Semantic Code Search and Retrieval System

Using Natural Language Queries

Jayakumar Rameshbabu
Dept. of Computer Science
University of Central Florida, USA
jayakumar.r@knights.ucf.edu

Sindhu Davuluri
Dept. of Computer Science
University of Central Florida, USA
sindhu_davuluri@knights.ucf.edu

ABSTRACT

Standard information retrieval methods do not work effectively in the code search domain as search engines for code never fully understand what we want, unlike traditional web search engines, because there is often little shared vocabulary between search terms and language in the code. Hence, prompting the need for a semantic code search and retrieval system.

Semantic Code search is the task of retrieving relevant code given a natural language query. While related to other information retrieval tasks, it requires bridging the gap between the language used in the code (often abbreviated and highly technical) and natural language more suitable to describe vague concepts and ideas.

To contribute to the progress of semantic code search, we are attempting the CodeSearchNet challenge using their publicly available dataset, the CodeSearchNet Corpus. The challenge consists of 99 natural language queries with about 4k expert relevance annotations of likely results from CodeSearchNet Corpus labeled by a human expert.

In our paper, we describe the methodology we implemented to locate and retrieve the relevant code snippets, along with their dependent code blocks using RoBERTa model based on semantic similarity between the user input text and the high-level comment of the function methods.

KEYWORDS

Semantic code search, neural networks, joint embedding, Transformers, BERT, RoBERTa, Masked Language Modeling, Next word prediction, Pytorch, Hugging face

1 INTRODUCTION

The deep learning revolution has fundamentally changed how we approach perceptive tasks such as image and speech recognition and has shown substantial successes in working with natural language data. These have been driven by the co-evolution of large (labelled) datasets, substantial computational capacity, and a number of advances in machine learning models. However, deep learning models still struggle on highly structured data. One example is semantic code search: while search on natural language documents and even images has made great progress, searching code is often still unsatisfying. Standard information retrieval methods do not work well in the code search domain, as there is often little shared vocabulary between search terms and results. Search engines for code are often frustrating and never fully understand what we want, unlike traditional web search engines.

Moreover, a code retrieval system offers major applications such as code reusability. Repurposing existing code optimizes any software application development process by increasing productivity, reducing development time, and cutting costs. Reusing code that has been tried and tested is highly reliable as it is most likely to run smoothly and be perfectly functional.

Also, as the scale of any software application development project increases, it becomes increasingly difficult to communicate effectively with all the developers involved and implement code reuse by manually brainstorming to find areas of the project where code can be reused. Hence having a system that automatically fetches the blocks of relevant code snippets that can be reused is effective. Further, it would expedite the process of on-boarding new software engineers onto projects and bolster the discoverability of code in general.

One of the other advantageous applications of having a code search and retrieval system is the security aspect that reusing internal code provides. It is generally safer to use internal code by organizations in comparison to code from third-party resources available publicly.

All in all, having a system in place that performs a semantic code search and retrieves the relevant blocks of code is effective and highly essential.

2 PROBLEM STATEMENT

Searching code on GitHub, for example, is currently limited to keyword search. This assumes either the user knows the syntax or can anticipate what keywords might be in comments surrounding the code they are looking for. Our motivation has been to

implement and leverage deep learning methods to enable the semantic search of code.

2.1 Initial Data

For the purpose of our project, we have collected over 2000 well-documented open-source code files from non-forked GitHub repositories. After data collection, we had to preprocess our files to identify the functional units in the code, for which we have used Abstract Syntax Tree package in case of Python files. Likewise, code files in other computer languages can be preprocessed accordingly with the help of corresponding relevant packages and libraries.

The pre-processing included splitting the entire code file into multiple blocks of code, mapping high-level function or method comments to the corresponding code blocks, and locating and mapping dependent blocks of code, which was particularly very hard and challenging to implement.

2.2 Objective

Our main goal in our project is to perform semantic code search and retrieval by comparing natural language user queries with the comments associated with the code blocks in the repository files.

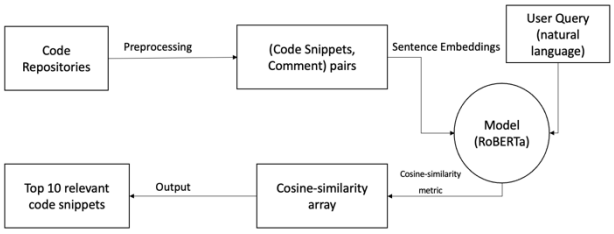


Figure 1: Overview of our system

After collecting over 2000 open-source code files from non-forked GitHub repositories and pre-processing them, we get our Code-Comment pairs, which are the input to our model.

