Journal of Science and Engineering Research
Vol. 2, No. 1, 2023, pp.17-28
eISSN: 2957-7187

Semantic Code Search using Hugging Face Transformer

Harshil Gandhi[1] and Jinan Fiaidhi[2*]

[1,2]Department of Computer Science, Lakehead University, Canada*
[2]jfiaidhi@lakeheadu.ca*

DOI: http://dx.doi.org/10.56828/jser.2023.2.1.3
***Article history**: Received (December 2, 2022); Review Result (February 1, 2023); Accepted (April 3, 2023)*

Abstract: Semantic code search is a well-defined task to retrieve relevant code snippets for the inserted language query. The semantic code search is an information retrieval task designed to help software engineers reuse the appropriate Code instead of writing the same Code repeatedly. This task closes the gap between the language used in code development and the language used in queries. Our approach here defines by creating a unique dataset in a python programming language, pre-processing the dataset, and training the machine learning model to get the result. A pertained state-of-the-art machine learning model from the hugging face library has been used to answer a search query. A Sequence-Sequence encoder using an attention mechanism also trains the dataset and produces the desired output. With the help of parsing and natural language processing techniques, we can create a semantic code search engine for Python datasets.

Keywords: Face transformer, Semantic code, Dataset, Artificial intelligence, Machine learning

1. Introduction

Code searching is one of the frequent tasks in software engineering. Software engineers often look for the piece of Code on the internet which complies with the project one is working on. To implement certain functionality, i.e., "sort the array," the developer uses the relevant Code queried on the larger codebase. Machine learning assembled with natural language processing and modern, powerful training tool revolutionized the training of machine learning models for trivial tasks. With this advancement, the training of a large corpus made the training easy and curbed down the computational time. Most of the previous research on information retrieval was done under keyword-based search. However, from the BERT algorithm [1] used by Google for information retrieval, AI and machine learning were introduce for information retrieval tasks [2].

There are mainly two types of search techniques. First, Keyword bases search, a conventional search approach [2], was used at the early stage of the search engine revolution. In this approach, the search engine looks for the exact words in the query, and as a result, it gives only those returns in which the keyword has appeared. This technique leads to a result with the same value or word. There is no middle ground; there is no understanding of the context of a query. To resolve this problem, semantic code search was introduced. The semantic search understands the inserted query's context and what is being searched and gives

desired and relevant results. The semantic search techniques use ontology [3], while keyword-based search uses a page ranking algorithm to find results. The semantic search not only focuses on the keyword but also on other relationships between different types of resources. One of the most powerful advantages of sematic-based search is that it can solve complex queries.

In this research paper, we have introduced new machine-learning techniques with the help of a state-of-the-art natural language processing model generated by a semantic information retrieval engine. The main contribution of this paper is as follows:

- Generate and preprocess dataset for Semantic code search model
- Train the dataset using the Hugging face Albert transformer model
- Train the dataset using an encoder-decoder with a multi-head attention mechanism
- using k nearest neighbor, find the top 5 relevant results using cosine similarities.

Following is the flow chart of the system implemented in this paper.

