Semantic code search using Code2Vec: A bag-of-paths model

by

Lakshmanan Arumugam

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2020

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The world is moving towards an age centered around digital artifacts created by individuals, not only are the digital artifacts being created at an alarming rate, also the software to manage such artifacts is increasing than ever. Majority of any software is infused with large number of source code files. Therefore, code search has become an intrinsic part of software development process today and the universe of source code is only growing. Although, there are many general purpose search engines such as Google, Bing and other web search engines that are used for code search, such search engines are not dedicated only for software code search. Moreover, keyword based search may not return relevant documents when the search keyword is not present in the candidate documents. And, it does not take into account the semantic and syntactic properties of software artifacts such as source code. Semantic search (in the context of software engineering) is an emerging area of research that explores the efficiency of searching a code base using natural language queries. In this thesis, we aim to provide developers with the ability to locate source code blocks/snippets through semantic search that is built using neural models.

Neural models are capable of representing natural language using vectors that have been shown to carry semantic meanings and are being used in various NLP tasks. Specifically, we want to use Code2Vec, a model that learns distributed representations of source code called code embeddings, to evaluate its performance against the task of semantically searching code snippets. The main idea behind using Code2Vec is that source code is structurally different from natural language and a model that uses the syntactic nature of source code can be helpful in learning the semantic properties. We pair Code2Vec with other neural models that represents natural language through vectors to create a hybrid model that outperforms previous benchmark baseline models developed in the CodeSearchNet challenge. We also studied the impact of various metatdata (such as popularity of the repository, code snippet token length etc.,) on the retrieved code snippets with respect to its relevance.

Acknowledgements

I would like to thank Meiyappan Nagappan for being my advisor and mentor during my Master's. I also thank the readers Michael Godfrey and Yaoliang Yu along with other members of the Software Architecture group at the University of Waterloo. I must also acknowledge other researchers from various publications, this work would not have been possible without their contributions.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Whenever a developer wants to search for any software related artifact (Source code, documentation, and other artifacts), they tend to rely on general-purpose search engines that are not optimized for searching source code. Prior work done by Rahman et al., [32] evaluated how developers use general-purpose web search for code retrieval. In their work, they build a classifier to identify the code intent of a given query which finds the characteristics of a query. They found that queries vary from code to non-code queries. More often than not, code related search takes more effort in terms of time and query modification. They also study how general-purpose search engines could be effectively used for code search.

Instead of improving the ways to use general-purpose engines for code search, we suggest to use semantic search, which attempts to identify the searcher's intent and contextual meaning of terms to retrieve more relevant results. Moreover, code search for source code (and other artifacts such as packages, libraries) is identified as a developer problem. There are StackOverflow [29, 12] discussions asking for an alternative for Google code search, a regular expression based index and search tool for large bodies of source code, which was previously started (2006) and shutdown (2011) by Google [46] as developers started moving towards GitHub. We also believe that searching over source code is a problem worth studying.

In this thesis, we want to apply deep learning based neural models to search for code snippets as such models have been shown to understand meaning [3] behind source code. Deep learning has been successfully applied to many natural language processing problems in the last few years. Researchers have found neural models to be effective in learning semantic properties of natural language text [3, 33]. Although code is not natural language, we believe neural models are capable of learning semantics as well as syntactic properties

available in code snippets. Specifically, we focus on applying a path-based neural model called Code2Vec [3] created by Alon et al. This model treats code snippets as bag-of-paths and its development was inspired by Word2Vec [33] (from Mikolov et al.), a natural language model that treats natural language as a continuous bag of words or tokens. We have investigated if path-based neural representation like Code2Vec is useful for semantic search. We have also investigated if project metadata is useful for reranking search results.

The main contributions of this thesis are,

- Adapting Code2Vec in order to build a neural model for semantic code search.
- Evaluating the impact of metadata such as popularity of the project, code token length etc., on the relevance of retrieved code snippets for reranking.
- Migrating CodeSearchNet [19] benchmark neural models in Tensorflow 2.0
- Implementing Code2Vec [3] into CodeSearchNet and evaluating its performance for the task of semantic code search.

The thesis is divided into following chapters. In chapter 2, relevant literature needed to create this thesis is discussed. In chapter 3, the methodology, data origins and preprocessing are explained. In chapters 4 and 5, results and implications are discussed. Finally, we conclude our thesis by summarizing our contributions and discussing potential future work.

Chapter 2

Background and Related Work

2.1 Background

Our thesis is a follow up research to two of the previous works: Code2Vec [3] and CodeSearchNet [19] which have been detailed below. We evaluated the application of Code2Vec to do semantic search and compared the results with other baseline models which were developed in the CodeSearchNet challenge. We then investigated if a project's metadata is useful in reranking the retrieved snippets in terms of their relevance.