After the pre-processing step, we are finding the sentence embeddings for all the comments that are mapped to the functions and the also to the input text or query by the user. We have computed sentence embeddings for these function comments in advance for optimized performance of our system.

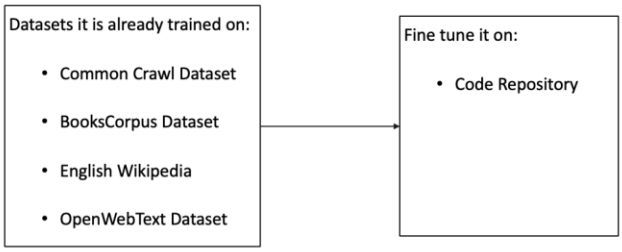
To find sentence embedding for the function comments and user queries, we are implementing Word2vec and also a pre-trained BERT model from hugging face library and fine-tuning it to our data.

Our model outputs a cosine-similarity array, allowing to find the top 10 most relevant code snippets corresponding to the user query.

2.3 Hypothesis Testing:

Can we fine tune the base pretrained-model such as Roberta-base or STSB-Roberta-based on downstream tasks like Mask language modeling or next word prediction, so that the model gets accustomed to the corpus and improves the accuracy on semantic similarity.

RoBERTa: A Robustly Optimized BERT Pretraining Approach. It is built on top of BERT Transformer trained on very huge dataset. It is trained on MLM task and has lots of features. Similarly, the other variant STSB-Roberta(semantic textual similarity benchmark) is trained on Sentence Similarity task which compares two sentences and shows whether the sentences are semantically similar or not. Within these models there are multiple variants based on their memory size and dataset size it is trained on. We will be taking the base version of these two models due to hardware constraints and both these models will again be fine-tuned on MLM and Next word-Prediction Task.



2.4 Constraints and Challenges

While we can implement any pre-trained model on raw code files obtained from GitHub repositories etc., evaluation for this semantic code search task is extremely hard, as there is no ground truth to compare our results to.

Alternative is to manually annotate datasets large enough for training high-capacity models, which is cumbersome and impractical.

Hence, we are instead making use of the benchmark datasets from the CodeSearchNet corpus.

2.5 Benchmark Datasets

CodeSearchNet corpus is a collection of datasets and benchmarks that explore the problem of code retrieval using natural language.

The corpus contains about 6 million functions from open-source code obtained from non-forked GitHub repositories, spanning six programming languages: Go, Java, JavaScript, PHP, Python, and Ruby. The Corpus also contains automatically generated query-like natural language for 2 million functions, making it large enough to enable training of high-capacity deep neural models on the semantic code search and retrieval task. These 2 million datapoints (Code-Comment pairs) consist of the Comment: a top-level function or method comment (e.g., docstrings in python), and Code: an entire function or method.

Since the scale of this benchmark dataset is enormously huge, we are only considering code files in python, which still turns out to be very huge for the purposes of our hyperparameter tuning, so we are further reducing down to considering only 15% of our

available Code-Comment pairs in python. Data has been split into train, validation, and test sets in a ratio of 90, 5 and 5 percent corresponding to 61,826, 3,466 and 3,326 samples respectively, such that code from the same repository can only exist in one partition.

3 RELATED WORK

CodeSearchNet Challenge, [1] a collaboration between GitHub and Microsoft research was hosted for researchers to develop models that can improve semantic code search and effectively retrieve relevant code snippets given natural language queries, with a focus on models that can generalize to new programming languages and new query types. They also created and released a CodeSearchNet corpus that was highly useful for us in our project and also to other researchers who only had access to smaller datasets due to the need of manual annotation of code files for semantic search and retrieval.

One of the key areas of machine learning research underway at GitHub [5] is representation learning of entities, such as repos, code, issues, profiles, and users. And they have made significant progress towards enabling semantic search by learning representations of code that share a common vector space as text where (text, code) pairs that describe the same concept are close neighbors, whereas unrelated (text, code) pairs are further apart. By representing text and code in the same vector space, they can vectorize a user's search query and lookup the nearest neighbor that represents code. Further, they are also working on determining the best way to augment existing keyword search with semantic results and a way to incorporate additional information such as context and relevance.

