Semantic Code Search Engine with a Survey of vector based data embedding techniques

BTP Report submitted for the Degree of
Bachelors in Computer Science and Engineering

by

Akhil and Yagyansh
(160101011, 160101079)

under the guidance of

Prof. Amit C. Awekar

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI
GUWAHATI - 781039, ASSAM

CERTIFICATE

*This is to certify that the work contained in this thesis entitled "**Comprehensive Survey of embedding based vector representation of data in various fields.**" is a bonafide work of **Akhil Chandra Panchumarthi and Yagyansh Bhatia (Roll No. 160101011, 160101079)**, carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati under my supervision and that it has not been submitted elsewhere for a degree.*

Supervisor: **Amit C. Awekar**

Assistant Professor,

May, 2020 — Department of Computer Science & Engineering,

Guwahati. — Indian Institute of Technology Guwahati, Assam.

Acknowledgements

We're highly indebted to **Prof. Amit C. Awekar** for his time, guidance, constant supervision and invaluable discussions as well as for providing necessary information and resources required for the project.

We want to extend our sincere thanks to **Abhishek** (former, PhD student) for all the time and kind co-operation which helped us throughout the project.

Lastly, we want to thank our colleagues **Aayush and Arpan** for all their help and encouragement. We want to extend our sincere thanks to everyone.

Contents

Chapter 1

Introduction

Our initial goal was towards writing a comprehensive survey paper of vector based embedding methods across various fields (code2vec, word2vec, code2seq, hin2seq, node2vec, img2vec, doc2vec). Since the introduction of word2vec, the idea of continuous vector representations has propagated to various fields, and such representations of data are being used quite extensively.

The main Goal of this survey is to put together all the applications of embedding based approaches, while understanding major bottlenecks in generating and using such representations through a cross domain study of this idea.

In the first part of this document we summarize the papers that we reviewed(so far) in the following areas: word2vec, code2vec, code2seq, graph2vec, node2vec.

Through our study of code2vec and code2seq, we developed an interest in using natural language queries to search for code snippets. Github's recently proposed open challenge "CodeSearchNet" made it possible to explore this interest. They've curated a large database of methods with their metadata, and made it open source for the community to use and contribute. As this was related to our study, we decided to attempt the challenge as part of our BTP.

In the second half of this document, we illustrate existing techniques for this semantic word search problem (through natural language queries) and then explore various possible solutions for this problem, defining the bedrock for our second phase.

Chapter 2

Summaries and Conclusions: Literature study.

In this section, we present a comprehensive review of the literature we went through, and some insights on recurring patterns and fundamental ideas that we've observed.

2.1 Word2Vec[1]

Word2Vec was a revolutionary step as the idea of continuous vector representation of objects that can be obtained from simple models trained over much larger dataset has been propagated to a variety of fields.

None of the previous approaches were able to be trained on more than a few hundred-million words, but word2vec with its simplicity allowed training over a text corpus with over a billion words. In this work, authors have proposed a novel method to represent words from Large text corpus in continuous vector space with a much lower dimensionality, while still capturing some very interesting properties. The representations not only capture the syntactic similarities, but also the semantic similarities.

Surprisingly, Similar words ended up together, and had *"multiple degrees of similarities"*. This means rep("man") - rep("boy") + rep("girl") is very close to rep("woman").

They provide an efficient and a simple method that is quite fast (can train on large sets of about billion samples in about a day, over numerous epochs).

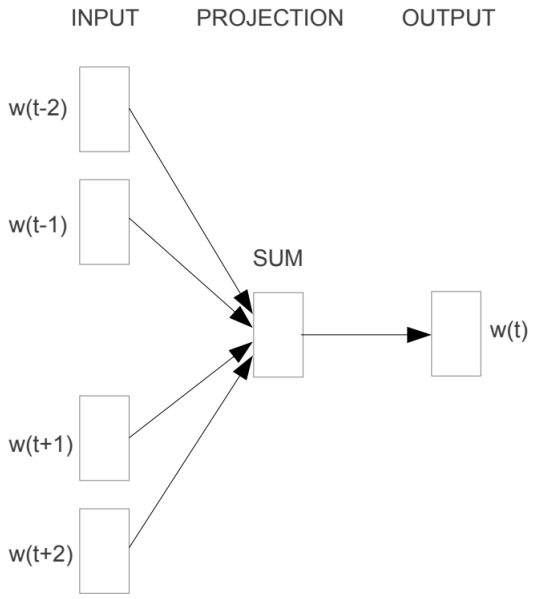
Training complexity : E x T x Q

E : number of epochs. (3-50)

T : number of words in the corpus. (usually upto 1B)

Q : defined further for each model.

The Authors have proposed two architectures, CBOW and Skip-gram model for learning distributed representations of words, while trying to minimise the computationally complexity.



CBOW

2.1.1 Continuous Bag of Words

The idea is to use the some words from the history, as well as from the future as "context" and try to predict the middle word. Some key points in this architecture are mentioned

below:

- The Order of words doesn't matter.
- It Uses words from future too(i.e. words that sequentially occur after the current word) and the best performance happens when one considers 4 words from past and 4 words from future window.

Q = ND + D log(V)

V : size of vocabulary.

N : size of context window used.

D : projected dimensionality.