2.2 Code2Vec

Alon et al. have recently proposed an interesting approach by learning distributed representation of code[3] similar to word2vec. In their work, they create a neural model for representing snippets of code as continuous distributed vectors "code embeddings") that would predict semantic representations of a snippet. They perform this by decomposing code to a collection of paths in its abstract syntax tree, and learning the atomic representation of each path. The authors make use of embedding similarity of similar code to predict method names. They evaluate their approach by training a model on a dataset of 14M methods and show that the model can predict method names from files that were completely unobserved during training. The model is trained on a dataset of 12 million Java methods and compared to other competitive approaches. It significantly outperforms them by having approximately 100 times faster prediction rate at the same time having a better F1 score of 59.5 at method naming. While they only tested their model for method

naming, the authors believe that there are a plethora of programming language processing tasks the model can be used for such as semantic search.

Prior to Code2Vec, the same authors devised an approach to predict program properties from a General path-based representation for code which led to Code2Vec. In this earlier work, they showed this approach obtains better results than task-specific handcrafted representations across different tasks and programming languages. To extract path based representations for different programming languages, Kovalenko et al. created a library called PathMiner [22].

2.3 CodeSearchNet Challenge

Researchers from GitHub have created an online code search challenge named CodeSearchNet [19] where anyone can build and submit results from a neural based code search engine. They have publicly released their training dataset which consists of code snippet-documentation pairs for six different programming languages: Go, Java, JavaScript, PHP, Python, and Ruby. A total of six million functions, of them 2 million functions have query-like natural language documentation. In order to evaluate the progress of semantic code search, the authors have also provided 99 search queries taken from the Bing search engine. These queries have been manually annotated by developers for relevant code snippets with a relevance scale of 0 to 3. CodeSearchNet also provides a benchmark of six different neural models that serves as a baseline for subsequent submissions.

2.4 Related Work

Researchers from Google have also investigated and performed a case-study on how developers search for code [34]. Their study indicates that developers search for code more often (12 queries per workday) and search patterns also varies with context. Search queries are often aimed at locating a particular location of code and seek answers for how to use an API, what a piece of code does and why something fails. They also used the search logs to explore the properties of a typical search query and series of queries with no result clicks between. The key observations of the developer behaviour include, 1.) Developers search very often and they search for examples more 2.) Developers search locally (as a quick navigation of the local codebase), and 3.) Search queries are incrementally redefined.

2.4.1 Deep learning for source code

Gu et al. [17] performed a similar study where they described a deep neural network (CODEnn) for searching code in which they demonstrate a proof of concept application of the model and evaluate the results against traditional IR-based approaches Lucene and CodeHow. The results generated by CODEnn are substantially better than IR-based approaches. There is also an ongoing experiment where engineers at GitHub tried to implement semantic search using neural models for code using code documentation and methods.

Hu et al. [18] performed a study to generate automatic comments or documentation for java methods. They propose a new approach named DeepCom that applies natural language processing techniques to learn from a large code corpus and creates comments from learned features. Their approach also uses neural networks to analyze structural information of Java methods for better comments generation. Their experiments were done on a large scale by preparing the corpus from 9,714 open source projects from GitHub and they evaluated their method based on a machine translation metric.

Yin et al. [50] have presented a novel approach to mine StackOverflow data to produce high quality aligned data: code and natural language pairs. Their approach uses two kinds of features: hand-crafted features using the structure of extracted features and features obtained by training a probabilistic model that apprehends the correlation between natural language and code using neural networks. This mined dataset called as CoNaLa can now be used as training data for models in downstream tasks such as code summarization and retrieval. Moreover, the authors have open-sourced benchmark sets to test and improve such models.

Briem et al. [5] have used Code2Vec's distributed representation of code in their work to evaluate its performance for bug detection, specifically, detecting off-by-one errors in Java source code. They treat bug detection as a binary classification problem and train their model on a large Java file corpus containing likely correct code. To properly classify incorrect code, during model training, they use false examples which are created by making simple mutations to correct code from original corpus. Finally, their quantitative and qualitative evaluations validates that the model which uses a structural representation of code can be applied to tasks other than method name prediction.

Allamanis et al. [1] have devised a new way to represent software programs with graphs. Their solution utilizes long-range dependencies induced by using the same variable or function in distant locations inside a program. They propose graphs to handle both syntactic and semantic properties of code and use graph-based deep learning methods to

learn to reason over program structures. Typically, graphs tend to be larger in sizes and the authors show how to scale Gated Graph Neural Networks training to such large graphs. Moreover, they evaluate their approach to two tasks: VARNAMING, in which the model attempt to predict the variable names when the usage is given, and VARMISUSE, in which the model learns to reason about selecting the correct variable that should be used at a given program location. Their results suggests that the neural network learns to infer meaningful names and solves VARMISUSE in many cases. After testing, they claim that VARMISUSE even discovered bugs in Roslyn [25], a mature open source project.