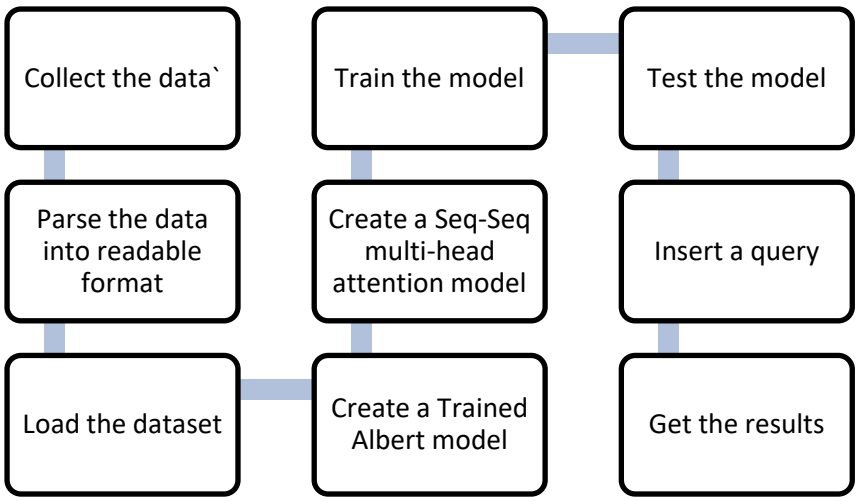

Figure 1: Flow chart of paper

2. Related Research Work

Code search is one of the recent areas of research. Following are some of the research works which has been in this field over the last few years.

Semantic Code Search using Code2Vec: A Bag-of-paths model

Lakshmanan Arumugam [3] developed a semantic code search using the Code2Vec bag-of-path model. He followed the neural model because it showed semantic meaning and could represent natural language using the vector used in various NLP tasks. In particular, the author evaluated the performance of a semantic search task for code snippets using Code2Vec, a model for learning a distributed representation of source code called code embedding. The main idea behind the use of Code2Vec is that the source code is structurally different from natural language [4], and models that use the syntactic properties of the source code help learn semantic properties. The author combined Code2Vec with other neural models representing natural language through vectors to create a better hybrid model than previous benchmark-based models. The author also examined the impact of various metadata

on the retrieved code snippets in terms of relevance. The model was evaluated using the BLEU algorithm [5]. BLEU is an algorithm for assessing the quality of text machine-translated from one natural language to another.

Code snippets are encoded by the migrated and implemented Code2Vec model, and the documentation for each code snippet is tokenized and encoded using one of the benchmark baseline query encoders. Query encoding uses an NBoW (Natural Bag of Words) technique that transforms each query token into a learnable embedding called a vector representation [6]. The author created the top 10 results using two data points, predictions calculated during the search, and a reranking algorithm based on repository metadata. The reciprocal rank (RR) is an information retrieval measure that calculates the reciprocal of the rank from which the first relevant document was obtained.

CodeSearchNet Challenge: -Evaluating the State of Semantic Code Search

Hamel Husan and Mitiadis Allamanis presented a code search net challenge in the poll above. They programmatically defined a custom corpus created by scraping open-source repositories and combining their functionality with documents treated as natural language annotations. In addition, we used a deep learning model to train the data for this task. In this study, the author created a dataset consisting of millions of functions that map to six programming languages and their corresponding programming languages. The dataset was collected from GitHub, and the Code was parsed using Treesitter, GitHub's universal parser [7].

In developing the Code, search engines used simultaneous embedding of code and search queries. The architecture of the author's model uses one encoder per input language and trains them to map the inputs to a single common vector space. The author's goal is to map the Code to the appropriate language. The authors have implemented a search method by embedding a query and returning the Code for a snippet close to vector space relative to the vector. To learn these embedded functions, the author combines a standard sequel model. First, the Code is preprocessed and converted to sub-tokens; then, the natural language tokens are split into byte pair encodings. In addition to their research, it included Elasticsearch [8], a widely used search engine. The author trains the model by keyword matching with a set of word models.

Deep Code Search

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim have developed deep code search engines that use deep learning and neural network models and NLP preprocessing techniques for deep code search. This model is trained with over 18.2 million Java code snippets and the CODEnn model and is evaluated on Stack Overflow for 50 simple queries [9].

Here we introduce a recurrent neural network for embedding sequential data. An RNN is a class of neural networks in which the hidden layers of a model are continuously used for computation. Use this to record dynamic movements over time. CODEnn's neural network architecture consists of three modules. A code embedding network for source code and a descriptive embedding network for embedding natural language descriptions in vectors. This similarity engine measures the degree of similarity between Code and description. In this study, the entire dataset corpus consists of only JAVA code snippets, so we will perform preprocessing to extract the training corpus method names, tokens, and API sequences. When searching for code snippets, the code vector for each code snippet is calculated and returned first. This high-level code snippet shows the high-level K vector closest to the query vector. The author uses frankness, success rate, accuracy, and medium-sized mutual rank for model evaluation. The author mainly uses the first hit of the flank and the result list. Users must scan the results from top to bottom. Smaller frank means lower inspection efforts to find desirable

results. The author uses Frank to evaluate the validity of a single code search query. The results are displayed by comparing DeepCodeSearch with other algorithms such as "Lucene" and "Codehow." [10][11] This shows that DeepCS generally produces something more relevant result.