2.1.2 Skip-Gram

This is very similar to the CBOW method described above, but instead of predicting the middle (current) word through the context, this approach attempts to predict the context through the middle word, essentially reversing the CBOW approach. Words in a certain range (window) are considered to be part of the context, and while increasing the window size results in better quality of the vector representations, this also leads to an increase in the computational complexity.

The complexity of this architecture is: Q = CD(1+log(V)).

C : maximum distance of words.

D : Projected dimensionality.

V : size of vocabulary.

2.1.3 Comparing the two approaches

The training time for skip-gram is more than that of the CBOW method to some extent.

While CBOW is much more efficient at capturing the syntactic similarities between words, skip gram performs significantly better when it comes to semantic similarities.

2.2 Code2vec[2]

Code2vec fundamentally deals with the problem of reducing a code snippet, usually a method, to a continuous vector representation. Doing so effectively opens up the possibility of solutions to well known programming-language tasks through well known neural techniques. It allows for solutions to various prediction problems without having to run the code snippet in question. The main application that this paper looks into is proper naming of a given method. By coming up with a novel technique involving using the AST of the given code snippet, the paper effectively revolutionized the concept of generating efficient and accurate continuous vector representations of code snippets. Other possible applications include assisting in code reviews, and suggesting API functions to the developer to prevent "reinvention of the wheel". The take home message of this paper is that by coming up with a better data representation, code can now be semantically better understood by learning algorithms. The resulting prediction method effectively senses semantic differences in syntactically similar code snippets.

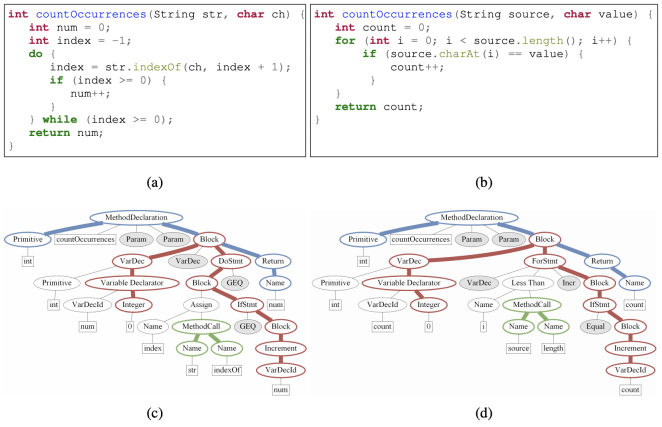
2.2.1 Overview of the Code representation problem

Code is fundamentally different from natural languages. When we look at any text, whether code or natural language, we look at the syntactic and semantic meaning. Code snippets have significantly more emphasise on structure and hence the syntactic meaning when compared to natural language snippets. For this reason the seemingly intuitive idea of tokenizing words in various NLP techniques usually fails in the code domain. This derives from the fact that the intuition of understanding words as a sequence is a more valid in the natural language domain. The idea of understanding code as a sequence of "words" is not an accurate model.

Therefore, standard nlp techniques would either fail to render usable vector representations(incorrect language interpretation model) or would require unreasonably large amounts of data to arrive at reasonable vector representations (correct interpretation language model, however the overhead of having to learn the syntax of a programming language is quite large). Added to that is the problem of generalizing over different programming languages; each has its own syntax and concepts.

2.2.2 The Concept of AST

The answer, as suggested by the previous work of Alon et al. 2018 and Raychev et al. 2015, lies in AST's syntactic paths. AST, better known as abstract syntax trees, is essentially a tree based representation of a code snippet. They are an important component of how a compiler sees the code, and are defined by the syntax and semantics of that programming language. Therefore, they are universal i.e. can be generated by any code snippet of any programming language. An AST is the more intuitive way of representing a code snippet, rather than the sequence of tokens as suggested by NLP methods. An example of the semantic capturing strength of an AST is shown below, not the similar syntactic structures of the code snippets.



Though we now have a graph, or rather a tree, as an accurate representation of the data

that we know to be a valid representation of a code snippet, the next problem is capturing the essence of the graph into a representable format(vectors). This can be loosely translated to capturing the semantics of the code represented by the AST. This is done through AST's syntactic paths. A syntactic path is a path from an AST leaf node (called a terminal) to another different AST leaf node. The path is defined by the terminals, and the set of non-terminals i.e. intermediate nodes of the path. An AST is now better represented as a set of AST syntactic paths. Each syntactic path is converted into a 3 part vector, two parts which represent the terminal values and one part that represent the path.

A set of paths in a tree is not a suitable input to a neural function whose output lies in a vector space. The key insight of this paper lies in compressing the set of syntactic paths into a semantically accurate code vector. Each syntactic path represented by a path-context(better described in the next section), and each path-context is represented as a vector. The set of paths is now a set of vectors, and by using an attention layer, the set is reduced into the required code vector.

2.2.3 Model overview and Architecture

A path-context is made of three components, defined as $< x_s, p, x_t >$. x_s defines the value of the starting terminal, x_t defines the value of the ending terminal, and p defines the path taken to travel from terminal s and t. p is defined by the sequence of non-terminals, starting and ending at s and t respectively, where an arrow defines the direction to travel between sequence elements s_1 and s_2; upwards arrow implies s_2 is the parent of s_1 and a downwards arrow implies s_2 is the child of s_1.// The model is learns the following representations:

- *path_vocab*:- the set of embeddings over the set of all possible(considered) paths. To limit the set of all possible paths we define hyperparameters k and $width$, which define the maximum length and width i.e. the distance between the child indices of the same intermediate node respectively.