2.4.2 Machine learning on reranking code snippets

Niu et al. proposed a code example search [30] approach that applies machine learning techniques to automatically train a ranking schema. They apply this trained ranking schema to rerank candidate code snippets for new queries at run time. The evaluation of the ranking performance of their approach is done using a corpus of over 360,000 code snippets crawled from 586 open-source Android projects. The results from the performance evaluation study showed that the learning-to-rank approach can effectively rank code examples, and outperformed the existing ranking schemas by about 35.65% and 48.42% in terms of normalized discounted cumulative gain (NDCG) and expected reciprocal rank (ERR) measures respectively. Their work is one of the very few works that showed reranking code snippets can be an effective approach to code search.

Recently, the application of machine learning or deep-learning-based models to solve research problems in software engineering is increasing [2, 27, 23]. Thus utilizing such state-of the art models and approaches in the context of semantic search opens up new research avenues. As semantic code search is also one of the downstream tasks, it could be leveraged as an evaluation task for such advancements.

Chapter 3

Methodology

In this thesis, we focused on performing semantic search in source code through a bag-of-paths neural model (Code2Vec) and evaluating the model against the benchmark models developed in the CodeSearchNet paper [19]. The general methodology behind semantic code search is representing code features in the same vector space as natural language features and then creating vector indexes to perform similarity search illustrated in figure 3.1. We divided the study into two parts: An initial feasibility study and as CodeSearchNet challenge. The feasibility study was performed to understand the process of applying deep learning models to code search. More details about this initial study are described in section 3.2.

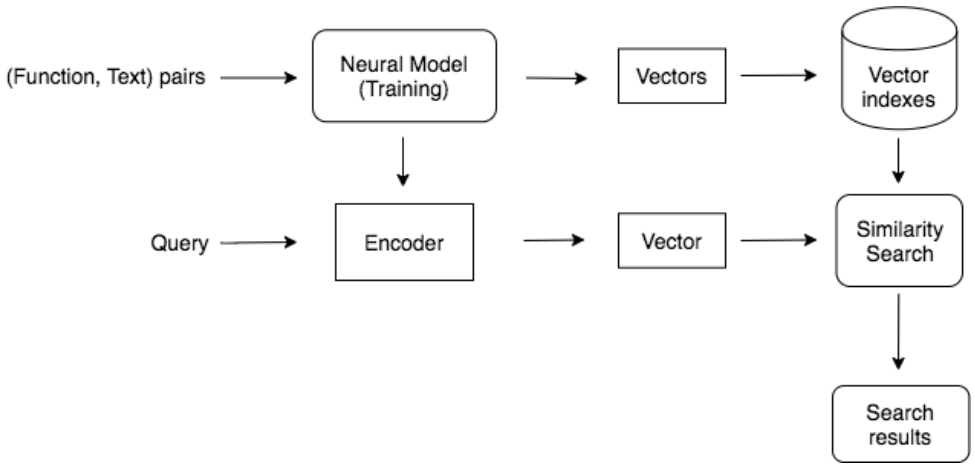

Figure 3.1: Process Overview

After the feasibility study, we approached the CodeSearchNet challenge in two different parts. First, we treated code search as an information retrieval problem and applied deep learning models to retrieve a candidate result set from a huge corpus. Second, we treated code search as a reranking problem in an attempt to recommend code snippets to developers. Specifically, we wanted to understand the impact of popularity of a repository containing the retrieved code snippets.

3.1 Research Questions

The overall goal of this thesis is to adapt and improve over the neural models developed in previous work [19], improve the state-of-the-art of semantic code search, and to investigate how code snippet recommendations can be improved.

RQ1: *Can bag-of-paths model be successfully applied to semantically search code snippets?* Learning distributed representations has worked well for natural language such as Word2Vec. Programming languages have syntax that is very different from natural language syntax and results in code being redundant in the form of variable names and conditionals. This means that the syntactic nature of code could be a crucial feature to any source code related research problems. Hence, we are interested in implementing distributed representations of code snippets as a feature to the search problem.

RQ2: *How effective is the resulting model, compared to previous benchmarks?* This neural model that uses Code2Vec to represent source code has the potential to achieve better results than the benchmark models that treats code as tokens. We wanted to analyse the performance of this model against the state of the art benchmark models that are already implemented. Baseline benchmark models are some of the complex models designed for Natural language processing tasks. Our hypothesis is that Natural language and Programming language are different in many ways [31] and a model that takes advantage of the syntactic nature of code will perform better than complex models that are specifically developed for NLP tasks.