3. Dataset Collection and Preprocessing

To get the data required for our semantic code search engine, we have created our own 32 Python files: consisting of different functions, function descriptions, multiple functions in the same file, and classes in a Python file. These Python files will be helpful to create a general-purpose search engine. Figure 2(a) shows an example of the function of a Python file. According to Python docs, a Python function is consisting of a decorator, docstring, function signature, and function definition.

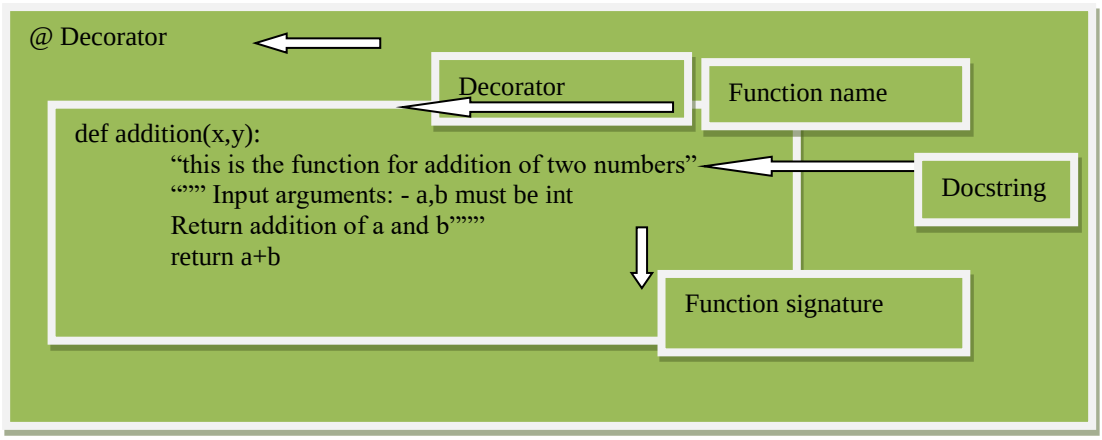

Figure 2(a): Ideal function structure

```
Function name: - addition(x,y)
Function description: -this is the function for addition of two numbers
Function Code: - return a+b
```

Figure 2(b): Description of extracted data

Consider Figure 2(a), in which a Python function is defined. Using the parsing method and function name, function docstring and function code have been extracted, shown in Figure 2(b). Function details will use further in this research.

After collecting the desired information of a Python file using parsing, we need to pre-process the data, so the dataset will be easy to learn for the machine learning model. The following are pre-processing methods implemented on the data.

Create Function-docstring Pairs

After gathering the dataset our next steps will be to create a function-docstring pair of each python file. Firstly, the Python file is compiled using the "trackback" [12] library in Python to check if there is any compilation error in the file; if there is an error, then the function or a Python file will be excluded from the dataset. After compilation, we need to extract the

function name and docstring from the given Python file. We will be using the AST [13] module in Python, which converts the Python code into an abstract syntax tree for analysis. So here we are, keen on extracting the function name and its corresponding docstring signature. So, with the help of “trackback” [12] and “nltk RegexpTokenizer" [24], we extract both things and tokenize them. One reason behind tokenization is that it removes punctuation and decorators and converts the whole text into lowercase. Figure 3 shows the example of the function-docstring pair.

Remove the Duplicate

All duplicate entries related to the function definition or function token are removed. This is done to prevent weights/biases from being added to string pairs in certain function documents during training. This is because more entries on specific data can affect your training. In this study, we have a relatively small dataset. This feature is included for large datasets.

