International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 07 Issue: 05 | May - 2023 SJIF 2023: 8.176 ISSN: 2582-3930

Neural Code Search with Unified Deep Semantic

Asst. Prof. Ms. Suvarna Unde – RGCOE Karjule Harya, Computer Engineering
Mr. Parshuram Pathave – RGCOE Karjule Harya, Computer Engineering
Mr. Harshal Kanwade – RGCOE Karjule Harya, Computer Engineering
Mr. Sandesh Dhonge – RGCOE Karjule Harya, Computer Engineering
Ms. Komal Gadakar – RGCOE Karjule Harya, Computer Engineering

***Abstract**: A tool that can search over large code corpus directly and list ranked snippets can prove to be an invaluable resource to programmers looking for similar code snippets using natural language queries. It must have a deep understanding of the semantics of source code and queries to evaluate their intent correctly. Over the years, many tools that rely on the textual similarity between source code and query have proven to be ineffective as they fail to learn the high- level semantic understanding of source code and query. While the previous models for code search using deep neural networks do a good job but, most of them only evaluate their models on only a single programming language, mostly Java. In this paper, we propose a novel deep neural network model called Unified Code Net that can handle the intricacies of different programming languages. This model borrows several vital features from different previous models and builds on top of those ideas to make a unified model that can generate document vector embeddings from source code, and using similarity search with the query vector embedding can return the most similar code snippets in any language. This tool can drastically reduce the programmer's efforts to look for an efficient and viable code snippet for problem at hand which ideally can replace use of search engines for the same.*

***Keywords**: semantic code search, natural language processing, information retrieval*

I. INTRODUCTION

Code Search can provide a massive boost in productivity of programmers as the recent uptick in the use of deep learning for code search, and rise of computing power has made it possible to retrieve related code from a massive code corpus that matches programmer's intent from natural language queries. This saves the programmer from the hassle of Google Searching for related code snippets to get something done or endless browsing of community forums like StackOverflow looking for possible usage of a proprietary API or some standard coding problems/algorithm implementation. This, even though troublesome might be fruitful for well-known and used languages like Python, Java, and C++, but for lesser-known languages or proprietary API with evolving communities, the answers may not exist at all in such forums. Semantic Code Search makes it possible to search for such snippets directly using natural language queries and get ranked semantically similar code snippets of a particular required language. In the implementation of the model, we leveraged the power of Open Source tools like FAISS [10], fastText [11] as well as Code Repositories like GitHub [4] to collect code snippets from public repositories using tools/techniques and benchmarks provided by

CodeSearchNet [6].

UnifiedCodeNet converts the code snippets into document vectors, and those vectors are mapped on a shared space where all the document vectors are mapped. When a user searches using a query, the query is also converted to a document vector and mapped to the same shared space and using similarity search [7] closest document vectors of required language are fetched and ranked according to similarity.

For example, a simple query like ***"How to read text file line by line?"*** returns ranked snippets in the required language. Below are the actual top results from the query in three languages PHP, Go, and Java.

```
public static function readLines(string $filePath):\Generator
{
if (! $fh = @fopen ( $filePath , ’r’)) {
throw new \ InvalidArgumentException (’Error :’. $filePath );
}
return self :: read ($fh);
}
```

Top Result for PHP - Query: Read Text File Line by Line

```
func (cr * countingReader ) Read (p [] byte ) (int , error )
{
n, err := cr. readerImpl . Read (p)
if recordInputHexdumpFlag && err == nil {
fwColLog . Debugf ("Hex dump of %d input bytes :\n% sEnd dump
of %d input bytes ",
len(p), hex. Dump (p), len(p))
}
cr. inputBytes += int64 (n)
return n, err
}
```

Top Result for Go - Query: Read Text File Line by Line

```
public String readLine () throws IOException {
StringBuilder result = new StringBuilder ();
for (;;) {
int intRead = read ();
if ( intRead == -1) {
return result . length () == 0 ? null : result . toString
();
}
char c = ( char ) intRead ;
if (c == ’\n’ || c == ’\r’) break ;
result . append (c);
}
return result . toString ();
}
```

Top Result for Java - Query: Read Text File Line by Line

In the above examples, the document vectors and query vector are semantically similar and are mapped closely, which means that the model has a high-level understanding of what the function does and similarly what the query intends to find.