RQ3: *Does the popularity of a repository impact the relevance of the retrieved code snippets?* A lot of factors could influence the relevance of code snippets in search results. Particularly, the metadata of the repository from which the code snippet was retreived. Metadata such as fork count, star count, repository size is correlated with the popularity of a repository [4]. This raises several research questions: Do developers prefer code snippets from a popular repository? Do developers prefer shorter code snippets in search results? CodeSearchNet challenge evaluates a ranked result set for a given set of queries against a

ground truth of relevant snippets and calculates a score. We wanted to rerank the retrieved code snippets based on the popularity of their projects and observe how it would impact the calculated score, in other words, relevance.

3.2 Initial study

As a first step, we replicated a study performed by GitHub engineers and attempted to improve the results using StackOverflow data that is rich in developer discussion. This initial study was performed on Python dataset provided by GitHub. The data was collected through Google BigQuery, which is an open dataset of GitHub repositories collected by Google. The data is then parsed into (code snippet, documentation) pairs using abstract syntax trees (ASTs) and segregated into train, test and validation sets. Github engineers had also provided the code to replicate the general approach needed to build a semantic search engine.

3.2.1 Data Preprocessing

The query used to retrieve the python repository and files only includes files that contain at most 15000 lines and has at least one function definition. The query results in 1.2 million raw Python files to be pre-processed. These raw python files were available for download as ten CSV files. During pre-processing, these ten files were used to generate ASTs and (code snippet, documentation) pairs were extracted.

3.2.2 Feature Extraction

During the initial study, code snippets are encoded using a seq2seq neural model [40] whereas documentation is encoded using a language model [51]. Then, the code vectors are mapped into the same vector space as documentation's vectors. Then the vectors are indexed using a non-metric space library, a library to perform similarity search in vector space, and the search is performed.

3.2.3 Model

The Model replicated during the study is called a seq2seq model [40], used typically to translate from one language to another. In our study, we translate code snippets to produce

natural language, in other words, code summarization. This summarized text is then encoded into vectors and indexed using NMSlib [28] for retrieval. NMSlib library is used to find nearest neighbours in a vector space. The seq2seq model is trained to predict/generate text that is similar to the actual documentation of the snippet. Once this step is completed, another model called a language model is trained to encode all the available documentation. This trained language model is used to convert a given query to a vector. Using this vector, a similarity search is performed on the indexed vectors (which are linked to code snippets). To evaluate the seq2seq model, Bilingual Evaluation Understudy (BLEU) metric was used. BLEU ranges between 0 and 1, this tells us how close a translated sentence is to that of a reference sentence. If the result is closer to 1, it implies that the accuracy of the translation is high. This metric is typically used in neural machine translation tasks [24].

The approach we replicated treats the problem as a code summarization problem and then uses it for code search which is inefficient as the search efficiency depends on how well the model summarizes the code snippet. Another caveat to the approach is that there was no common evaluation dataset available for code summarization and semantic code search. In hindsight, this study helped us in understanding the process of applying neural models to semantic code search.

3.2.4 CoNaLa

We then applied the same model (mentioned in section 3.2.3) with CoNaLa dataset [50] instead of Github data. CoNaLa is a publicly available dataset that was crawled from stackoverflow to mine high quality Code/Natural Language pairs. Results from these experiments were promising and served as a preliminary step in applying neural models to semantic source code search.

3.3 CodeSearchNet Challenge

The initial replication study we conducted did not have a proper evaluation dataset with ground truth to measure the performance of the neural models applied. However, with the release of CodeSearchNet [19] challenge from GitHub, evaluating the progress of semantic search was made possible. CodeSearchNet challenge, along with baseline benchmark neural models, provides CodeSearchNet corpus which consists of 6 million functions from open-source code spanning six programming languages (Go, Java, JavaScript, PHP, Python, and Ruby). It also provides researchers with 99 natural language queries with about 4k expert relevance annotations of likely results from CodeSearchNet Corpus for evaluation.

3.3.1 Data Collection

CodeSearchNet Corpus comprises of data collected from publicly available open-source non-fork GitHub repositories and projects which are used at least by one other project. Moreover, the projects that do not contain licenses or re-distribution restricted licenses are removed. Files from each projects were parsed to functions (methods) and then tokenized accordingly for six different programming languages: Go, Java, JavaScript, Python, PHP and Ruby using TreeSitter — GitHub's universal parser — and additionally if needed, respective documentation text was extracted using a heuristic regular expression. For our study, we only use the Java corpus provided by the challenge.

3.3.2 Data Preprocessing

CodeSearchNet dataset has already been preprocessed and all the steps involved in data collection and processing has been provided in the original paper [19]. However, we have mentioned some of the crucial steps in this section. As a first step, *(function, documentation)* pairs were extracted. Documentation is truncated to first paragraph and pairs with documentation less than three tokens are excluded. Pairs with function names that start with "test" are removed in order to exclude test case functions. Finally, duplicates are removed by identifying near duplicates.