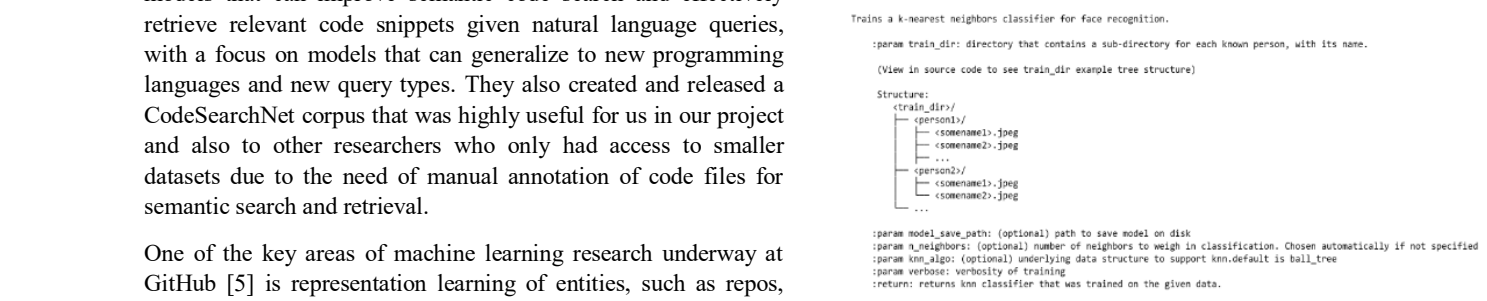
Meta AI has developed a code search tool, called Neural Code Search (NCS), [6] an unsupervised model that applies natural language processing and information retrieval (IR) techniques directly to source code text, by accepting natural language queries and returning relevant code fragments retrieved directly from the code corpus. They have also developed UNIF, an extension of NCS that uses a supervised neural network model to improve performance when good supervision data is available for training. Leveraging other open source Facebook AI tools such as fastText, FAISS, and PyTorch, NCS and UNIF both represent natural language queries and code snippets as vectors, and then train a network such that the vector representations of semantically similar code snippets and queries are close together in the vector space. To evaluate NCS and UNIF, they have created a dataset of the public queries on Stack Overflow. Their NCS model captures program semantics, that is, the intent of the code snippet by using embeddings as continuous vector representations that, when computed appropriately, have the desirable property of putting semantically similar entities close to each other in the vector space. In their NCS approach, each code fragment during model generation is embedded in a vector space at a method-level granularity. Once the model is built, a given query is mapped to the same vector space, and vector distance is used to estimate the relevance of code fragments to the query.

4 TECHNIQUE

4.1 Exploratory Data Analysis

As part of our Exploratory Data Analysis , we searched for any anomality in the datasets.

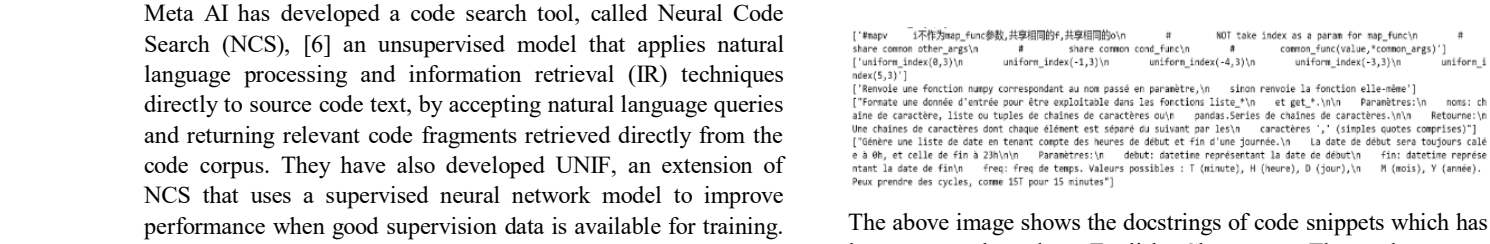
A. Unwanted Data



The Above image is docstring of one document, the first line is enough describing the functionality of the code snippet, the other data's describes parameters and their functionality which is not required for semantic similarity, which we can remove it. So, first line/para ending in newline character (\n) is enough.

B. Other language sentences

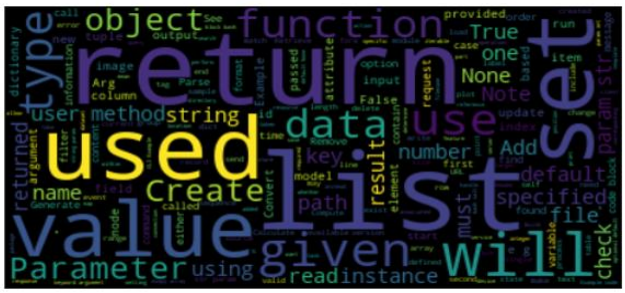
The Dataset contained very minimal amount of data of other languages, but we are only considering the English language Text. Hence, we are removing the documents with other languages.



