# Indian Institute of Technology, Kanpur


**CS657A: Information Retreival**
**Project Report**

---

**Title: Semantic Search on Codebases**

---

**Supervised By: Prof. Arnab Bhattacharya**

## Submitted By: Group 13


| | |
|---|---|
| Maj Ashish Ahluwalia (21111073) | ashisha21@iitk.ac.in |
| Binay Kumar Suna (21111021) | binayas21@iitk.ac.in |
| Chabil Kansal (21111022) | chabilk21@iitk.ac.in |
| Shubham Sinha (21111409) | ssinha21@iitk.ac.in |

# Contents


**1 Abstract** **2**

**2 Broad Aims of the project** **2**

**3 Approaches Used** **3**

**4 Approach - 1** **4**

4.1 Dataset Implementation (Part 1): . . . . . 4

4.1.1 Data Source . . . . . 4

4.1.2 Data Collection: . . . . . 4

4.1.3 Data Preprocessing: . . . . . 4

4.2 Translating Functions to its English Description (Docstrings) using Transformer (Part 2): . . . . . 5

4.3 Searching via Semantic Similarity (Part 3): . . . . . 7

4.4 Building the Search Engine Web Application (part 4): . . . . . 8

**5 Approach - 2** **9**

5.1 Dataset Implementation (Part 1): . . . . . 9

5.2 Converting Docstrings to Vectors (Part 2): . . . . . 9

5.3 Converting Functions to Vectors (Part 3): . . . . . 10

5.4 Modifying the architecture . . . . . 11

5.5 Creating the semantic search engine . . . . . 12

5.6 Building the Search Web Application (Part 4): . . . . . 12

**6 Conclusion** **12**

**7 Future Work** **13**

**8 Important Links and References** **13**

# 1 Abstract

The power of modern search engines is undeniable: you can summon knowledge from the internet at a moment's notice. Unfortunately, this superpower isn't omnipresent. There are many situations where search is relegated to strict keyword search, or when the objects aren't text, search may not be available. Furthermore, strict keyword search doesn't allow the user to search semantically, which means information is not as discoverable. Today, we share a reproducible, minimally viable project that illustrates how you can enable semantic search for code texts. We show you how to create a system that searches python code semantically and also try to compare the results of semantic search to keyword search. We developed a web application that will take a search query and try to give python functions/definitions that match the query semantically and syntactically.

# 2 Broad Aims of the project

The primary aim is to develop a web application that will give it's user interface to search for functions/codes that are semantically similar i.e. contain the same meaning as their search query. For this our application has to successfully extract python codes that are semantically similar to the query, thus, providing better search results in comparison to keyword search. Also provide user's a application to search the codebase's without getting into the misery of trying to search it in the various links provided by google search.

# 3 Approaches Used

| Approach 1 | | |
|---|---|---|
| | Details | Remarks |
| Part 1 | Dataset Implementation | We discussed on gathering the data for training our model and preprocessing the raw data to make it useful for our model |
| Part 2 | Translating functions to its English description (docstrings) using transformer | 1. Data Formatting<br>2. Modification of Transformer<br>3. Generating a Vocabulary<br>4. Encoding the function to generate a sentence by training via Transformers. |
| Part 3 | Searching via Semantic Similarity | The basics of the word embedding model GloVe is used, which maps words into numerical vectors. Carrying out unsupervised learning |
| Part 4 | Building the Search Engine Web Application | We have used flask as front end user interface for query searching |
| Approach 2 | | |
| Part 1 | Dataset Implementation | Data set of Part-I ( Approach 1 ), as mentioned in above column was used in this approach as well |
| Part 2 | Converting Docstrings to Vectors | The docstrings are converted to vectors using a pretrained ALBERT model which is fine-tuned on our data set. |
| Part 3 | Converting Functions to Vectors | In continuation with part 2 of approach 1 we have performed the conversion of functions to 768-dimensional vectors. |
| Part 4 | Building the Search Engine Web Application | We have used flask as front end user interface for query searching |

# 4 Approach - 1

This section is further divided into three subsections: Dataset Implementation and Data Cleaning, Translating Functions to its English description (Docstrings) using transformer and Searching via Semantic Similarity.

## 4.1 Dataset Implementation (Part 1):

### 4.1.1 Data Source

The dataset's being used in this project are github python repositories and Raw Python Code Corpus from the following link: CodeSearchNet dataset. These dataset contain hundred of files containing python codes. Each python file contains various functions/classes and their descriptions in the form of comments.