- *value_vocab* :- the set of embeddings over the set of all possible terminal values. This

is dependent on the code snippet.

- Matrix W - A fully connected layer is used to compress the $3d$ dimension of the path-context vector to d.
- Attention vector a - the attention vector, which determines the weight each path-context will contribute to the code vector.
- *tag_vocab* - The vector vocabulary (of dimension d) that each tag is represented by. This vocabulary is also learned during training.

A code snippet C has a corresponding label L. Each path context c_i is converted into its $3d$ length vector by looking up its terminal values and path from the *value_vocab* table and *path_vocab* table respectively respectively. It is then compressed into dimension d path context compressed vector by a matrix W of dimension $(d, 3d)$, i.e. the fully connected layer, via the operation $tanh(W * c_i)$. Each c_i is now a d dimension vector. The attention each path-context gets is defined by the $alpha_i$ of each such c_i to the code vector V. $alpha_i$ is defined by the softmax of its dot product with the attention vector a. This is ensure that each weight $alpha_i$ is positive. The code vector V is therefore the weighted average of the path context compressed vectors c_i, weighted by its attention factor $alpha_i$, and is of dimension d. Therefore, for a given (X, Y) supervised learning pair, the prediction vector p of dimension d is drawn by soft-max of the dot product of the code vector V with the tag vectors T_i where i varies over the number of tags in the data set. p is therefore a probability distribution of the predicted tag of the corresponding code vector V. Training is done with cross-entropy loss of the two probability distributions p and q, where q is simply a one-hot encoding of the correct tag value, drawn from Y. Define the index where there is a 1 in q as Y_{true}. This reduces the loss term to be negative log of the probability $p_{Y_{true}}$.

The greatest advantage of this paper lies in the number of parameters that the model has to learn. Taking the paper's definition of d (same as ours), $|X|$(the size of *value_vocab*),

$|P|$(the size of the *path_vocab*), and $|Y|$ (the size of the *tag_vocab* i.e. the number of labels), we have the number of parameters to be $O(d * (|X| + |P| + |Y|))$.

The results of this model (measured by F1, precision and recall values), were compared with other well known techniques in this particular field. Table 3 shows the best of this comparison, taken directly from the paper. One should take special note in prediction rate of 1000 methods per second, when compared to 10 per second in the next fastest model (Path+CRFS).

Model	Sampled Test Set (7454 methods) Precision	Recall	F1	Full Test Set (413915 methods) Precision	Recall	F1	prediction rate (examples / sec)
CNN+Attention [Allamanis et al. 2016]	47.3	29.4	33.9	-	-	-	0.1
LSTM+Attention [Iyer et al. 2016]	27.5	21.5	24.1	33.7	22.0	26.6	5
Paths+CRFs [Alon et al. 2018]	-	-	-	53.6	46.6	49.9	10
PathAttention (this work)	**63.3**	**56.2**	**59.5**	**63.1**	**54.4**	**58.4**	**1000**

Table 3. Evaluation comparison between our model and previous works.

As a conclusion, particular interests point that stand out are as follows:

- Why would they define AST paths from terminal to terminal? In particular, could there be a better path representation of the AST, say from the root to the terminal?
- Is there another representation of code that is as universal as the AST that could be used, with potentially better results? This would lead to an investigation in compilers.

2.3 code2seq[3]

Code2seq is along the same idea of code2vec. Which traditional code summarization (essentially commenting) techniques define code to be a sequence of tokens, an improper model as we've seen in code2vec, code2seq leverages the the concept of AST to better model code for machine understanding. Doing so, as expected, improves the generated summary quality for a code snippet. Better data representations do indeed have better results.

Notably, a few important modifications in the architecture arise from the fact that code2seq is essentially a seq2seq problem (in a crude sense). seq2seq problems are usually

solved by Neural Machine Translation(NMT) methods. The use of LSTMS for encoding and decoding is heavily emphasized in such techniques.

A path-context is, like in code2vec, two terminal values and the path from one terminal to another. The terminal values are tokens in the code, and they are split into subtokens, to which a embedding is drawn from an embedding matrix $E^{subtokens}$. Camelcase especially helps in deciding subtoken splits. The encoded value of the terminal value is the sum of all its subtokens. A path however, is represented by the final states in a bidirectional lstm, whose input is the sequence of embeddings of each node in the path, from E^{nodes} of the path nodes. These three encodings are then compressed into a lower dimension encoding similar to code2vec. The decoder generates the output sequence while attending over all the context path encodings, to which the attention weight to be given to each encoding is calculated from an attention vector. The attention vector is generated for each timestamp of the decoding LSTM, and is dependent on the state of the decoder and the input path context encodings. Other notions are similar to code2vec.

The use of an LSTM in path encoding is useful in generalizing over different unknown paths, and reduced the required vocubulary size to do so. Here, we just need the embeddings of the nodes(linear space in number of nodes), whereas in code2vec, we need each embedding for each path combination to be stored (exponential space in number of nodes). Doing so is more efficient, as this LSTM based approach proved to be a better solution in label generation as well, likely due to this generalization feature of the LSTM path encodings.