II. DATASET

For creating a dataset for our supervised learning model, we lever- aged the power of Open Source GitHub repositories. Source Code is fetched from GitHub using the tools and techniques specified in CodeSearchNet. The corpus we created contains about 6 million functions from open-source code spanning six programming languages (Go, Java, JavaScript, PHP, Python, and Ruby). More than 2 million functions in the dataset contain that description of the function in natural language, which is present in the source code in the form of comments or docstrings. The main reason for choosing this dataset for our model training is the scope of expanding the dataset with relative ease and still get the same quality of data as the current one. As our model is a supervised one, it will only perform better with a larger training set. However, we believe that for proof of concept, the current dataset will suffice.

A. Preprocessing

CodeSearchNet uses TreeSitter - GitHub's universal parser to ex- tract pairs of functions and docstring. The parser focuses on extracting code and descriptions pairs from the source code if the description does not exist the description is scrapped from documentation of the software if one exists. For tokenization, Firstly, all the special characters are removed from the function as they are unnecessary and cannot be used for creating document embeddings of the code. Secondly, the tokens are broken into individual tokens from their representation in camelCase, which would be broken into [camel, case] or snake_case, which would split into [snake, case]. Also, during this, the tokens are converted to lowercase. Further, all the duplicate tokens are removed. In case of functions that are part of a class, the class name is also tokenized and added to the list of tokens as it helps in providing context to the function.

B. Filtering

To maintain the quality of the dataset, some filtering techniques are employed as duplicate code has adverse effects in the machine learning model of code [1]. As open-source codes are full of duplicates, whether it be the case of copy-pasting or multiple versions of auto-generated code. Further, functions like constructors, destructors, getters, setters, or inbuilt functions are not included. Moreover, functions with short/poor descriptions are also not included as they are not informative.

C. Limitations

Even though this dataset is better than what the community has tried to conjure up over the years, it does have some shortcomings. Unsurprisingly, the scrapped dataset is quite noisy. Firstly, the description is fundamentally different from search queries as they are written by the same person mostly at the time of writing the code itself. It uses the same vocabulary as the code itself. Secondly, the description might not closely resemble the gist of the function to its entirety or might be an outdated description for the function even though the code is scrapped from popular and well-known repositories. Finally, the quality of the code can be sub-par at times, which would mean that there can be a faster and better implementation elsewhere. It might also be full of bad coding practices or antipatterns.

III. MODEL

Each entry in the dataset has a function snippet along with its description in natural language. It tokenizes and creates word embeddings of both the function and its description and the model trains such that the document embeddings of both the code and description are as same as possible. Our model uses bidirectional LSTM's to capture the context of the function and description. The dissimilarity between those two document vectors is the loss, and the model fits to minimize the loss.

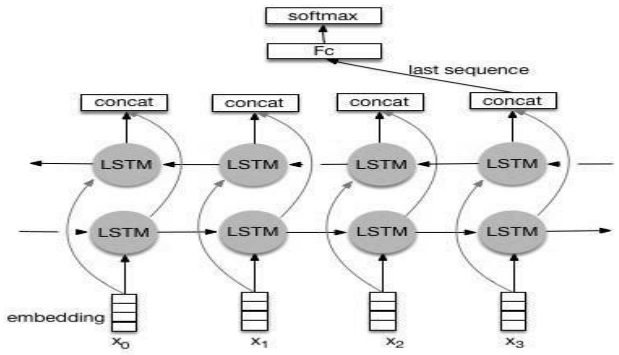

Fig. 1. Bidirectional LSTM Architecture A. Input

All the tokens in the function are given sequentially to the input as the sequence in which the tokens appear in the list has semantic information. This what helps differentiate between functions with similar tokens, for example, the function names "convertStringToInt" and "convertIntToString" although having exact same tokens have different semantic contexts.