3.3.3 Running Baseline Models

Four baseline model architectures [19] developed in the original paper: *Neural Bag of Words* (NBoW), *Bidirectional RNN models* (bi-RNN), *1D Convolutional Neural Network* (1D-CNN), and *Self-Attention* (SelfAttn). We obtained the code for each of the baseline models and dataset associated from the CodeSearchNet challenge. During this step, we setup the pipeline to run the baseline models and generated results. We configured the environment variables, project directories, model logs and checkpoints; after this, We were able to successfully run the baseline models. Out of the four neural baseline models, NBoW (Neural Bag of words model) performed the best. NBoW model treats both query(text) and snippets(code) as bag of words/tokens during encoding. However, NBoW may be better for encoding text but for code, it completely discards the syntactic structure. This led us to explore semantic search through a neural model that captures not only the semantic representation but also the syntactic representation of a code snippet.

3.4 Code2Vec Model

Code2Vec is a neural model that can represent code as continuous distributed vectors, or in other words, code embeddings [3]. This model decomposes code snippets to a collection of paths using Abstract Syntax Trees (ASTs) and learns the atomic representation of aggregated paths.

3.4.1 Feature Extraction

Function-documentation pairs for Java language is available in the CodeSearchNet corpus as .jsonl (json lines) files. Each line in the file represents a json object containing the function definition, documentation and metadata about the repository from which the function was taken. These json lines are then fed into a parser script line by line and paths are extracted. The script is developed by authors of Code2Vec model (Alon et al. [3]), based on an opensource AST miner called PathMiner [22] from JetBrains research.

```
int f(int n) {
    return n*n;
}
```

Listing 3.1: code snippet example for calculating square of a given number

An example of how paths are extracted from code snippets through ASTs is explained as follows. Consider a code snippet (listing 3.1) that calculates the square value of a given number. The snippet is parsed to construct an AST which is shown in Figure 3.2. Then, AST is traversed and syntactic paths between AST leaves are extracted. Each path is represented as a string containing the sequence of AST nodes, linked by up and down arrows, which symbolize the up or down link between adjacent nodes in the tree. Some of the example paths extracted from the above AST would be,

- (f, Method Declaration↓BlockStmt↓ReturnStmt↓Binary Expr:times , n) denoted by the colour green in Figure 3.2.
- (n, parameter↑Method Declaration↓BlockStmt↓ReturnStmt↓Binary Expr:times , n) denoted by the color red in Figure 3.2.

The path composition is kept with the values of the AST leaves it is connecting, as a tuple which is referred to as a path-context [3]. These paths are then added to the corresponding json object (from which the function definition was taken) and stored. In order to reduce space, during model training, a hashing function is applied on the paths before storage.

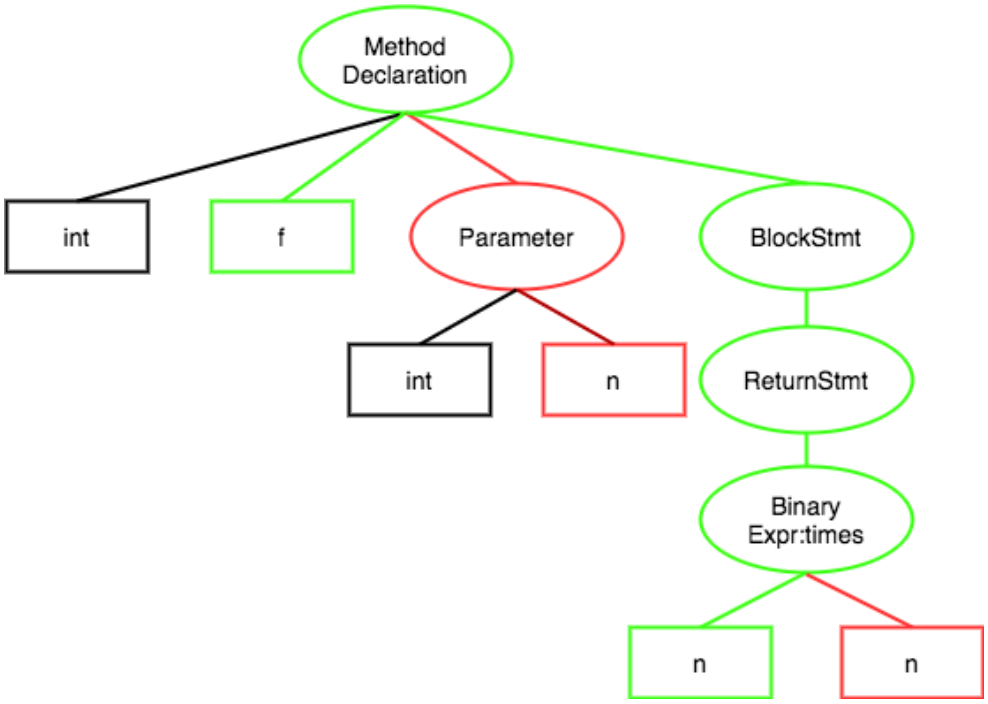

Figure 3.2: AST Representation of listing 3.1

3.4.2 Tensorflow migration

Code2Vec model is written originally in Tensorflow 2.0 and CodeSearchNet framework is written in Tensorflow 1.12. In order to use the path-based neural model for source code, one of the repositories had to be migrated to the other version to eliminate version mismatch issues. We decided to migrate the CodeSearchNet framework to Tensorflow 2.0 as upgrading (than downgrading) a repository is better in the long term for the research community as new versions are frequently updated for performance or bug fixes by the developers.