2.4 node2vec[4]

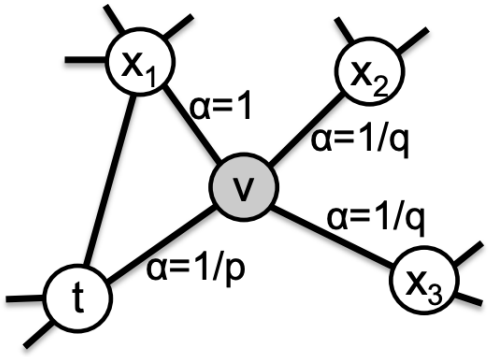
Network analysis often require predictions that are based on edges/nodes of the graph, like link prediction, prediction of users with specific interests, or clustering tasks.

In this work, authors propose a novel method to generate continuous vector representations of nodes in a graph, that preserve their local neighbourhood qualities through *learning* the features. Node2vec is a semi-supervised method, and the key contribution of this work

is the way they capture the notion of neighbourhoods. This is done through a set of random biased walks, which efficiently captures the neighbourhood for a given node. The notion of node embeddings can be easily extended to edge embeddings by using the hadamard operator (or other methods).

To achieve this, the authors have formulated the problem of feature learning into a maximum likelihood optimisation problem, by extending the skip-gram architecture to graphs. They've defined an objective function which tries to maximise the probability of observing the neighbourhood nodes for a given node, given its feature representation (Under certain assumptions).

$$\max_{f} \quad \sum_{u \in V} \log Pr(N_S(u)|f(u)).$$



To generate these walks, we have the two extremes in the way we sample the walks. BFS and DFS. BFS tend to capture the structural qualities better, and DFS on the other hand explore a more broader (macro) view of the graph. The biased random walks used in node2vec are an interpolation of the pure BFS/DFS extremes. The two parameters 'p' (return parameter) and 'q' (In-Out parameter) control the nature of the traversal.

Chapter 3

CodeSearchNet Challenge

Deep-Learning has revolutionised how we approach speech and image recognition tasks, but these approaches still struggle with highly-structured data. One such task is to find relevant pieces of code from natural language queries (semantic code search).

A major bottleneck while evaluating methods for this task is the unavailability of a common dataset, and everyone seems to be working on their own curated small datasets created by scraping websites like stackoverflow for queries and relevant code snippets.

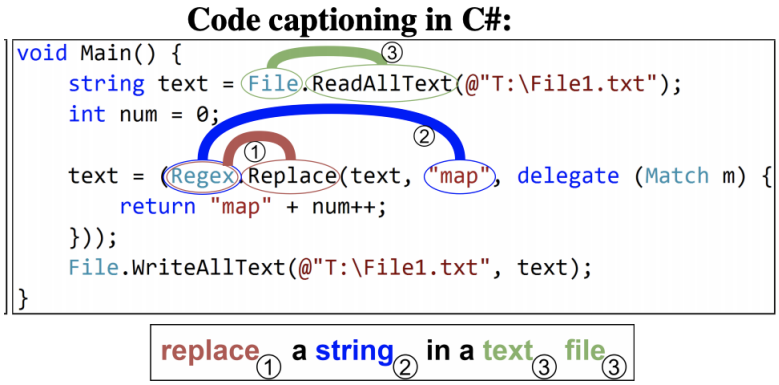
Github, along with Microsoft research introduced the codesearchnet challenge to address these problems by creating the codesearchnet corpus by extracting code with their metadata(comments,and documentation) from open-source projects. This includes over 6M methods, 2M methods of which have their associated metadata available. This challenge is focussed on providing a platform for evaluating code search results on the basis of natural language queries. A gold standard of 99 queries is given, to each of which it is expected that a challenge submission will order 1000 unknown functions. The relevance of the ordering of the functions is calculated with NCDG, and the scores are put up in a leaderboard, meant to encourage to people to study this interesting task. The filtered dataset, and specific instructions on how to use the dataset was made available at CodeSearchNet.

In the following sections, we'll be describing the possible approaches we're going to

be implementing as the next step in our BTP, for the above challenge. These are heavily inspired from our study of continuous vector representations of code (Code2Vec, Code2Seq).

3.1 Approach I

A very trivial extension of the **code2seq**[3] model for this problem would be to use code2seq for the task of *"code captioning"*. An example of code captioning is shown below.



After converting all the methods in the code corpus to their corresponding natural language caption/summary, the problem of retrieving relevant code snippet reduces to a much simpler, and well studied problem of *"relevant document search"*.

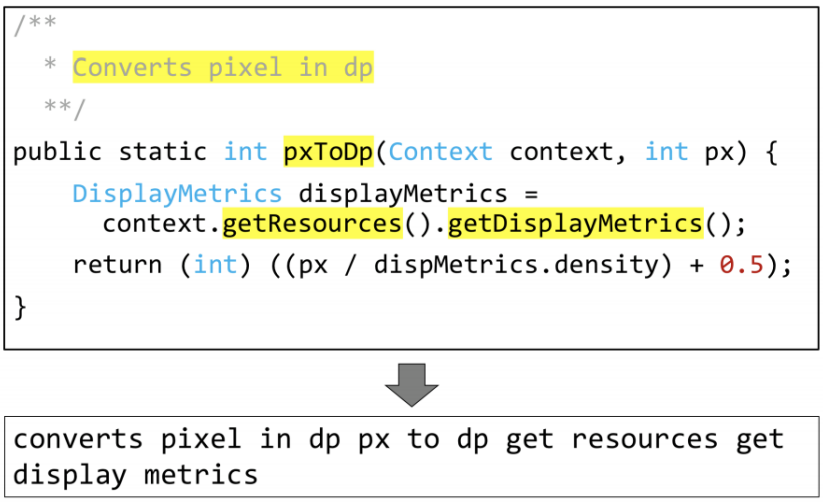
We don't have very high expectations from this approach as we suspect a loss of a lot of information during the code-captioning step. This will then directly degrade the quality of document search. However, it a good first step in understanding the potential of the Code2seq model.