### 4.1.2 Data Collection:

Our Semantic Search Engine uses large code bases which are available in this google drive link.The data set contains code from repositories of diverse domains and can help in building a general purpose search engine.This corpus contains 1,50,000 files which can be seen below:

| | nwo | path | content |
|---|---|---|---|
| 0 | 2_hidden_layers_neural_network.py | Python_files/2_hidden_layers_neural_network.py | """\nReferences:\n - http://neuralnetworksa... |
| 1 | 3n_plus_1.py | Python_files/3n_plus_1.py | from __future__ import annotations\n\n\ndef n3... |
| 2 | a1z26.py | Python_files/a1z26.py | """\nConvert a string of characters to a seque... |
| 3 | abbreviation.py | Python_files/abbreviation.py | """\nhttps://www.hackerrank.com/challenges/abb... |
| 4 | abs.py | Python_files/abs.py | """Absolute Value."""\n\n\ndef abs_val(num):\n... |
| ... | ... | ... | ... |
| 564 | world_covid19_stats.py | Python_files/world_covid19_stats.py | #!/usr/bin/env python3\n\n"""\nProvide the cur... |
| 565 | xor_cipher.py | Python_files/xor_cipher.py | """\n author: Christian Bender\n ... |
| 566 | zellers_congruence.py | Python_files/zellers_congruence.py | import argparse\nimport datetime\n\n\ndef zell... |
| 567 | z_function.py | Python_files/z_function.py | """\nhttps://cp-algorithms.com/string/z-functi... |
| 568 | __init__.py | Python_files/__init__.py | |

569 rows × 3 columns

### 4.1.3 Data Preprocessing:

For this task we will be using the AST library which just as python compilers converts the source code into an abstract syntax tree for analysis. It identifies all different components in the source code and helps us extract them for processing.
We are interested in extracting the function definitions along with its corresponding docstring removing other information like comments, decorators and function signatures. After we extract the function definition and its docstring we tokenize each of them to remove punctuation, decorators and convert all the tokens to lower case.
Once we have extracted our function-docstring pairs and their tokens which are free from decorators and other unwanted elements,we stack our findings in a data-frame with every row containing details about a function and its corresponding docstring.All entries which were duplicates with respect to function definitions or function tokens were removed. It

was decided not to randomly split the data set instead to group the entries according to the repositories they belong and then split the data set.
Below is the preview what the training set looks like.The function tokens and docstring tokens are what will be fed downstream into the models. The other information is important for diagnostics and bookeeping.

| nwo | path | function_name | lineno | original_function | function_tokens | docstring_tokens | url | function_tokens_count |
|---|---|---|---|---|---|---|---|---|
| cycle_sort.py | Python_files/cycle_sort.py | cycle_sort | 7 | def cycle_sort(array: list) ->list:\n """\n... | cycle sort array list list array len len array... | cycle sort 4 3 2 1 1 2 3 4 | https://github.com/cycle_sort.py/blob/master/P... | 122.0 |
| greedy.py | Python_files/greedy.py | test_greedy | 42 | def test_greedy():\n """\n >>> food = ["... | test greedy | food burger pizza coca cola rice sambhar chick... | https://github.com/greedy.py/blob/master/Pytho... | 105.0 |
| graph_list.py | Python_files/graph_list.py | add_edge | 84 | def add_edge(self, source_vertex: T, destinati... | add edge self source vertex destination vertex... | connects vertices together creates and edge fr... | https://github.com/graph_list.py/blob/master/P... | 101.0 |
| unknown_sort.py | Python_files/unknown_sort.py | merge_sort | 9 | def merge_sort(collection):\n """Pure imple... | merge sort collection start end while len coll... | pure implementation of the fastest merge sort ... | https://github.com/unknown_sort.py/blob/master... | 77.0 |
| game_of_life.py | Python_files/game_of_life.py | run | 54 | def run(canvas: list[list[bool]]) ->list[list[... | run canvas list list bool list list bool curre... | this function runs the rules of game through a... | https://github.com/game_of_life.py/blob/master... | 79.0 |

## 4.2 Translating Functions to its English Description (Docstrings) using Transformer (Part 2):

To begin training using transformers we need to prepare and format our data so that it can be used by transformers as an input to train upon.The steps for it are -

1. **Generating a Vocabulary**
   We begin with generating two separate vocabularies one from our function tokens and the other from the docstring tokens and subsequently use these vocabularies to convert our function and docstring tokens into ids(which is the token's index position in the vocabulary). We do not use the SubwordTextEncoder as suggested in the tutorial instead we use the BertWordPieceTokenizer. This during our analysis was found to give us a better vocabulary as we found lesser breakage of common words used into subwords.

