be seen in Fig. 3, we train a sequence-to-sequence model that learns to summarize code, such that the docstring is the target variable for the code. We can then fine-tune the Code Representations to the Same Vector-Space as Text. The input code is split into tokens according to semantics into subtokens (for example: binarySearch becomes binary search). For queries we use Byte Pair Encoding (Sennrich et al., 2015) for building the subword information as well as for tokenisation to generate natural language tokens.

3 Models

Once we have the tokens generated for code as well as natural language, the token sequences are processed to obtain contextualised token embeddings. The token embeddings are then combined

into a sequence embedding using a pooling function, for which we have implemented mean/max-pooling and an attention-like weighted sum mechanism.

1. **Neural Bag of Words**: In this model, each (sub) token is embedded to a learnable embedding (vector representation).

2. **Bidirectional RNN model**: Here, we employ the GRU cell to summarize the input sequence.

3. **1D Convolutional Neural Network**: In this model, we take one-dimensional convolution over the input sequence of tokens.

4. **Self Attention model**: In this model, multi-head attention is used to compute representations of each token in the sequence

5. **Convolutional Self Attention model**:

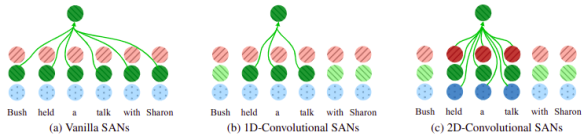

Figure 4: Illustration of (a) vanilla SANs; (b) 1-dimensional convolution with the window size being3; and (c) 2-dimensional convolution with the area being33. Different colors represent different subspaces modeled by multi-head attention, and transparent colors denote masked tokens that are invisible to SANs.

   (a) As seen, Convolutional SANs are only allowed to attend to the neighboring tokens, instead of all the tokens in the sequence. For each query q_i^h, we restrict its attention region (e.g., $K^h = k_1^h, ..., k_i^h, ..., k_I^h$) to a local scope with a fixed size $M+1 (M \leq \text{I})$ centered at the position i:

$$\hat{\boldsymbol{K}}^{\boldsymbol{h}} = k_{i-M/2}^h, ..., k_i^h, ..., k_{i+M/2}^h$$
$$\hat{\boldsymbol{V}}^{\boldsymbol{h}} = v_{i-M/2}^h, ..., v_i^h, ..., v_{i+M/2}^h$$

   This SAN-based model is implemented as multiple layers, in which higher layers tend to learn semantic information while lower layers capture surface and lexical information.

   (b) Multihead mechanism allows different heads to capture distinct linguistic properties, especially in diverse local context. We can expand the 1-dimensional window to a 2-dimensional area with the new dimension being the index of attention head. Suppose that the area size is (N+1)(M+1)($N \leq$H), the keys and values in the area are:

$$\tilde{\boldsymbol{K}}^{\boldsymbol{h}} = \bigcup(\hat{K}^{h-N/2}, .., \hat{K}^h, .., \hat{K}^{h+N/2})$$
$$\tilde{\boldsymbol{V}}^{\boldsymbol{h}} = \bigcup(\hat{V}^{h-N/2}, ..., \hat{V}^h, ..., \hat{V}^{h+N/2})$$

   The 2D convolution allows SANs to build relevance between elements across adjacent heads,thus flexibly extract local features from different subspaces rather than merely from an unique head

4 Evaluation

4.1 Dataset Details

We use the dataset made publicly available by (Husain et al., 2019) to train and test our models. They have created a dataset known as **CodeSearchNet Corpus**.They have paired functions in open-source software with the natural language present in their respective documentation. However, to do so requires a number of preprocessing steps and heuristics. For making the dataset **CodeSearchNet Corpus**, they collected the corpus from publicly available open-source non-fork GitHub repositories, using libraries.io to identify all projects which are used by at least one other project, and sorted them by **popularity** as indicated by the **number of stars and forks**. Then, they removed any projects that did not have a license or whose license did not explicitly permit the re-distribution of parts of the project. They then **tokenized** all **Go, Java, JavaScript, Python, PHP and Ruby** functions (or methods) using **TreeSitter** – GitHubs universal parser and, where available, their respective documentation text using a heuristic regular expression.