3.2 Approach II

During our research, we stumbled across an interesting approach for this task, proposed by the Facebook AI research team[5].

They take a somewhat different approach to this problem, by mapping code snippets and queries to the same high dimensional vector space, and then calculating relevance of a given query q, to a code snippet c, by vector distance between their representations.

This is done by first converting methods to documents simply by concatenating words in method name, class name, method invocations etc. An example is shown below.



A variant of word2vec (FastText) is used to obtain word embeddings, and these words (in the document obtained above) are combined by taking a weighted average (using TF-IDF) to map this document to a vector space.

Note that the vector space is in the word domain. Queries are mapped to the vector space similarly and vector distances are used to find similarities between the query and code snippets. Relevant snippets are extracted by using closest cosine distances. Authors have used FAISS for this in their implementation.

We see that there is *hope of improvement* to this method, as we think that the way code is converted into documents is very naïve, in the sense that it only exploits syntactic properties of the code. If we use code2seq, we can exploit **structural** information to extract semantic information contained in the code through the use of ASTs.

We hope to use this idea to improve the quality of code embeddings in this method, while still mapping code and queries to the same high dimensional vector space.

In the case that mapping them to the same space is not possible (the output of Code2seq's variant Code2vec is a code vector), we can try to learn a mapping from the code to word domain.

Chapter 4

Building the solutions

The challenge of then building a semantic code search engine is can be broken down into the following tasks:

1. Building a reliable representation of the code snippets
2. Textual Analysis of the query string and indexing the code snippets(functions)
3. Defining a similarity metric that relates the query string to the index

This chapter discusses the solutions to the above problem in depth. We've explained the building of Approach 1 as discussed in the previous chapter. Unfortunately, Approach 2 is not viable with the the introduction of the Covid19 lockdown, mainly due to hardware limitations. A clearer explanation of difficulties faced can be found in 6. The solution approach of approach 2 is reiterated in 5.

4.1 Building a semantic code search engine Using Code Captioning of Code2seq (Approach 1)

The idea here is to use Code2seq to caption all functions in the CodeSearchNet dataset and then using the captions as well as the docstring of each function as a field named "full_doc" to index the file in an ElasticSearch instance. Essentially, each function is reduced into a

document, reducing our problem to document search. We then query on the field "full_doc", where the query similarity is calculated by the default TF-IDF of the ElasticSearch Engine.

4.1.1 Introduction to relevant tools

Code2seq

Code2seq is a machine learning model that has proved to be exceptional in the task of captioning code snippets. The paper, accepted in ICLR 2019, was built off of idea of using AST (Abstract Syntax Tree) paths to represent code snippets, previously introduced by Code2vec. The encoding architecture has been shown to be exceptional in the task of code summarization, where a method name was predicted given the body of a method and in code captioning. The paper demonstrated the model's superior encoding strategy with compared with previously dominant models on a standard dataset(CodeNN (Iyer et al., 2016).

The architecture of Code2seq accepts a function, and then produces the prediction of it's relevant caption. It can thus be used to generate captions for all the functions in the dataset of CodeSearchNet.

CodeSearchNet

The CodeSearchNet dataset is present in *jsonlines*, however they have provided the necessary scripts to generate the relevant pandas dataframe. The simplest way to understand the *jsonline* is as a text document where each line is a json string. It can be read as a python dictionary as it is simply a textual (human readable) representation of key value pairs. The dataset is split based on languages, and for each language into train, valid and test sets. They have also for each function, provided certain helpful fields. The relevant field are, as taken from CodeSearchNet Readme (Schema and Format):

- **original_string:** the raw string before tokenization or parsing

- **func_name:** the function or method name
- **code:** the part of the original_string that is code
- **code_tokens:** tokenized version of code
- **docstring:** the top-level comment or docstring, if it exists in the original string
- **docstring_tokens:** tokenized version of docstring

A special mention is that CodeSearchNet took care to make sure all relevant code is open source and can be used under that specific Github Repository's license. Information regarding the repository is given in the fields described below:

- **repo:** the owner/repo
- **path:** the full path to the original file
- **url:** the url for the code snippet including the line numbers

ElasticSearch

ElasticSearch is a scalable, reliable and well maintained open source search engine that can search on documents. We've selected it to search on relevant documents since it is scalable, it uses HTTP requests to build and make its index (thus allowing it to be easily interfaced with a website), and because it a good solution to the document search problem. ElasticSearch organizes by defining *index*, *type* and *id* fields. The *index* is the unique namespace we assign to the data we insert into elasticsearch. Its best to think of it as a folder. The index contains all the documents/data one would like to run a relevant search on. *Type* refers to the document type, for our case we treat it as a "_doc" type, referring to standard document search. *Id* refers to the unique identifier that we assign to each document in an index.

ElasticSearch also has well defined APIs that can be used to define or set certain query parsing algorithms as the default. The same extends to calculating the relevance score of a function. Relevant functions will be discussed in the following sections.