The above image shows the docstrings of code snippets which has languages other than English Characters. These documents removed by a simple check, where if any characters have ascii value of more than 127, it is discarded. Simple and effective, but this increases the processing time since it takes char by char comparison.

C. Frequent Words in our docstring

To check out the frequent words in our dataset. We used word cloud library to build a word cloud.



In the above image we can see the word cloud and those are seeming to be common in docstring and didn't find any abnormality.

D. Other Details and documents Info

```
Total number of sentences or docs 457461
least no of words in a sentence 1
Sentence with word count <= 2 593
No of Other language sentences  3937
No of Sentences with empty string  0
No of sentences which cannot be used because of above conditions =  4465
no of usable sentences =  452996
```

The Above data shows the details of the entire corpus on which the search will be performed, not the training. From the above dataset, will split it into train, validate and test , then 15% of the dataset will be trained on MLM and next word prediction tasks.

4.2 Preprocessing Code Repository

The Above data shows the details of the entire corpus on which the search will be performed, not the training. From the above dataset, will split it into train, validate and test , then 15% of the dataset will be trained on MLM and next word prediction tasks. The Code repository on which the solution is going to work on top needs to be processed. The preprocessing includes checking the code scripts are:

1) Syntactically Correct
2) Segmenting the code script into multiple code snippet
3) Finding the comments within the code snippet and mapping to it.
4) Discarding the comments which are in another language and discarding the data from second line.
5) Removing the unwanted characters but not the stop words.
6) Finding the dependent module within the code scripts and mapping them.

To do all these tasks, we are using a package called AST (Abstract Syntax Tree) for python codes. This Implementation is performed for python scripts but can be extended to any other languages.

Sample Code from a repository:

```python
import math

def func1(a,b):
    # this function just adds two numbers and performs math ceil and floor
    temp = a+b
    print(math.ceil(temp))
    print(math.floor(temp))
    return temp

#this logic checks which number is smallest
a=10
b=20
if b<a:
    print("b is smaller")
else:
    print("a is smaller")

def find_A_lessthan_B(a,b):
    # This function checks if the given input a is less than b or not
    if a<b:
        return True
    else:
        return False

#to check the smallest of all three numbers
c=100
if find_A_lessthan_B(a,b):
    #we have to check smallest of other two numbers
    if find_A_lessthan_B(a,c):
        print("a is smaller")
    else:
        print("c is smaller")
elif find_A_lessthan_B(b<c):
    print("b is smaller")

else:
    print("c is smaller")
```

Sample preprocessed output with comment and dependent code mapped to a particular code snippet:

.\Repo\sample code.py	def func1(a,b):
	# this function just adds two numbers and performs math ceil and floor
	temp = a+b
	print(math.ceil(temp))
	print(math.floor(temp))
	return temp
		import math	this function just adds two numbers and performs math ceil and floor

.\Repo\sample code.py	def find_A_lessthan_B(a,b):
	# This function checks if the given input a is less than b or not
	if a<b:
	return True
	else:
	return False
			This function checks if the given input a is less than b or not

.\Repo\sample code.py	def Function_Name(a, b):
	
	#to check the smallest of all three numbers
	c=100
	if find_A_lessthan_B(a,b):
	#we have to check smallest of other two numbers
	if find_A_lessthan_B(a,c):
	print("a is smaller")
	else:
	print("c is smaller")
	elif find_A_lessthan_B(b<c):
	print("b is smaller")
	else:
	print("c is smaller")
		def find_A_lessthan_B(a, b):
		if (a < b):
		return True
		else:
		return False
			to check the smallest of all three numbers$$@$$we have to check smallest of other two numbers

Once We trained our model and as a preprocessing step, we can compute all the embedding values of the comments in the code repository and store it. So that we don't have to compute it again and again for each and every query. This is essential, especially when we consider for very big repository, otherwise it will take hours to perform a single query search.