Only those functions in the corpus that have documentation associated with them have been considered. This yields a set of pairs (c_i, d_i) where c_i is some function documented by d_i. They then implement a number of preprocessing steps:

- For **documentation** d_i, it is truncated to the first full paragraph, to make the length more comparable to search queries.
- Pairs in which d_i is shorter than three tokens are removed, as those comments are not very informative.

- Functions c_i whose implementation is shorter than three lines are removed.
- Functions whose name contains the substring test are removed. Similarly, we remove constructors and standard extension methods such as __str__ in Python
- Duplicates are removed from the dataset by identifying (near) duplicate functions and only keeping one copy of them using (Allamanis, 2019) and (Lopes et al., 2017).

The dataset contains about 2 million pairs of function-documentation pairs and about another 4 million functions without an associated documentation. We split the dataset in 80-10-10 train/valid/test proportions.

4.2 Evaluation Metrics

We have used the Mean Reciprocal Rank (MRR) evaluation metric to evaluate all our models. It is a statistic measure for evaluating any process that produces a list of possible responses to a sample of queries, ordered by probability of correctness.

Let FRank be the rank of the first hit result in the result list and Q be the list of queries evaluated on. Then **Mean Reciprocal Rank** is computed as follows:

$$\text{MRR} = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\frac{1}{FRank_q}$$

4.3 Results

For our evaluation we have trained our models on 412178 Python samples and validated on 23107 python samples. For all of our evaluations we have made the embedding space dimension to be 128, and run the models for 12 epochs.

Model Name	Validation MRR	Test MRR	Training Time
Neural Bag of Words	0.4674	0.642	26m 12s
Bidirectional RNN	0.5016	0.629	1h 35m 47s
1D CNN	0.4554	0.516	32m 18s
Self Attention	**0.5815**	**0.667**	1h 11m 41s
Convolutional Self Attention	0.5591	0.587	1h 1s

Table 1: MRR scores on Validation & Test Dataset

Table 1 shows the final validation MRR as obtained on 60K python samples. Figure 5, shows the validation MRR on running for different epochs for all the models as described above. We clearly see that Self Attention models is performing the best followed by Convolutional Self Attention with MRR of 0.5815 & 0.5591 respectively. For testing purposes on CodeSearchNet Corpus, a set of 999 distractor snippets c_j is generated for each test pair (c_i, d_i) and is tested on all trained models. Here, we see that Self Attention Neural Bag of words performs better than the rest with test MRR of 0.667 & 0.642 respectively. We see that the bag of words model is particularly good at **keyword matching**, which seems to be a crucial facility in implementing search methods. As we see Self Attention model is performing the best, as the self-attention model has the **highest capacity** of all considered models.

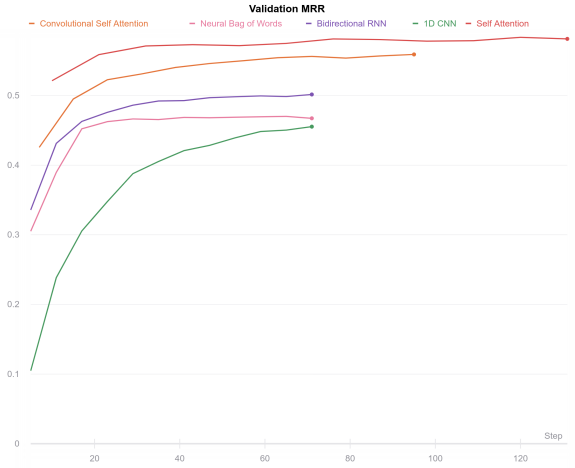

Figure 5: MRR Scores on Validation Dataset for different models

4.4 Analysis

- For Self Attention model, we see that for some cases, the word order is not taken into account. For example, for the query **"convert int to string "**, the top result returned is code for converting any value to int. This is the because of the inverse functionality of the given query, e.g. convert int to string would be answered by stringToInt. This suggests that the baseline models used for pre-filtering have trouble with understanding such semantic aspects. Convolutional Self-Attention also falls into the same trap but the results are accentuated by locality. The code returned by it is specifically for converting string to int. The NBOW model also suffers from positonal indifference and hence the results from NBOW model are also from string to int.