2. **Encode Function**
   Since we modified the tokenizer we need to make a few changes to the encode function from the tutorial.

3. **Sorting the Data**
   During the preprocessing phase the last step was to sort the data set according to the count of function tokens. We now will be able to reap its benefits as when creating batches the padding tokens are added based on the largest number of function tokens in each batch. Sorting the data set based on the count of function tokens in an entry will cause all similar sized inputs being together in a batch, which will reduce the padding tokens added to every input drastically. On shuffling the train data set we would loose this benefit and will have varied lengths of function tokens in every batch and hence the padding tokens added per batch will increase. Therefore, we must not shuffle the data set.

4. **Values of start token and end token**
   Since we used a different tokenizer the start token in our case is '[CLS]' and it's id

is 2 and end token is '[SEP]' and its id is 3 and for the tokenizer used in the tutorial it is the last two ids of the vocabulary. We must replace these values wherever they are used.

Examples of the output obtained from our trained transformer model is shown below.

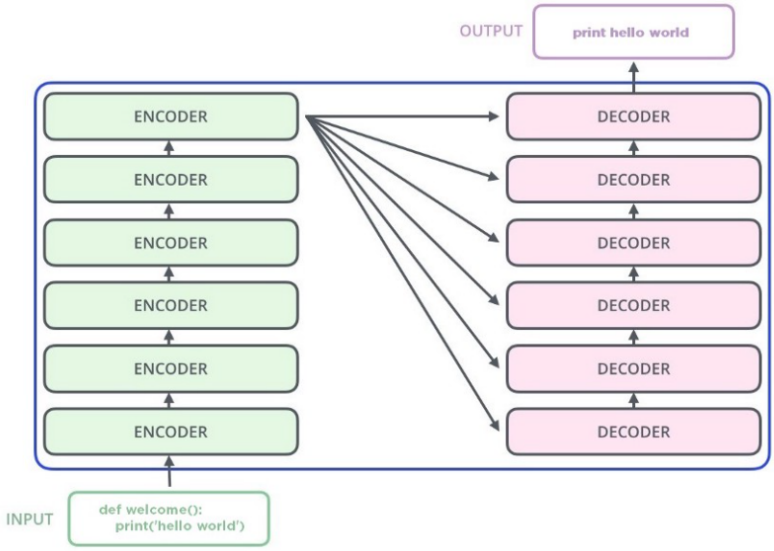


Figure 1: Transformer Translation Function in English

```
Input: open filename return file filename
Predicted translation: open a file or file like object
Real translation:  open a hdf5 file

Input: num cores self raise notimplementederror
Predicted translation: returns the number of cores
Real translation:  return the number of processes

Input: init self none self state
Predicted translation: initialize the object
Real translation:  initialize an instance

Input: html self return render markdown self wiki path
Predicted translation: return the html for this page
Real translation:  render the page for display
```

Figure 2: Translation Made by the Model

The function to text conversion is stored in a translation file, the output of the file is as shown below:-

20000: "add a user to the user.",
20001: "r sends a log file to the device.",
20002: "convert a string to a string.",
20003: "create a new function that is used to create a new function.",
20004: "set the number of values for a given type.",
20005: "set the value of a single variable.",
20006: "r.. versionadded : : 2015. 8. 0",
20007: "return a list of ( name _ id ) tuples.",
20008: "run the request.",
20009: "r compute the expectation of a gaussian distribution of a given function.",
20010: "create a new elastic network with a given name.",
20011: "save the data to the file.",
20012: "r compute the difference between two columns.",
20013: "return the default options for the given environment.",
20014: "ensure that the named file exists.",
20015: "return a dictionary of parameters for a given field.",
20016: "return a list of dictionaries that are used to create a new list of dictionaries.",
20017: "ensure that the named object is present in the given namespace.",
20018: "r return a list of vim. vm. virtualdevicespec objects representing the specified properties.",
20019: "add a new layer to the layer.",
20020: "returns a list of tasks that are not in the context.",
20021: "create a new instance from a module.",
20022: "run the plugin",
20023: "creates a new state for a given state.",
20024: "set the default configuration.",
20025: "r configures the container.",
20026: "return a new value.",
20027: "r set the number of devices to be used to use this function to ensure that the user is not well as the number of ways.",
20028: "return a dictionary of vim. vm. vm. vm. vm. vm. vm. vm. vm. vm. vm. vm. vm. vm _ name.",
20029: "returns a dictionary of config files.",
20030: "run a single file on the system.",
20031: "set the default value for the section section section.",
20032: "return the default value for a given section.",
20033: "return the default value for a section of the section section.",
20034: "ensure that the named key is present in the config file.",
20035: "return a dictionary of parameters for the given key.",
20036: "r return a dictionary of parameters for a given key.",
20037: "run a command.",
20038: "add a new package to the given package.",
20039: "add a new message to the database.",
20040: "return a new file with the given name.",