4.1.2 Important pipeline notes and Terminology

Code2seq offers support in two languages, Java and C++, however, since the CodeSearchNet dataset only offers code in the languages Java, Python, Javascript, Go, Ruby and Php. It doesn't have a C++ repository, and thus we have focused our implementation in Java. Since training Code2seq is rather difficult in the current scenario, a pretrained model of Code2seq was used to generate the Code snippets. Code2seq requires a preprocessing function that cannot be easily run on a personal computer, for this reason we've built the search engine only for the valid partition of the java data files of the CodeSearchNet dataset.
As mentioned before, CodeSearchNet data in raw form is a collection of jsonl(jsonline) files. ElasticSearch expects input in **jsonline** format, and expects HTTP GET or POST requests. Therefore, each function is fed into an index as a jsonline, which is essentially its jsonline description drawn from the CodeSearchNet raw dataset.

4.1.3 Building the Architecture

Using Code2seq

The pretrained model of Code2seq was trained for 52 epochs and is the model that is used to generate the predictions on the code2seq site, where 10 sample and their predictions are demonstrated. We made a folder that contains all the Code2seq data, split based on the test, valid and train partitions. A file is named based on its partition and a unique id. For example, "train_582.java" refers to the 582nd row of the train partition of the CodeSearchNet java dataframe. The dataframe is stationary; its rows don't change position once created. We used this pretrained model of Code2seq to generate a log file that contains a method name and its corresponding prediction (i.e. its caption/summarization).

Analysis of Output Log file

Now, using that log file, we build of dictionary that captures method names as keys and the predictions as the corresponding values. Not that in the case of commonly occurring method names, we may of many values of the predicted caption. This is because a prediction is generated from the code body, but is mapped to the code's method name. Identically named methods (for example in the case of constructors or methods implementing inheritance) may have different functions especially in the case of constructors. Therefore, the relationship from methods to captions is actually one to many. In such a case, we take the first caption assigned to method name. Statistically, of the given 15328 code samples, Code2seq has detected 16660 methods, and therefore the log file has 16660 (method, caption) pairs. The number of functions that occur more than once make up about 5K samples, with a the maximum number of occurrences being 82 times. However, such an occurrence is an rare since most method names don't occur more than 10 times.

Building the index

Since each function is represented as a *jsonline*, we define a new field(dictionary key) in each such jsonline (dictionary) named as *func_name_prediction* which is that function's caption. Also, we define another field called *full_doc* defined as:

$$full\_doc = func\_name\_prediction + func\_name + docstring\_tokens + code\_tokens$$

where + is the binary string concatenation opeation. We then insert this document into the an ElasticSearch index (which has been named "valid" since the predictions were only built on the valid partition of CodeSearchNet's java data. The insertion was done with ElasticSearch's Bulk API, with the index set as "valid", type as "_doc" and the id is extracted from the file name as defined in 4.1.3.

Querying the Index

We can query in a number of ways as defined by the various text querying APIs. Special mention is the simple query string query. In the particular querying paradigm i.e. "simple query string query", a text analyzer is used to convert a query string into a set of tokens, and each token is then checked for certain special symbols given as a syntax. "Simple text query" also allows for assigning weights to fields, thus defining importance among fields that are being queried. Currently we are searching on the *full_doc* field of each jsonl(*jsonline*) document. There is no sense of weight here since we query on only one field.

Calculating a Relevance Score

ElasticSearch implements Lucene, from which it uses the Practical Scoring Function to calculate the relevance score of some document to a query. The exact implementation is discussed in this article. For a better understanding, as taken directly from the previously referenced article:

"For multiterm queries, Lucene takes the Boolean model, TF/IDF, and the vector space model and combines them in a single efficient package that collects matching documents and scores them as it goes."

Loosely put, the boolean model first verifies if a document (field) has a term from the query, and then only allows for calculating that terms relevance score. Each relevant document (field) is then represented as a giant vector, where each entry in that vector is a function on that term's TF(term frequency, number of occurences of a term in that particular document) and IDF(Inverse of the number of documents a term has appeared in). Vector space model simply refers to the idea of representing a document as this vector. The query is also processed to be represented in a similar way. The take home message is the search query is parsed token wise, and so is the set of documents.

A HTTP GET/POST query now returns the top 10 relevant documents as a json element, which can easily be read as a document and parsed for relevant fields.

4.1.4 Analyzing this Solution and Limitations

The hope in this approach is that the output of Code2seq can further boost the relevance of an important code snippet for a particular query. However, by focusing in the word domain (after all the output of Code2seq is a set of words, a sentence) we have potentially lost critical information generated by Code2seq(the code vector!). This problem is is goal of approach 2.

Code2seq's output has the problem of repeated method names, which loses reliability on generating a one to one mapping from method name to prediction when building the search index. Each code snippet needs to be further preprocessed in a way that it can generate unique method names. A suggestion is using the "func_name" field, which gives the folder and file hierarchy of each function, however a lexical analyser sort of approach would be needed to find where this well defined method name should be inserted into a code snippet. Unfortunately, this still doesn't answer the problem of functions that demonstrate inheritance, as they will have the same folder and file hierarchy.

Additionally, defining weights for different fields in a document and to the query tokens (not possible under simple-query-string paradigm, but under Query-string) depends heavily on the performance of the search engine (they can be optimized as hyper parameters). The performance of the search results can be quantified by submitting a run on the CodeSearchNet site, however, the results generated on an search space of 18k functions, when evaluated on 500K dataset, will be very uninspiring. This problem is further explored in the following section.