Remove the Function without Docstring

In this study, we perform supervised machine learning. Therefore, a well-structured and defined dataset is required for more accurate results. As with this study, there are two critical things. The Code and the corresponding docstring. However, some functions do not have a docstring in the real world, are too short, or are poorly described. Here, some of these cases are excluded from the dataset.

Split the Dataset

Splitting the dataset is one of the most important in the field of machine learning. In this study, we divided data into three things, training dataset, validation dataset, and testing dataset. The ratio of the splitting dataset is 80:10:10 for the train:validation:test [14].

	Source_Code	repo_path	Source_code	pairs
0	NaN	/content/gdrive/MyDrive/Dataset/1.py	# -*- coding: utf-8 -*-\n"""\nCreated on Wed F...	[(FindFact, 8, def FindFact(self, n):\n """...
1	NaN	/content/gdrive/MyDrive/Dataset/2.py	# -*- coding: utf-8 -*-\n"""\nCreated on Thu F...	[(decimal_to_binary, 9, def decimal_to_binary(...
2	NaN	/content/gdrive/MyDrive/Dataset/4.py	# -*- coding: utf-8 -*-\n"""\nCreated on Thu F...	[(__init__, 17, def __init__(self, k: float, w...
3	NaN	/content/gdrive/MyDrive/Dataset/5.py	# -*- coding: utf-8 -*-\n"""\nCreated on Thu F...	[(maxpooling, 12, def maxpooling(arr: np.ndarr...
4	NaN	/content/gdrive/MyDrive/Dataset/6.py	# -*- coding: utf-8 -*-\n"""\nCreated on Thu F...	[(binary_search, 11, def binary_search(a_list:...

Figure 3: Dataframe of the dataset with source code and function-docstring pairs

	repo_path	_	pair	function_name	lineno
0	/content/gdrive/MyDrive/Dataset/1.py	0	(FindFact, 8, def FindFact(self, n):\n """\...	FindFact	8
1	/content/gdrive/MyDrive/Dataset/2.py	0	(decimal_to_binary, 9, def decimal_to_binary(n...	decimal_to_binary	9
2	/content/gdrive/MyDrive/Dataset/4.py	0	(__init__, 17, def __init__(self, k: float, wi...	__init__	17
3	/content/gdrive/MyDrive/Dataset/4.py	1	(__str__, 30, def __str__(self) ->str:\n re...	__str__	30
4	/content/gdrive/MyDrive/Dataset/4.py	2	(detect, 34, def detect(self, img_path: str) -...	detect	34

Figure 4(a): Dataset split into function name, pair, and line of number

original_function	function_tokens	docstring_tokens
def FindFact(self, n):\n """\n funct...	findfact self for in range 1 1 if 0 print end	function to find a factorial of a n number
def decimal_to_binary(no_of_variable: int, min...	decimal to binary no of variable int minterms ...	function to convert a decimal to binary
def __init__(self, k: float, window_size: int)...	init self float window size int if in 0 04 0 0...	k
def __str__(self) ->str:\n return f'Harris ...	str self str return harris corner detection wi...	
def detect(self, img_path: str) ->tuple[cv2.Ma...	detect self img path str tuple mat list list i...	returns the image with corners identified img ...

Figure 4(b): Dataset split into function tokens and docstring tokens

4. Convert Docstring to Embedding using a Transformer

After collecting, preprocessing, and splitting the dataset, the next part will help you understand one of the presented data docstrings into a vector and gain great insight into the data. This search uses the Albert transformer [15] provided by the hugging face [16] library to convert the document string to a vector. This section explains why you should choose Albert, how to use this dataset to train transformers, and finally how to convert docstring to vector.