## 4.3 Searching via Semantic Similarity (Part 3):

Semantic similarity scores words based on how similar they are, even if they are not exact matches. It borrows techniques from Natural Language Processing (NLP), such as word embeddings. This is useful if the word overlap between texts is limited, such as if you need 'fruit and vegetables' to relate to 'tomatoes'.
We have used the word embedding model **GloVe** which maps words into numerical vectors which are points in a multi-dimensional space so that words that occur together often are near each other in space.We create a similarity matrix, that contains the similarity between each pair of words, weighted using the term frequency then calculate the soft cosine similarity (as regular cosine similarity return zero for vectors with no overlapping terms),which considers the word similarity between the query and each of the documents. Moreover,semantic similarity is good for ranking content in order, rather than making specific judgements about whether a document is or is not about a specific topic.We have used self-contained **DocSim class**, that can be imported as a module and used to run semantic similarity queries without additional code.

## 4.4 Building the Search Engine Web Application (part 4):

We have used **Flask** which is a small and lightweight Python web framework that provides useful tools and features that make creating web applications in Python easier. It gives developers flexibility and is a more accessible framework for new developers since you can build a web application quickly using only a single Python file. Flask is also extensible and doesn't force a particular directory structure or require complicated boilerplate code before getting started.Flask uses the Jinja template engine to dynamically build HTML pages using familiar Python concepts such as variables, loops, lists, and so on.


Figure 3


Figure 4

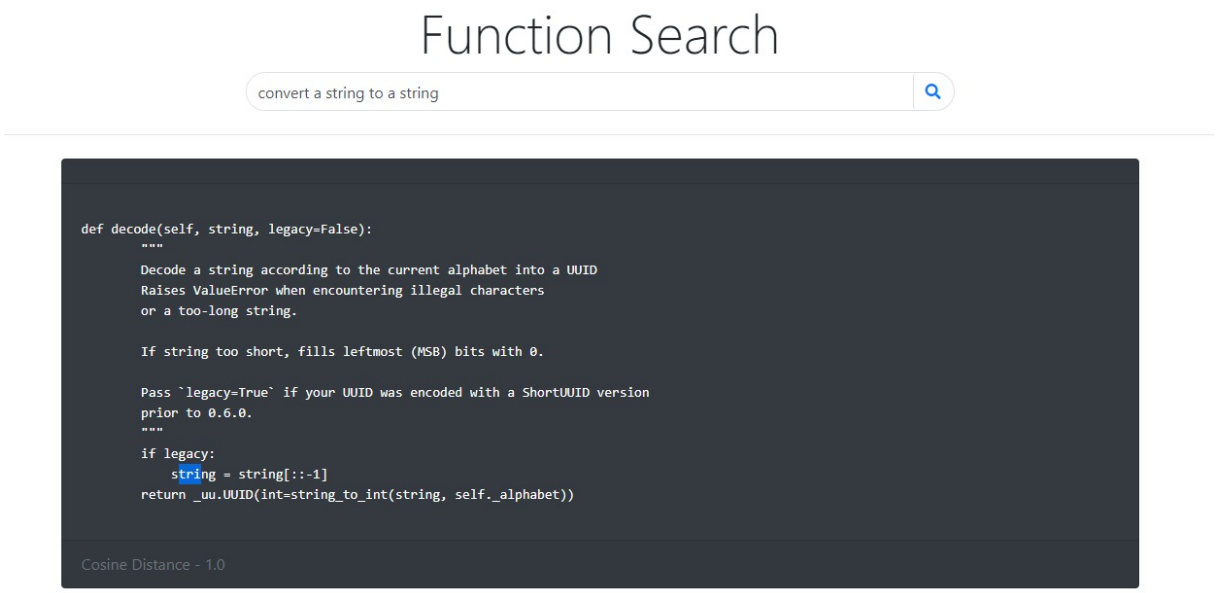


Figure 5

# 5 Approach - 2

## 5.1 Dataset Implementation (Part 1):

The Dataset implementation and Preprocessing are similar to approach 1.