4.1.5 The very Real Problem of Defining a Good Search Result

The CodeSearchNet problem generated NDCG scores for a given set of search results. This score is heavily dependent on the search space, and for the challenge has defined two search spaces, one being within the set of all java functions with code documentation (the 500K set), and the set of all java functions (2 Million set). With a sample space of 18K(the current

build due to system limitations), the NDCG score will be very unreliable in defining a good Search result. Only with a larger scale will one be able to statistically test the effectiveness of a search result. For this reason, the NDCG scores have been neglected on the current implementation.

Chapter 5

Alternation Directions to Explore

There were a few Ideas that we encountered along the way while our literature survey and weekly meetings that we wanted to explore but couldn't due to a variety of reasons.
We're documenting these ideas for future batches to build upon our work using these as starting points. The sections haven't been ordered with any priority, but in general earlier sections are more reliable research problems.

5.1 Code2vec and USE: Learning a cross-domain embedding mapping

This is essentially the second approach illustrated in 3.2. The primary problem in code search using Natural language queries is that they are in different domains. Facebook AI solves this by treating code as a collection of words. We have already pointed out the problem in doing that in multiple instances. Code2vec creates Vector embeddings for code snippets, while we can use techniques like word2vec (or more advanced like USE) to generate sentence embeddings.

There have been previous attempts to learn such mappings (see this).

We wanted to try and train a neural network to directly learn this mapping. For this we planned to use comments (and metadata available) as sentences while training and learn

mapping to corresponding code embeddings. While at test time, treat each query as a sentence and predict a code embedding corresponding to this sentence. Now use something like Nearest neighbour to get the most relevant code snippets.

5.2 CBOW on code2vec

The target here is the use the intuition of CBOW to further improve the quality of code vectors generated by Code2vec. The prime motivation is that CBOW is a very powerful and quick model, so if a parallel can be drawn, as was the primary intention of the starting of this BTP project, we can now have a model that is rediculously accurate (as Code2vec) and much faster. Code2vec is a quick model itself, but CBOW is a much better standard. Code2vec takes context paths from a code's AST, thus reducing a code snippet to a bag of context paths. Each context path is then represented by a fixed size vector, referred as a context vector, thus a code snippet is now treated as a bag of words. Recall that a context path is a node to node path in the AST. code2vec uses a large dictionary like data structure is used to reference each individual context path to a vector. No attempt at checking for path similarity among the set of paths to better improve context path vector representations has been made. In code2seq however, each path is encoded with a bidirectional LSTM, thus allowing for the learning of path similarities.

Now, consider the following approach. Suppose we come up with a way of ordering the context paths such that we have similar paths located near each other. Now, treating each context path as a word, a code snippet can be treated as large sentence. We can now compose a very large paragraph that contains all code snippets. On this document, we can run word2vec, to learn each context path's vector representation.

We now require to order the context paths meaningfully. Suppose we order all leaves in the AST, from left to right (based on this simple programming problem). Recall that context paths are represented by terminal leaf nodes l_a and l_b and the unique path between them p_{ab} as $< l_a, p_{ab}, l_b >$.

A context path $< l_i, p_{ij}, l_j >$ where $i! = j$ (since it doesn't make sense to have paths in a Tree starting and ending with the same leaf node) is defined to be ordered before another path $< l_m, p_{mn}, l_n >$ where $m! = n$ iff one or more of the following conditions hold true:

- $i < m$
- $i = m$ and $j < n$

Now, paths that are next to reach other have a very high probability that some part of their leaf to leaf paths (p_ij and p_mn) overlap. We can define a sentence to be a ordering of paths that all start from the same node. The hope is this data preprocessing makes it easier for the model to reach optimal code representations, thus reducing training time while not sacrificing accuracy. Other path orderings can be explored as in the next section.

5.3 Assigning similarity between context paths

Instead of context paths, what if represented a code function as a set of root to leaf paths? Arguments can be made that leaf to leaf interactions will be lost, but remember, having a common path will signify some similarity among context paths. Under this paradigm, we now have another metric to define similarity between paths (length of common prefix, where nodes in the common prefix can be weighted). Some ordering similar to the previous section can be made. Hopefully this results in better code representations.

5.4 Using graph2vec, node2vec to improve code embeddings

The main paper when talking about code embeddings is code2Vec which uses ASTs to generate paths and feeds it to the network to generate embeddings.

We argue that these naive Paths might not be able to capture the code structure as efficiently as graph embedding techniques like graph2vec and node2vec might. AST's

are also graphs and we can try to use graph2vec and node2vec to improve directly upon code2vec.

5.5 Leverage Comment location

An important information no one is using is the comment location. Not all comments are about the entire method and written on top of the function. Some comments are above a portion of code and that comment can give an indication of what's happening inside that particular section only. This we believe could be a potentially rewarding research to dive into. This will definitely increase the quality of embeddings and understanding of code snippets.

5.6 Faster code2vec

The model is really heavy. Is there something we can do to reduce the training time and remove some redundancies? this is not exactly a pointer to start with but a direction you can possibly think towards. This lies in parallel with the ideas in the section on CBOW on code2vec.