Albert stands for "A Lite Bidirectional Encoder Representation." Albert is one of the transformers of a BERT family introduced by Google. Albert is an upgrade from BERT, which advances the state-of-the-art performance in 12 over NLP tasks, and some of them are Standford Question Answering Dataset. The Albert is implemented on top of TensorFlow, combined with a pre-trained Albert model. Choosing Albert over other transformers is parameter sharing, smaller word embeddings, and sentence over prediction. Here we will be using Albert's pretraining model, which can be found on the TensorFlow hub. The reason behind choosing a pre-trained model is that it can save us time and resources for building a machine-learning model from scratch. Here for training, our dataset is docstring which is nothing but an English representation of how the Albert transformer is trained.

Albert has an encoder number similar to the BERT transformer, but Albert shares weights between encoders and treats Albert as a single encoder with multiple embeddings applied. This significantly improves training speed and reduces specific parameters across Albert's models. Also, since only one set of weights is stored, the model takes up much less space on GPU memory. The parameter-sharing technique also serves as a form of regularization that stabilizes training and aids in generalization. The author of Albert suggests several ways to share parameters, such as ALBERT's default decision is to share all parameters between layers.

Another advantage of Albert is the small embedding, with only 1/6 of the BERT embedding with 128 features. This is one of the reasons why the Albert Transformer has a small number of parameters. This will speed up model training, lose weight and help the model learn a better display. Another advantage of the transformers, as mentioned above, is that they can predict the order of sentences. Albert is trained in a sentence ordering task that defines whether the two sentences are coherent, that is, what comes before and after.

Training of Albert Transformer

Now that we have a pre-trained Albert Transformer model, we need to tune the model to work with a particular dataset. First, the pre-trained model needs to be refined because it is trained on many English word datasets. Perform incremental training on a pre-trained Albert model using the docstrings collected from the dataset for fine-tuning. Here we need to generate and create escaped sentence pairs so that the model uses n-gram escaped language

modeling. We could use the same word file author used to train the Albert model. However, some docstring programming contexts have different meanings than English words. Therefore, we will use the Docstring dataset and the Albert model to create pre-training data for Albert's model. This dataset will be further used for incremental training [17] to fine-tune the Albert model. After training the entire model, the Docstring is transformed into 768 dimensions of the vector for further processing. This is how the Docstring is converted to a vector.

5. Function Token to Embedding Vector

The previous section will convert the Docstring to a 768-dimensional vector. Therefore, this section transforms the function token into the same 768-dimensional vector so that both the function token and the corresponding document string are in the shared vector space. If those values are very similar to the vector generated for the corresponding document string, then the function vector in the shared vector space is called. This helps in interpreting the code and generating the code vector. Figure 5 shows the architecture of the transformer model used for the translation [18].