## 5.2 Converting Docstrings to Vectors (Part 2):

The docstrings are converted to vectors using a pretrained ALBERT model which is fine-tuned on our data set. ALBERT is chosen because its faster to train, low on memory consumption and trains on harder tasks as compared to BERT.
ALBERT is trained on a large English corpus. A code description text is slightly different from the sentences in these documents as words have special meanings for certain words not in adherence to the meaning in English used while general communication. To solve this problem we could fine-tune ALBERT i.e. fine-tune the word embedding weights and the weights of the encoders to make it understand the words and infer its meaning from a computer programming context. The argument though being valid as using our own vocabulary would help the ALBERT learn representations for even programming jargons, like 'SQL, csv ' etc., which might not be present in its own vocabulary.
Steps before training are need to download an ALBERT pretrained model from tensorflow hub, download two scripts from the ALBERT repository in github which are present in *create_pretraining.py* and *run_pretraining.py* and write the docstrings into a text file.
Now, run the *create_pretraining_data.py* file which creates data in the format which can be used by ALBERT to do inference on. Once we have generated the training data from

the above process we can begin the training process by running the *run_pretraining.py* file. This does incremental training on the pre-trained ALBERT model using the same methods of training, i.e. n-gram Masked Language Modelling and Sentence Order Prediction using docstrings from our data set. This is favorable for our task as it fine-tunes weight custom to our data set of docstrings.

```
loss = 3.9983284

masked_lm_accuracy = 0.41996095

masked_lm_loss = 3.5465455

sentence_order_accuracy = 0.7625

sentence_order_loss = 0.4517824
```

Figure 6: Training Report

Used HuggingFace library to realize our trained ALBERT model by converting tensorflow checkpoint file to a pytorch dump. We get *pytorch_model.bin* file.To get docstring vectors from it, encoded the input via the *albert_tokenizer* and then send it as input to the model. We can scale this to convert all our docstrings in the train set to vector representation. Once done we save it in a csv file to fetch it comfortably for later use.

## 5.3 Converting Functions to Vectors (Part 3):

We converted our docstrings to 768-dimensional vectors in part 2 of approach 2 Similarly next we need to convert the functions into 768-dimensional vectors such that the function vector and the docstring vector are in a shared vector space.
Transformer is widely popular due to its state-of-the-art results in the field of machine translation, hence we decided to train a transformer model to translate functions into their English descriptions (meaning). After training the model we would remove the decoder from the transformer architecture and use the trained encoders of the transformer to give an encoded representation of the function. This representation is further passed through LSTM and Dense layers to get the output as a 768-dimensional vector which is trained to share the vector space and be similar to docstring vectors.
The Steps used for the training using transformers:

1. **Generating a vocabulary** —
   First we have to generate two separate vocabularies one from our function tokens and the other from the docstring tokens and subsequently use these vocabularies to convert our function and docstring tokens into ids.We used the BertWordPiece-Tokenizer for this purpose. This during our analysis was found to give us a better vocabulary as we found lesser breakage of common words used into subwords.

2. **Encode function** —

We have used encoders of the transformer to give us the encoded representations of the the function.

3. **Preprocessing for batch Creation** —
   During the preprocessing phase the last step was to sort the data set according to the count of function tokens. When creating batches the padding tokens are added based on the largest number of function tokens in each batch. Sorting the data set based on the count of function tokens in an entry will cause all similar sized inputs being together in a batch, which will reduce the padding tokens added to every input drastically. On shuffling the train data set we would loose the benefit of sorting and will have varied lengths of function tokens in every batch and hence the padding tokens added per batch will increase.

4. **Values of start token and end token** —
   The start token in our case is '[CLS]' and it's id is 2 and end token is '[SEP]' and its id is 3. We must replace these values wherever they are used.