5.7 Clustering coders or projects by code

The idea that code in the CodesearchNet dataset contains references to its origin repository inspired this. This is a vague idea but the idea is using code embeddings to make a linkedin type platform where we search for relevant people by giving sample code that engineer will have to write, and engineers are clustered by the type of code they write. This would be far more effective than current methods of head hunting, and will be automated.

5.8 Code2seq as an Antivirus

One of the more unique applications mentioned in Code2vec was the potential of using code vectors to detect potentially harmful programs. Current anti-virus software use various code similarity techniques as mentioned in this article here. Code2seq and code2vec can be better used to represent code, thus opening up the idea that better antivirus software can be made.

Chapter 6

Problems faced & Advice

This section illustrates the problems and potential solutions we faced in the development phase of this project. We first describe the development environment and then for each, list out problems and solutions. It has been mainly written in the view of students of the next batch that might attempt this project.

6.1 Environment

There are a few things you need to learn before starting this project. IITG CSE provides GPU access in two ways, one with Kubernetes and the other with individual GPU access. Individual GPU access requires self setup; they don't have any prior packages, libraries, or languages in them. The prime advantage is that it is immediately accessible. Kubernetes on the other hand is much easier to set up, since it uses Docker, however, is very limited in accessibility.

Your decision on getting an account should be heavily dependent on these features. Also, get storage of around half terabyte. Code2vec's data is around 30 gb, however Code2seq deals with data sized around 130 gb. Space isn't an issue with the CSE department.

Kubernetes

Kubernetes is a container management system. This github repository is a good resource. The GPUs we use, we don't get sudo permissions on it, rightfully so to avoid misuse. Primarily, any GPU request is made in a pod. First, a pod needs to be created with:

```
kubectl create -f pod_specifications.yaml
```

where the yaml specifications will describe your account's access codes and the docker file to load. Kubernetes access gives a login node that only has Ubuntu 18.04 installed, none else. Packages can't be installed or managed. This leads to the problem of creating our custom environment with specific dependencies. This brings us to docker.

Docker

Docker is a way of specifying the requirements to be installed on a container. It is important to learn to host it and use it in your pod. You can specify what libraries and support you want in a dockerfile, execute it on your system to create the file, and the host it on Docker hub to be accessed from anywhere on the internet. The yaml file has a field where you would point to the docker file location. Not that it points to Docker hub, and not Github. Docker does not have support on Windows home edition (the one on standard pcs). It's support was introduced for linux in Ubuntu 18.04. It is easier to use on the Mac. Refer this for installation instructions. Recently, an update to WSL(windows subsystem for linux) as WSL 2 allows for installing docker, however, it will be released on May 28th for public users. Refer here for WSL installation and here for docker installation on WSL 2.
Our current docker hub image can be found here.

Other critical information

Kubernetes, as mention earlier does not give root access. When a pod runs, it copies all data in a login node into a folder named "/data". Note that it is the Absolute path. If

you attempt to run programs without /data added to the path, by default they will fail. The current Code2seq implementation hosted on the "Yagyansh" login node has fixed these issues, but be warned, the github downloaded Code2seq or Code2vec repos will not work.

6.2 The Challenge of Availability

First and foremost, as everyone is sharing the same set of GPUs they are rarely available. Sometimes, it took a week before a job started running from the day it was scheduled. Kubernetes for this reason is very tedious to work with, especially for first time users of Docker, and for handling paths.

Try to get a seperate GPU for this reason(with sudo access), while it might be difficult to install, it will be better to use than Kubernetes. Another idea would be to try to get a cluster for our panel: Students under Prof. Ashish Anand, Prof. A. Sahu and Prof. Amit Awekar.

6.3 Connectivity Issues

With the new internet pipeline, to use internet you will need to log in to agnigargh on the GPU (unless someone has already). For this we recommend using Lynx (command line utility browser), but Lynx is a static browser and can't refresh itself. So Don't expect it to run more than 10 minutes. You might have to manually refresh it by running it again. You can also try running a browser on your PC through ssh on login node.

For longer downloads, we suggest downloading it locally on your PC and then use scp/ftp it to the GPU.

Use 'Screen' to keep the connection alive even when you get disconnected. Screen by default isn't installed, try to include it in your docker image.

6.4 Accountability

There needs to be a system to keep track of your progress and staying consistent. Sir will schedule weekly meetings to discuss that week's progress and steps to be taken till next week. Fix a time for weekly meetings after thinking about it and stick to it.

We also had a shared notebook with sir, where we logged everything we did, any progress we made, any ideas we brainstormed, and any issues we were facing and are stuck due to.

This will keep you accountable to your BTP, and sir can keep track of your consistent progress through this log whenever he wants to. Discuss all of those then in the weekly meetings.

6.5 A final note

Regardless of these challenges, it was fun to see and use these tools. Many tools that we've learned are quite relevant in industry, and are usually a standard in their respective domains. Times will get tough, but keep your chin up and keep your focus on the goal.

References

[1] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. 2013.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code, 2018.

[3] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code, 2018.

[4] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

[5] Hongyu Li Sonia Kim and Satish Chandra. Neural Code Search: ML-based code search using natural language queries. https://ai.facebook.com/blog/neural-code-search-ml-based-code-search-using-natural-language-queries/, 2019.

[6] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search, 2019.

Note We have added reference to only papers and articles used while making decisions like word2vec[1] and codesearchnet[6], None of the references related to tools we used like ElasticSearch, docker, kubernetes are added.