4.3 Finding the Similarity

Basic idea is the sentences which are similar are closer to each other in high dimensional vector space and are farther, when they are dissimilar, semantically. Word embedding for the sentences are generated from any pretrained model. We can use any

similarity metric like, Jaccard similarity, Cosine similarity, Euclidean Distance to find the similarity, but for our evaluation we are using Cosine similarity, since it computes the context between two sentences very well.

$$\text{cosine similarity} = S_C(A,B) := \cos(\theta) = \frac{\mathbf{A}\cdot\mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}},$$

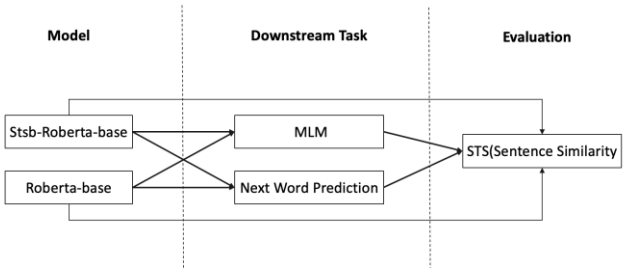
4.4 Generation of the word Embedding

Word Embedding for a sentence can be generated from any pretrained model like word2Vec, Glove or any model trained on Semantic textual Similarity. The model we are using the solution is STSB-Roberta-base and Roberta-base, which is a pretrained model trained on large sentence similarity dataset. To use these pretrained model, we are using Transformers package which helps us to load these models and its tokenizer from Hugging Face repository. With few lines of code in python, we can easily to do all these.

4.5 Fine Tuning the model

Both the models Roberta-base and STSB-Roberta base will be fine-tuned on two downstream tasks separately, which are:

1) next word prediction
2) masked language modeling tasks.



15% of our dataset will be used to train on these downstream tasks for few epochs, so that the model won't move towards these tasks completely.

This fine tuning is performed to make the model accustomed to the repository and to improve the accuracy on sentence similarity tasks which will be used in the final solution.

5 EVALUATION

We will evaluate our finetuned model on the dataset which are manually generated. This dataset is found in the CodeSearchNet Challenge.

The format of the Dataset:

Language	Query	URL(maps to original Comment)	Relevance
Python	Function to add two numbers	Github.com/basicfunction /addtwonumbers	3

Cosine similarity between the query and the comment present in the code of the link is computed, then this score will be compared with relevance score. Since cosine Similarity is continuous and relevance score is discrete, will map the relevance score to the cosine similarity bucket.

Relevance Score Range 0-3, where

0-More irrelevant

3- Most relevant

Relevance vs Cosine Similarity:

0 – 0-25

1 – 25-50

2 – 50-75

3 – 75-100

If the Cosine Similarity of the score falls in the respective bucket, will increase the score by 1. If not, will increase the score based on how close the cosine value to the mid of the bucket.

Formula we used:

If Score value lies in respective relevance Bucket:

Score+=1

Else:

Score += (100 – abs(cosine value – mid(relevance bucket))/100

Once the score is calculated for all the documents, we can find the accuracy.

Accuracy = Score/Total no of documents.

5.1 Results

These are the below result showing our evaluation performed on our base models and fine-tuned models on the manually annotated datasets.

Roberta-base fine tuned	
Task	Accuracy
MLM	57.22
Next Word	74.26

Stsb-Roberta-base fine tuned	
Task	Accuracy
MLM	56.84
Next Word	66.41

Stsb-Roberta-base fine tuned	
Task	Accuracy
MLM	56.84
Next Word	66.41

6 DISCUSSION

6.1 Sample Result

As we can see from our Evaluation Results the base line models are performing very well in semantic similarity. Fine tuning on MLM tasks makes the model degrade on the semantic textual similarity, whereas there is a slight improvement when it trains on Next word prediction compared to the base-line model. With Further fine tuning and exploration, we can improve the accuracy further.

Sample Result in Real Time:

Sample query = "give a function to sort the array"

Top 20 Results with Similarity Score:

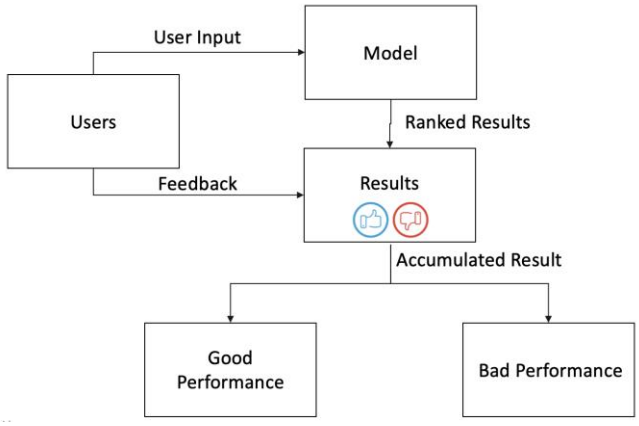
```
1. Sort array by a column.: [0.8046305179595947]
2. Sort the arrays.: [0.7864565253257751]
3. Return the indices that would sort this array.: [0.7787270545959473]
4. Sort an array, in-place.: [0.7656118273735046]
5. Sort an array along its rows or columns.: [0.7511001229286194]
6. Sort all values in this SArray.: [0.7223422527313232]
7. Returns the indices that would sort the array.: [0.7222514152526855]
8. Returns the indices that would sort an array.: [0.7147722840309143]
9. Collection function: sorts the input array in ascending
 or descending order according: [0.6849580407142639]
10. Sort key function.: [0.663195729255676 3]
11. Sort: [0.6576447486877441]
12. Sort by a given variable.: [0.6544890403747559]
13. This will insert the value into the array, keeping it sorted,
 and returning the: [0.6502787470817566]
14. Create a sort key (when used in sort_by) by the passed array
 expression or: [0.6494648456573486]
15. Sort an integer array using the Shell Sort
 algorithm.: [0.6435363292694092]
16. Compute the product set of array_a and array_b and
sort it.: [0.6365556716918945]
17. Return function for sorting.: [0.6317663788795471]
18. Sorts array "a" by columns i: [0.6309460997581482]
19. Sort by the values.: [0.6285112500190735]
20. Used by sort(): [0.6111859679222107]
```

This is the Sample result from the fine-tuned model on a very random query. It gives a very good similar scores for the similar sentences.

6.2 Deployment and monitoring

Once Evaluation is Done, we can deploy this model in any of the repository and ready to use. It is necessary to check whether our model performs well in real time as well.

The below diagram shows the flow how we can use the User feedback during the real time execution to check and evaluate our model.



The feedback not only helps us in monitoring the performance but also it will create a labelled dataset. This is a very good strategy to further fine tune our model when performance degrades and also it shows other different analysis like what kind of functionalities are repeatedly used by the developers and which are not available, anyone can quickly create the function share it among others. This way it increases the productivity of the developers.

7 CONCLUSION

From the above result we can see that our hypothesis is true, and it performed well. The Baseline model can be finetuned on any code repository on Next word prediction task for a few epochs and can be used for semantic search of code snippet efficiently. With our preprocessing strategy and computing sentence similarity, we can efficiently search for any code snippet and improve the productivity of developers.

ACKNOWLEDGMENTS

We would like to thank the authors of CodeSearchNet for making their system and data public. Similarly, we thank the authors Hamel Husain and Ho-Hsiang Wu for making their GitHub blog post available. We thank Fereshteh Lux for her feedback on an earlier draft of this paper.

CONTRIBUTION : Sindhu has done the data collection, model fine-tuning and evaluation components. Jayakumar has done the pre-processing and exploratory data analysis. And report work has been done equally.

REFERENCES

[1] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis and Marc Brockschmidt. 2020. CodeSearchNetChallenge: EvaluatingtheStateofSemanticCodeSearch. arXiv:1909.09436, 6 pages. DOI:https://doi.org/10.48550/arXiv.1909.09436.

[2] Dawn Drain, Changran Hu, Chen Wu, Mikhail Breslav and Neel Sundaresan. 2021.GeneratingCodewiththeHelpofRetrievedTemplateFunctionsandStackOverflowAnswers. arXiv:2104.05310, 8 pages. DOI:https://doi.org/10.48550/arXiv.2104.05310.

[3] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen and Satish Chandra. 2019. WhenDeepLearningMetCodeSearch. arXivpreprintarXiv:1905.03813, 10 pages. DOI:https://doi.org/10.48550/arXiv.1905.03813.

[4] Geert Heyman and Tom Van Cutsem. 2020. Neural Code Search Revisited: Enhancing Code Snippet Retrieval through Natural Language Intent. arXiv:2008.12193, 18 pages. DOI:https://doi.org/10.48550/arXiv.2008.12193.

[5] https://github.blog/2018-09-18-towards-natural-language-semantic-code-search/

[6] https://ai.facebook.com/blog/neural-code-search-ml-based-code-search-using-natural-language-queries/

[7] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, Veselin Stoyanov. 2019. RoBERTa:ARobustlyOptimizedBERTPretrainingApproach. arXiv:1907.11692, 10 pages. DOI:https://doi.org/10.48550/arXiv.1907.11692.