- For another query where we are trying **extract data from HTML content** the result returned by Convolutional model is more aligned with the goals as it returns a function for extracting title, microdata, metadata, json, rfda and other kinds of HTML

page embeddings but in case of self-attention model it seems to have learned the representations globally and hence returns more general parse function but it fails to acheive the objective as the function is for parsing the URL to generate more nested URLs.

- In another example, we are giving a natural language query to "**convert a date string to yyyymmdd format**", and the top result for this by the Convolutional self-attention model is **datedif**, which calculates the difference between two given dates, instead of converting the date string to yyyymmdd format. This tells us that it fails to capture the complete semantic meaning of the natural language query or the source code.

5 Challenges

- The CodeSearch data covers a wide range of general-purpose code queries. However, anecdotal evidence indicates that queries in specific projects are usually **more specialized**. Adapting search methods to such use cases could yield substantial performance improvements.

- **Code quality** of the searched snippets was a recurrent issue with the CodeSearch's expert annotators. Despite its subjective nature, there seems to be agreement on what constitutes very bad code. Using code quality as an additional signal that allows for filtering of bad results (at least when better results are available) could substantially improve satisfaction of search users.

- **Proprietary code search** is an issue as we cannot always rely on Google search and StackOverflow results. Proprietary code contains much accurate code for any desired functionality.

- Learning which parts in the representation are relevant to prediction of the desired property (**Query Relevance**) and learning the order of importance of the part - Path Based Attention Model

6 Code

The publicly available link to the Github repository of our project is: https://github.com/scorpionhiccup/SemanticCodeSearch.

Commands needed to run:

1. pip install -r requirements.txt
2. bash main.sh This command will install and run all the models as part of this project.

List of Major Software Requirements

- Tensorflow (2.0.0)
- Docker
- Torch
- Torchvision
- Nvidia-Docker
- Docker-Compose
- graphql-core (2.0.0)
- Keras (2.3.1)
- annoy

7 Conclusion

We have experimented with different encoding models for encoding the code & query representations and then created a joint embedding space to search the relevant query with the code representation's space. We have tried out a new sequential model called Convolutional Self Attention, however, we did not find it to perform better than Self Attention model, which clearly seems to emerge victorious. We have also experimented with using **Concrete Syntax Tree** (CST) instead of AST representation for code's tokens.

In the evaluation of our models we found embedding sequential information into the model would help as we were loosing out to capture postional relationship between tokens. In the (Shaw et al., 2018) paper on , we see that incorporating **relative positional information** greatly improves accuracy for English to German with BLEU score increase by 1.2%.

Furthermore, Tree-based LSTMs (Tai et al., 2015), (Dam et al., 2018) and syntax aware tokenization can be explored to improve the code summarizer and incorporate the syntactic structure of code into the encoder. Further improvement can be done by leveraging the control and data flow of the code to further improve code representations.

References

Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of

code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 143–153. ACM.

Erik Bernhardsson. 2013. Annoy: Approximate nearest neighbors in c++/python optimized for memory usage and loading/saving to disk, 2013. *URL https://github. com/spotify/annoy*.

Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim, and Chul-Joo Kim. 2018. A deep tree-based model for software defect prediction. *arXiv preprint arXiv:1802.00921*.

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*.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2017. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*.

Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. 2009. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18(2):300–336.

Cristina V Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. Déjàvu: a map of code duplicates on github. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):84.

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

Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 111–120. ACM.

Marius Muja and David Lowe. 2009. Flann-fast library for approximate nearest neighbors user manual. *Computer Science Department, University of British Columbia, Vancouver, BC, Canada*.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-attention with relative position representations. *arXiv preprint arXiv:1803.02155*.

Kai Sheng Tai, Richard Socher, and Christopher D Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*.