B. Word Embeddings

As the source code is broken up into snippets with the granularity of a function and further tokenized in the process specified above each of the tokens of the function converted to word embeddings using a Word2Vec model, called fastText [11] which uses a continuous skip-gram model with a window size of 5, which means the words in a sentence within a distance of 5 words are considered neighbors. Generated word embeddings are of 500 dimensions, which captures the full intent of each vocabulary.

The word embeddings can be of higher dimensions, but embeddings of size 500 are optimized for both accuracy and speed.

C. Encoder

We used a bidirectional LSTM cell, specifically GRU cells to summarize the input sequence. Our model contains 3 hidden layers with 256 hidden units in each direction. The optimized used is Adam Optimizer, with a mini-batch size of 128. The model was trained for 100 epochs lasting over 12 hours on Nvidia RTX 2060, having compute capability of 7.5.

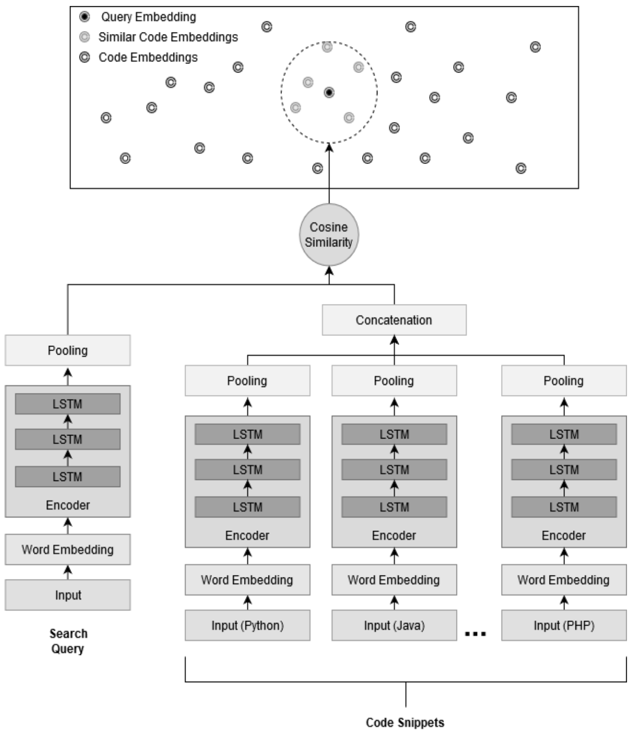

Fig. 2. Unified Neural Network Model for Document Embeddings D. Pooling

Max Pooling is used to combine the word embeddings into a sequence embedding to get a final document embedding to be mapped into the shared vector space for similarity search.

E. Similarity Search

When the user submits a query, the query is also converted to a document vector using the encoder, and then using Cosine Similarity k-nearest neighbors of the vector embedding is found for any particular programming language and returned to the user in descending order with code snippet with higher similarity values ranked higher.

$$\cos(c,d) = \frac{c^T d}{|c||d|}$$

We used FAISS [10], an open-source implementation of the similarity search which leverages GPU and is the fastest k-selection algorithm being 8.5x faster than any other implementation. FAISS helps in drastically reducing the training as well as search time for the queries as our dataset is enormous. higher similarity values ranked higher.