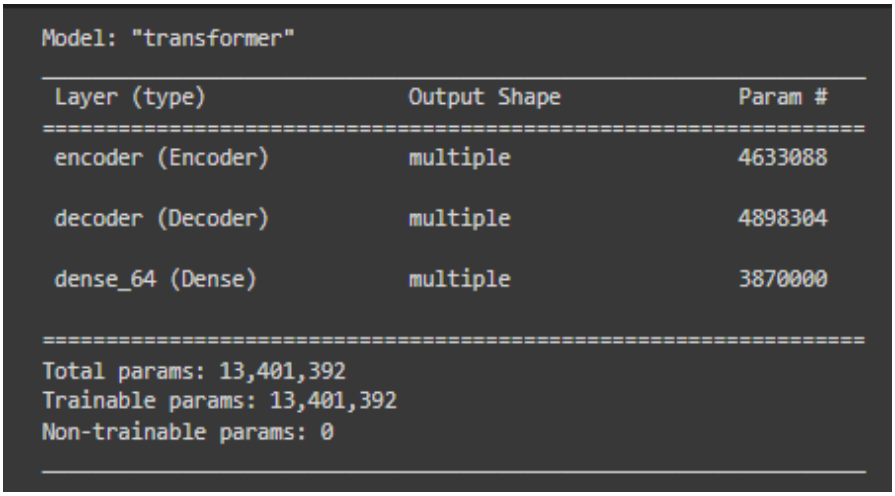

Figure 5: Transformer model architecture for translation

We start by generating a vocabulary of docstrings and function codes and converting them to IDs. This is also known as the index position of the token in the vocabulary. Use BertWordPieceTokenizer. This improves your vocabulary and reduces the breakthroughs of common words into subwords. In addition, add start and end tokens to the vocabulary you want to model to recognize the start and end of a function or document string. Other features such as padding and masking are also used in the vocabulary. If you use padding to create the same number of vectors and understand the existence of a transformer model for the padding masking method, not all docstrings or function tokens contain the same number of tokens. In addition, after this preprocessing, the transformer model is trained at 100 epochs to predict the writing of functional code. Figure 6 shows some of the results of predicted and actual function translation with the help of a machine learning model. In the first result, we can see that model predicted the docstring perfectly while in the second we can see the exact opposite.

Function Code	insertion sort elements for in range 1 len elements anchor elements 1 while 0 and anchor elements elements 1 elements 1 elements 1 anchor
Predicted docstrig	perform an insertion sort for given n elements of array
Actual Docstring	perform an insertion sort for given n elements of array

Function Code	decrypt cipher list int key list int str plain for in range len key int cipher key 2 key plain append chr return join for in plain
Predicted Docstring	function to convert a decimal to binary
Actual Docstring	function to decrypt text using pseudo-random numbers

Figure 6: Output from the precision of docstring

After creating a model which can encode function to docstring successfully, we create the dataset by encoding the function tokens with the transformer model and generate the 768 dimensions of the vector for each function token. After this, the function token and the docstring will be in the same vector space. Figure 7 shows the architecture of the Transformer model used for vectorization.

```
Model: "transformer_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 encoder_2 (Encoder)         multiple                  10656768

 lstm (LSTM)                 multiple                  6052000

 dense_89 (Dense)            multiple                  768768

=================================================================
Total params: 17,477,536
Trainable params: 17,477,536
Non-trainable params: 0
_________________________________________________________________
```

Figure 7: Transformer model architecture for the vectorization

6. Cosine Similarities

In this section, we will be going over how we get the result for the inserted query and how to find the similarity in the docstring to the corresponding function code. In this section, we have chosen cosine similarity over all the other similarity methods because cosine similarity can understand the context and give relevant results.

In data analysis, cosine similarity measures the similarity between two sequences. The sequence is considered a vector of inner product space, and the sine similarity is defined as the inner product of the angles between them, that is, the inner product of the vectors divided by the product of the lengths. Therefore, the cosine similarity does not depend on the

magnitude of the vector, only on the vector's angle. Cosine similarity always belongs to the interval [1,1]. For example, two proportional vectors have a cosine similarity of 1, two orthogonal vectors have 0, and two opposite vectors have a similarity of 1. Cosine similarity is primarily used in positive spaces where the result is limited by [0,1] [19].

For example, in information retrieval and text mining, each word is assigned different coordinates, and the document is represented by a vector that represents the number of times each word appears in the document. Second, cosine similarity is a valuable measure of how similar two documents are in subjects, regardless of the length of the document.

Following is the mathematical representation of the cosine similarity. Consider two vectors a and b, [20].

$$\cos\theta = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \cdot \left\|\underset{b}{\rightarrow}\right\|}$$

7. Finding the Results

Nsmlib[21] creates a hierarchical, navigable Small world graph for each function vector used for the search. The Small World Graph is a mathematical graph in which the overall distance between two randomly selected nodes increases in proportion to the number of nodes in the network. To find the node closest to the query point, one can use the Smallworld property to navigate the graph quickly and efficiently to find the approximate closest node.[22] Use the greedy approach to select any node in the graph randomly, calculate the distance of the adjacent node from the query point of the selected node, and find the adjacent node closest to the query point. If the adjacent node found is closer to the query point than the selected node, move to that node and continue moving in that direction. Follow the same process to find the adjacent node for that node. This is the node closest to the dem query point. This continues until you reach a node with no nodes adjacent to the query point. This node is declared as the neighbor closest to the query point.