Tensorflow provides migration scripts to migrate from one version to the other. Although, the automation scripts made the migration a lot easier, manual changes to code were still required to get the framework running for the baseline models. These manual changes were made using the Tensorflow migration guide [41].

3.4.3 Model Architecture and implementation

After the framework migration was completed, Code2Vec model was rewritten into framework of CodeSearchNet baseline models. This re-implementation makes the model possible to be used as a encoder for code snippets. This change is also necessary for evaluating the model along with other baseline models. The framework already has certain input, output and evaluation format which is uniform across all the baseline models. This enables future research – including replication and minor improvements – once a new model is added and released to the benchmark.

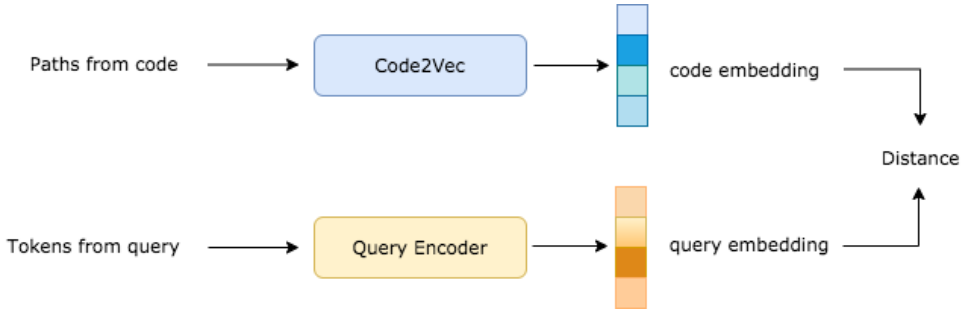

Figure 3.3: Model Architecture Overview

A brief overview of the architecture of the model we designed for CodeSearchNet challenge is shown in Figure 3.3. Code snippets are encoded using the migrated and re-implemented Code2Vec model and documentation for each of the code snippet is split into tokens and encoded using one of the benchmark baseline query encoders [19].

Query encoders that are available in the benchmark follows one of the architectures mentioned below,

- *Neural Bag of Words (NBoW)* [36], in which each token is embedded to a learnable embedding (vector representation).
- *Continuous Bag of Words (CBoW)*, where the model learns context by predicting next token by using surrounding tokens [26].
- *Bidirectional RNN model*, in which each input sequence is summarized using Gated Recurrent Units (GRU) [6].
- *1D Convolutional Neural Network*, applied over the input token sequence [20].

- *Self-Attention*, where each token representation in the sequence is computed using multi-head attention [42].

By model architecture, we refer to the overall framework we follow rather than the individual neural architectures of Code2Vec or the baseline models. Detailed information on each of those neural architectures can be found in the original papers [3, 19].

3.4.4 Model training

The objective is to map code and the corresponding documentation onto vectors that are closer to each other, as one can then implement a search method by embedding the query and then returning the set of code snippets that are "closer" in embedding (vector) space. To learn these embedding functions, the encoder models in the architecture shown in Figure 3.3 are combined. First, input sequences are preprocessed according to their semantics: identifiers appearing in code tokens are split into subtokens (i.e., a variable *ToLowerCase* is split into three subtokens *to, lower,* and *case*), and natural language tokens are split using byte-pair encoding(BPE) [37]. The training data for the model comprises of 542,991 java function-documentation pairs. This dataset is segregated into 80-10-10 train-validate-test sets.

During training, each code-documentation pair (c_i, d_i) is passed into code encoder E_c and query encoder E_q respectively. Then the model is trained by minimizing the loss,

$$-\frac{1}{\mathrm{N}}\sum_{\mathrm{i}}\log\frac{\exp(\mathrm{E_c(c_i)E_q(d_i)})}{\sum_{\mathrm{j}}\exp(\mathrm{E_c(c_j)E_q(d_j)})}$$

In the above equation, (c_j, d_j) denotes a negative pair (distractor snippet). This method of learning using negative samples is called a triplet loss function [49]. In other words, maximize the inner product of the code and query encodings of the pair, while minimizing the inner product between each c_i and the distractor snippets c_j. The code for the code2vec model implemented in the benchmark repository can be found here [16]. Mean Reciprocal Rank (MRR) is the metric used while training the model to evaluate its performance against the validation dataset. During testing, all the functions in CodeSearchNet search corpus is indexed using Annoy [38], an approximate nearest neighbor (nn) indexing and search library offered by Spotify [39]. We also experimented with different nearest neighbor searching techniques offered by the library such as Dot distance, Euclidean distance, Hamming distance, and Manhattan distance.