After the above steps we have a trained transformer which can take as input a function and translate it in English. The above task was performed by us as a precursor to our task of conversion of functions to 768-dimensional vectors. Doing so has helped us initialize our encoders weights and made them understand functions in English. This eased the task of conversion of functions to vectors in shared vector space with the docstring vectors.
Let's begin with making our trained transformer convert functions to vectors, **Generating the data set** We generated a data set of functions and their corresponding docstring's vector generated from the ALBERT model. With this we will be able to train our model to learn to map the functions to a vector similar to its corresponding docstring vector and in this way the function vectors and the docstring vector will be in a shared vector space. The main challenge with creating the data set is that each function will have a 768-dimensional docstring vector and we have more than a lakh of them! Storing such a huge data set for processing and training into variable will be impossible on average systems with low RAM. So, for this, in each step of the training process we generate a batch from our data set. These batches are what are sent through the model for inference and updation of its weights. From this it can be observed that technically at a certain point of time during training all we need is one batch of training data and once this batch is processed it can be discarded so that it doesn't take up memory and the next batch can be fetched. This would not need the entire data set to be in RAM. We can fetch the docstring vectors needed from the text files during the training process on the fly and generate batches for training.

## 5.4 Modifying the architecture

As we have trained the transformer in previous steps now we modify it by not including the decoder layers in the transformer instead send the input from the encoder layers of the transformer to an LSTM layer which passes its output to a dense layer to finally output a 768-dimensional vector.

### 5.5 Creating the semantic search engine

As we have created both the function as well as the docstring vectors now have to use them to find semantically similar functions to the search query, we used the Non-Metric Space Library (nmslib) which does the heavy-lifting for us to construct a scalable and efficient search infrastructure.
The process followed is to encode the search query to a vector using our trained ALBERT model. All the function vectors similar to the search query vector are searched and we are returned index values and the distances of five nearest neighbors to the search query. We extract its details using the index value which corresponds to its index value in the data set and display the results

### 5.6 Building the Search Web Application (Part 4):

We have used **Flask** which is a small and lightweight Python web framework that provides useful tools and features that make creating web applications in Python easier. It gives developers flexibility and is a more accessible framework for new developers since you can build a web application quickly using only a single Python file. Flask is also extensible and doesn't force a particular directory structure or require complicated boilerplate code before getting started.Flask uses the Jinja template engine to dynamically build HTML pages using familiar Python concepts such as variables, loops, lists, and so on.
Once the query is typed in the search box, we embed it in a GET request and convert it to a vector which is further used by our ALBERT model.Using the search graph we find the the 5 nearest neighbors. The search function will return us index values and using theses indices the function definition and URL of these nearest neighbors are gathered from the data corpus. Their cosine distance from the search query is also captured and these results are displayed in the results page of the web application.

## 6 Conclusion

In the project we have implemented two approaches, the approach one used the transformer model to convert the functions to English meaning, and, thereafter that meaning is used to semantically match with query to find the best functions corresponding to the search query. This way is used to carry out a semantic search on the codebase's. In approach two we have generated the vectors for both functions as well as the docstrings in the shared vector space, the doc-string vectors are used in the finding the function vectors making both vectors similar, then function vector's are used to carry search for the desired function for our search query using the nearest neighbour search on Hierarchical Navigable Small World (HNSW) graph's. At the end we show how both the approaches use semantic search to find the better/more relevant result's when compared the traditional keyword search.

# 7 Future Work

As Semantic Search is a very rich domain to work on, thus it has applicability in various fields of Information Retrieval. The approaches shown above can be extended for different kind of arbitrary objects such as images or sound bites. Also this code search application can be extended to work on more personalized data/codebase's as compared to the more general standard code database's.

# 8 Important Links and References


1. Semantic Code Search Using Transformers and BERT- Part I: Overview and Data Preprocessing

2. Semantic Code Search Using Transformers and BERT- Part II: Converting Docstrings to Vectors

3. Semantic Code Search Using Transformers and BERT- Part III: Converting Functions to Vectors  Deploying the Search Engine

4. How To Create Natural Language Semantic Search For Arbitrary Objects With Deep Learning

5. CodeBERT: A Pre-Trained Model for Programming and Natural Languages

6. code2vec: Learning Distributed Representations of Code

7. ALBERT Implementation

8. Research Paper on ALBERT

9. TensorFlow