Now, we need to find the relevant result for the inserted query. Albert transformer model, which was fine-tuned and trained on the desired dataset, will be used to encode inserted query into a 768-dimensional vector. Using the cosine similarity and k-nearest-algorithm, for all the feature vectors identical to the search query vector, the top 5 answers will be extracted, and by using the index from the dataset, relevant code will be displayed. Figures 8(a) and (b) show the top 2 answers for the inserted query "Matrix multiplication." In the results, we can observe that the model understands the inserted query's semantics. Both the output shows the relevant result for the matrix. However, figure 8(b) shows the Python code for matrix multiplication using a list, the desired output for an inserted query.

```
def inverse(matrix: list[list]) ->(list[list] | None):
    """
    Inverse of a matrix a with n Dimension
    """
    det = determinant(matrix)
    if det == 0:
        return None
    matrix_minor = [[determinant(minor(matrix, i, j)) for j in range(len(
        matrix))] for i in range(len(matrix))]
    cofactors = [[(x * (-1) ** (row + col)) for col, x in enumerate(
        matrix_minor[row])] for row in range(len(matrix))]
    adjugate = list(transpose(cofactors))
    return scalar_multip(adjugate, 1 / det)
```

Figure 8(a): First answer for the inserted query

```
cosine dist:0.0856
 def multiply(matrix_a: list[list], matrix_b: list[list]) ->list[list]:
    """
    Regular matrix multiplication using list
    """
    if _check_not_integer(matrix_a) and _check_not_integer(matrix_b):
        rows, cols = _verify_matrix_sizes(matrix_a, matrix_b)
    if cols[0] != rows[1]:
        raise ValueError(
            f'Cannot multiply matrix'
            )
    return [[sum(m * n for m, n in zip(i, j)) for j in zip(*matrix_b)] for
        i in matrix_a]
```

Figure 8(b): Second answer for the inserted query

8. Conclusion and Future Enhancements

To conclude, in this research, we propose a new approach for semantic code search by using an Albert transformer from hugging face and a seq-seq encoder with a multi-head attention mechanism from a pool of Python code. In this research, by creating own dataset and custom pre-processing techniques such as a unique dataset founded. i.e., python code with compilation error is excluded. Using the Albert model, we converted a text format of a docstring to a 768 dimension of the vector. In addition to that, by creating a custom transformer model aligned with the seq-seq encoder, we encoded a function code to a 768-dimension vector so both function and docstring will be in shared vector space. By using cosine similarity and the k-nearest-neighbor algorithm, we will be able to extract the top 5 results for an inserted query.

However, we are doing supervised learning for a machine learning model by using the given data in this research. A situation may arise when there is no relevant code for the

corresponding inserted query. So, to improve this research, firstly, we can collect more data; secondly, we can create an automatic code generation aligned with this research, which can create a relevant code for the query even though the corresponding code is not available in the dataset. In addition to that, in this research, we only focused on Python language code; however, in further research work, we can focus more on the multi-programming language code search by using the latest transformer.

References

[1] Devlin, J., Chang, M. –W., Lee, K., & Toutanova, K. (2018). BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv, 2018, https://arxiv.org/abs/1810.04805.

[2] Semantic Search using NLP, Ajit Rajput Aug 31, 2020. https://medium.com/analytics-vidhya/semantic-search-engine-using-nlp-cec19e8cfa7e.

[3] Arumugam, L. (2020). Semantic code search using Code2Vec: A bag-of-paths model.

[4] Alon, U., Zilberstein, M., Levy, O., & Yahav, E. (2019). Code2vec: Learning distributed representations of code. Proc. ACM Program. Lang. 3, POPL, Article 40 January, 29 pages. DOI: https://doi.org/10.1145/3290353.