3.4.5 Hardware and Hyperparameter Specifications

We trained Code2Vec on a normal desktop machine that has NVIDIA GEFORCE GTX 1070 GPU engine and has 8GB of memory. We use the following set of hyperparameters to train models: batchsize as 200, learning rate as 0.01 and dropout keep rate as 0.9. We used the Adam [21] optimizer to update the parameters. Training 2400 batches of data consisting of approximately 500,000 samples costs 300 minutes of user time for 75 epochs. We also experimented with different embedding sizes (32, 64 and 128) and found that the training loss was marginally minimal when the embedding size is set to 64. During evaluation, creating vector representations and indexes in memory for six million code snippets needed a bigger machine. Hence, predictions were generated using a machine that had 200GB of RAM and three Tesla k80 GPUs each having 25 GB of memory. Evaluation took approximately 60 minutes to complete one run of predictions.

3.5 Reranking

A trained model and the indexes generated using Annoy are used to create the model predictions. Model predictions file contain the 99 search queries and a retrieved result set of 100 code snippets per query. We treat these predictions as a candidate result set to a reranking problem. We wanted to find out if there was a correlation between a repository's popularity and a retrieved code snippet.

3.5.1 Data Collection and Pre-processing

CodeSearchNet corpus provides repository URLs from which the corresponding code snippets were extracted. We parsed the Java dataset and obtained all the unique URLs from which the training data came from. Once the URL information is available, repository name can be extracted using simple regular expression. We can pass this information to GitHub and query all the metadata information that is publicly available for this repository. GitHub provides a comprehensive REST API [13] to query such information using a developer token that can be created from a valid GitHub user account. Scripts essential to do this data collection were written in Python and made available online.

Before the data collection, basic analysis was performed on the Java dataset to check the distribution of snippets across repositories. Distribution of the snippets can be seen in Table 3.1. For instance, the first row in the table indicates that four repositories, each provided at least 10000 code snippets to be used for training.

Snippets Threshold	Number of Repositories
>10000	4
5000 - 9999	6
1000 - 4999	60
100 - 999	810
<100	3893

Table 3.1: Distribution of snippets to repository

From the distribution, we observed that the snippets come from a wide variety of repositories. For instance, the four repositories having at least 10000 snippets indicate that these repositories offered more functions with documentations. The four repositories were *aws/aws-sdk-java, OpenLiberty/open-liberty, alkacon/opencms-core,* and *Azure/azure-sdk-for-java*. Upon manual inspection, we identified that two of these repositories are popular open source projects maintained by private corporations Amazon and Microsoft for their cloud services. These repositories [35] had better fork counts, subscribers count and contributors count than those repositories which had fewer code snippets. This may mean that the top projects were better engineered than the others or it could just mean the top projects were better documented. In either case, it is safe to assume that the snippets come from different kinds of repositories.

3.5.2 Re-ranking strategies

As the ground truth relevance annotations were not open sourced by the CodeSearchNet authors, we came up with different re-ranking strategies and evaluating them by performing multiple submissions and obtaining the score for each submission. By executing this method, we can quantitatively understand the variations in relevance score and conclude whether the re-ranking was successful or not. The various re-ranking strategies we employed are as follows:

Re-ranking with only metadata

- Top 100 results retrieved were re-ranked by the repository's fork count.
- Top 100 results retrieved were re-ranked by the repository's size.
- Top 100 results retrieved were re-ranked by the snippet's token length.

- Top 10 results retrieved were re-ranked by the repository's fork count. Here, the reasoning being highly relevant results to be only in the top 10 results.

Re-ranking with prediction scores and metadata

In this strategy, we re-ranked the snippets based on two data points: Prediction scores calculated during retrieval and Repository metadata. We re-ranked snippets based on metadata(fork count, repository size, and code token length) only within a specified window size and difference in prediction scores is not more than 0.02. For instance, if the window size is 10, we take the first 10 snippets and each pair from one through ten is compared for difference in prediction scores, if the difference was less than 0.02 then we rerank them using the metadata. This is repeated for the next batch of 10 snippets all the way up to 100. We follow this process for each of 99 evaluation queries.

The reasoning behind this strategy is if the scores are substantially different between the two snippets being compared, re-ranking should not be done in order to avoid ranking irrelevant snippets higher than relevant snippets purely because the metadata is better. It should be noted that, this strategy assumes the model does not return highly relevant snippets at lower ranks (for instance, at rank 90, 95, 100). We also applied re-ranking to submissions that had the best results in the CodeSearchNet leaderboard. We report our findings in Chapter 4.