IV. RESULTS

A. Evaluation Metric

Mean Reciprocal Rank (MRR) is a rank-aware evaluation metric; it is a measure of where does the first relevant item lies in a ranked list. MRR is easy to compute and evaluate and puts a high focus on ranking the best result at the top of the list. It is a go-to metric to evaluate navigational or factual ranked lists. The metric, however, does has a downside; it only evaluates the first relevant recommendation in the list and does not consider other results. Mathematically, it is the sum of the multiplicative inverse of rank of the most relevant result of the i-th query.

$$\text{MRR} = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{\text{rank}_i}$$

Normalized Discounted Cumulative Gain (NDCG) is a qualitative ranking measure. Both MRR and NDCG values ranking relevant results top of the ranked list. However, NDGC quantifies the fact that some results are more important than others. NDCG can measure the usefulness and gain of the result accumulated over all the results with low ranked results having discounted gains. Simply put, highly relevant items must be ranked before medium relevant items and non-relevant items at last. It is often used to measure the effectiveness of web search engine ranking algorithms and such related query-based applications. To calculate NDCG, Discounted Cumulative Gain (DCG) is divided by the Ideal Discounted Cumulative Gain(IDCG) of the query. This is done to have an accurate evaluation of the ranking algorithm over a set of queries as this can't be consistently achieved using DCG alone, NDCG is used. The mathematical formula for this metric is as follows:

$$\text{DCG}_\text{p} = \sum_{i=1}^{p}\frac{2^{rel_i}-1}{\log_2(i+1)}$$

$$\text{IDCG}_\text{p} = \sum_{i=1}^{|REL_p|}\frac{2^{rel_i}-1}{\log_2(i+1)}$$

$$\text{nDCG}_\text{p} = \frac{DCG_p}{IDCG_p}$$

B. Evaluation Method

The evaluation method comprises of 100 search queries with human ranked relevant code snippets and NDCG scores are calculated using the predicted ranked snippets vs. the ideally ranked snippets. This method is not perfect and might not correlate to the actual task of code search, but this method has been widely used as a proxy for training and evaluating similar models.

Given below are the NDCG and MRR Results from the trained model on various languages and mean NDCG and MRR values of the model in its entirety.

Table I: Measure of Search Relevance using Normalized Discounted Cumulative Gain (NDCG) and Mean Reciprocal Rank (MRR)

Language	NDCG	MMR
Python	0.2578	0.6673
PHP	0.2925	0.5712
Java	0.3159	0.6082
Ruby	0.4612	0.7336
JavaScript	0.2235	0.4704
Go	0.2407	0.6985
Mean	**0.2925**	**0.6248**

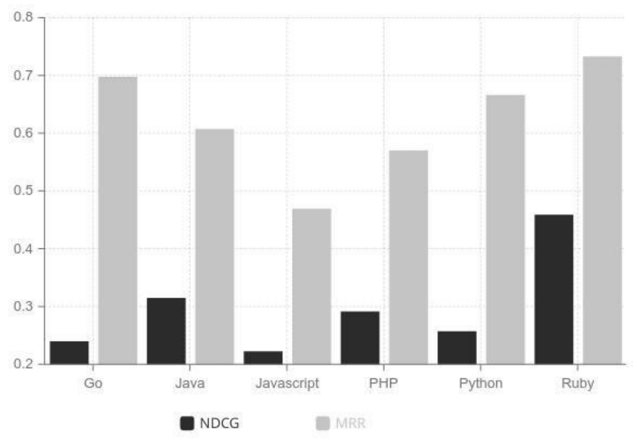

Fig. 3. NDCG vs MRR for all languages

V. THREATS TO VALIDITY

Even though our model is significantly better than statistical models that rely on textual similarity and many of the simplistic deep neural network embedding models; however, our model is far from perfect. We cannot stress this enough, but understanding the context and intent of any code snippet is challenging, and there exists no perfect way to extract this information from the source code. Our model doing these embeddings on multiple languages is the step in the right direction, but there exist some threats to the validity of our results, which we will discuss below.

Our model being a supervised learning model is hugely dependent on quality and quantity of data, and with noisy data or code with bad coding practices, the model will not be able to perform document embeddings with expected accuracy. In some cases, it was observed that the most relevant code snippet is ranked bellow some other non-relevant code snippets due to the code being semantically similar but not contextually similar.