[5] A Gentle Introduction to Calculating the BLEU Score for Text in Python by Jason Brownlee on November 20, 2017, https://machinelearningmastery.com/calculate-bleu-score-for-text-python/#:~:text=BLEU%2C%20or%20the%20Bilingual%20Evaluation,of%20natural%20language%20processing%20tasks.

[6] Top 4 Sentence Embedding Techniques Using Python! purva91 — August 25, 2020, https://www.analyticsvidhya.com/blog/2020/08/top-4-sentence-embedding-techniques-using-python/.

[7] Husain, H., Wu, H. –H., Gazit, T., Allamanis, M., & Brockschmidt, M. (2019). CodeSearchNet Challenge: Evaluating the State of Semantic Code Search.

[8] Simple Search Engine with Elastic Search, Jul 14, 2020, Vivek Vinushanth Christopher. https://towardsdatascience.com/simple-search-engine-with-elastic-search-d36086591d26.

[9] Lv, F., Zhang, H., Lou, J. –G., Wang, S., Zhang, D., & Zhao, J. (2015). CodeHow: Effective code search based on API understanding and extended boolean model (E). 260-270. DOI: 10.1109/ASE.2015.42.

[10] Białecki, A., Muir, R., & Ingersoll, G. Apache Lucene, vol.4, pp.17-24.

[11] Lv, F., Zhang, H., Lou, J. –G., Wang, S., Zhang, D., & Zhao, J. (2015). CodeHow: Effective code search based on API understanding and extended boolean model (E). pp.260-270. DOI: 10.1109/ASE.2015.42.

[12] Traceback in Python, 01 Aug 2020, https://www.geeksforgeeks.org/traceback-in-python/#:~:text=Traceback%20is%20a%20python%20module,stack%20trace%20at%20any%20step.

[13] Deciphering Python: How to use Abstract Syntax Trees (AST) to understand code, MATT LAYMAN. https://www.mattlayman.com/blog/2018/decipher-python-ast/.

[14] How do you Split Data into Training and Testing Sets in Python using sklearn? https://www.askpython.com/python/examples/split-data-training-and-testing-set.

[15] Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., & Soricut, R. (2019). ALBERT: A Lite BERT for self-supervised learning of language representations. arXiv 2019. https://arxiv.org/abs/1909.11942.

[16] HuggingFace Library - An Overview. Lalithnarayan C, https://www.section.io/engineering-education/hugging-face/.

[17] Run masked LM/next sentence masked_lm pre-training for ALBERT. https://github.com/google-research/albert/blob/master/run_pretraining.py.

[18] How to Develop an Encoder-Decoder Model with Attention in Keras by Jason Brownlee on October 17, 2017. https://machinelearningmastery.com/encoder-decoder-attention-sequence-to-sequence-prediction-keras/.

[19] Cosine Similarity – Understanding the Math and how it Works, October 22 2018, by Selva Prabhakaran. https://www.machinelearningplus.com/nlp/cosine-similarity.

[20] How to calculate Cosine Similarity. https://clay-atlas.com/us/blog/2020/03/27/cosine-similarity-text-calculate-python/.

[21] Approximate Vector Search using NMSLIB. https://radimrehurek.com/gensim/similarities/nmslib.html.

[22] Logvinov, A., Ponomarenko, A., Krylov, V., & Malkov, Y. (2010). Metrized small world approach for nearest neighbor search. DOI: 10.15514/SYRCOSE-2010-4-30.

[23] Code reuse: How to reap the benefits and avoid the dangers, by Ken Prole, Nov 12, 2018, https://codedx.com/blog/code-reuse-how-to-reap-the-benefits-and-avoid-the-dangers/.

[24] Python NLTK | tokenize.regexp(), 07 Jun, 2019, https://www.geeksforgeeks.org/python-nltk-tokenize-regexp/

[25] BERT: Google's New Algorithm That Promises to Revolutionize SERPs, Larissa Lacerda, Nov 30, 2020. https://rockcontent.com/blog/google-bert/#:~:text=In%20Google%2C%20BERT%20is%20used,content%20just%20as%20we%20do.