For the queries to return relevant results, the intended code snippet must exist in our code corpus. If not, then all the queries will return is similar code snippets, which will be of no exact use to the user. So for the model to perform as expected, it must be trained on a huge amount of good quality data to achieve its core function.

VI. RELATED WORK

Deep Code Search [5] uses a neural network called CODEnn (Code Description Embedding Neural Network) to embed code snippets in high-dimensional vector space. DeepCS breaks the inputs the code snippet in three ways Method Name, Tokens, and API Sequences are used as input. DeepCS is only trained and evaluated on Java/Android codebase. It uses bi-directional Recurrent Neural Network (RNN) for document embeddings. The predictions are evaluated against Lucene [2] and CodeHow [9] predictions for the same dataset.

Neural Code Search [12] used Word2Vec to create word embeddings and using term frequency-inverse document frequency (TFIDF) method to calculate the weighted average of all the tokens in the code snippet to get document embedding and using these vectors to find similar code snippets to a query. Dataset [8] used is Android-related codes from GitHub and StackOverflow's answers to Android related questions for evaluation.

UNIF (Embedding Unification) [3] is a minimal extension to NCS, which provides an attention-based weighing scheme to improve the accuracy of unsupervised NCS method. Dataset used is a combination of CODEnn and NCS datasets. This research shows that even a simple solution can outperform complex models like CODEnn. Like the other related works, this also trains and evaluates only Java codebase.

VII. CONCLUSION

Our novel deep neural network model (UnifiedCodeNet) built using state of the art systems and techniques proves efficient in finding semantically similar code snippets from open-source code across six programming languages with a high level of accuracy and relevance. The effectiveness of the model across language differs understandably due to the difference in the quality of source-code and difference in coding practices. Python embedding model being the most efficient and Go embedding model being the least efficient model for the set of queries we used. This is not an exact indicator of the accuracy of the model as it is possible that there exists no exactly semantically similar code snippet for a particular set of the query in the evaluation set, which would skew the results significantly. Also, given a different set of evaluation queries, it is possible that the accuracy of models may vary depending on the queries. Our model is significantly better in ranking snippets in some areas; there exist areas of improvement like the model would perform significantly better on a uniform and larger dataset than the one we used. This dataset is huge; we were only able to train for 100 epoch; with more resources, the model is sure to perform significantly better.

In the future, we will try to investigate on tree-LSTM model for token embeddings to get a better semantic representation of the code snippets as well as the query. This method can be by far the best method for semantic search, but this will require further study.

REFERENCES

1. Miltiadis Allamanis. 2018. The Adverse Effects of Code Duplication in Machine Learning Models of Code. arXiv:1812.06469 [cs.SE]
2. Apache. 2020. Lucene. https://lucene.apache.org/
3. Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When Deep Learning Met Code Search. arXiv:1905.03813 [cs.SE]
4. GitHub. [n.d.]. https://github.com/
5. Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep Code Search. In Proceedings of the 40th International Conference on Software Engineering (Gothenburg, Sweden) (ICSE '18). Association for Computing Machinery, New York, NY, USA, 933–944. https://doi.org/10.1145/3180155.3180167
6. Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. ArXiv abs/1909.09436 (2019).
7. Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2017. Billion-scale similarity search with GPUs. arXiv:1702.08734 [cs.CV]
8. Hongyu Li, Seohyun Kim, and Satish Chandra. 2019. Neural Code Search Evaluation Dataset. arXiv:1908.09804 [cs.SE]
9. F. Lv, H. Zhang, J. Lou, S.Wang, D. Zhang, and J. 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). 260–270.
10. Facebook Research. 2020. FAISS. https://github.com/facebookresearch/faiss
11. Facebook Research. 2020. fastText. https://fasttext.cc/
12. Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on Source Code: A Neural Code Search. In Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (Philadelphia, PA, USA) (MAPL 2018). Association for Computing Machinery, New York, NY, USA, 31–41. https://doi.org/10.1145/3211346.3211353