3.6 Evaluation and Metrics

Evaluation is performed against a set of 99 queries [14] obtained from common search queries from Bing that had high clickthrough rates to code. Search space is a huge corpus of approximately 1.57 million functions(1,569,889). The search queries have already been manually annotated with relevant snippets with the help of developers and researchers [19]. The snippets were graded on a relevance scale of 0-3. Zero being irrelevant, 1 being a weak match, 2 being a strong match and 3 being an exact match. Specifically, for java dataset, there were 823 manual annotations that serves as the ground truth for the search queries. The authors of the CodeSearchNet challenge did not release this ground truth to the public to avoid over-fitting the neural models and enable a fair evaluation strategy to all the model submissions.

To evaluate the model's performance, the predictions file containing the results should be uploaded to an online platform called Weights and Biases (WandB) [44]. WandB has

a leaderboard that tracks model submissions from various models built by researchers and developers around the globe. Once the predictions file is uploaded, evaluation is run against the ground truth and an NDCG [45] measure is generated. A detailed overview of this metric is explained in the following section. We calculate the NDCG score for the Code2Vec based neural model and use this as the baseline for reranking the results as well as the ground truth is hidden. After reranking the results using multiple strategies, we analyse the variation in the resulting NDCG score to measure the impact on relevance.

3.6.1 BLEU

BLEU (bilingual evaluation understudy) [47] is an algorithm for evaluating the quality of text that has been machine-translated from one natural language to another. In this case, the translation of programming language to natural language (predicting the documentation). Quality is considered to be the correspondence between a machine's output and that of a human: "the closer a machine translation is to a professional human translation, the better it is" – this is the central idea behind BLEU. This metric does not take into account intelligibility or grammatical correctness of a translated text. We only used this metric in our initial preliminary study as the model we applied summarizes the code snippets before performing the retrieval.

3.6.2 Mean Reciprocal Rank

The Reciprocal Rank (RR) is an information retrieval measure that calculates the reciprocal of the rank at which the first relevant document was retrieved. For example, RR is set to be 1 if a relevant document was retrieved at rank 1, if not it is set to 0.5 if a relevant document was retrieved at rank 2 and so on. When this measure is averaged across queries, it is referred to as the Mean Reciprocal Rank (MRR) [7].

This metric is usually used with a retrieval model where only one relevant document needs to be found. Typically, this measure is very useful to evaluate the model during training as there is usually one right answer per query available for training.

3.6.3 Normalized Discounted Cumulative Gain

Discounted Cumulative Gain (DCG) [48] is a measure of ranking quality and is used to evaluate models based on the effectiveness of ranking. This metric uses a graded relevance

scale of documents against a search model's result set, and measures the usefulness or gain of a document based on its position in the result set. This measures relies on two fundamental assumptions,

- Highly relevant documents are more useful when appearing earlier in a search engine result list (have higher ranks).
- Highly relevant documents are more useful than marginally relevant documents, which are in turn more useful than non-relevant documents.

Below example will describe the difference among Cumulative Gain, Discounted Cumulative Gain, and Normalized Discounted Gain.

Cumulative Gain (CG) is the sum of the graded relevance values of all results in a search result list. This predecessor of DCG does not include the rank (position) of a result in the result list into the consideration of the usefulness of a result set. The CG at a particular rank position p is defined as:

$$\mathrm{CG_p} = \sum_{i=1}^{p} rel_i$$

Where rel_i is the graded relevance of the result at position i. This value calculated (CG) is not affected by the ordering of the search results. In other words, even if a highly relevant result is placed above a less relevant result, the score remains unchanged.

In order to correct that, DCG penalizes highly relevant documents appearing lower in a search result. This is accomplished by logarithmically reducing the graded relevance value proportional to the position of the result.

DCG accumulated at a particular rank position p is defined as:

$$\mathrm{DCG_p} = \sum_{i=1}^{p} \frac{rel_i}{\log_2(i+1)}$$

Search result lists vary in length depending on the query. Comparing a model's performance from one query to the next cannot be consistently achieved using DCG. Hence, the cumulative gain at each position should be normalized across queries. This is done

i	rel_i	$\log_2(i+1)$	$\frac{rel_i}{\log_2(i+1)}$
1	3	1	3
2	2	1.585	1.262
3	3	2	1.5
4	0	2.322	0
5	1	2.585	0.387
6	2	2.807	0.712

Table 3.2: DCG calculation for each position

by sorting the relevance documents by their relative relevance to produce the maximum possible DCG called as Ideal DCG. Then, NDCG can be calculated as:

$$\mathrm{NDCG_p} = \frac{DCG_p}{IDCG_p}$$

Given a list of documents in response to a search query, an experiment participant is asked to judge the relevance of each document to the query. Each document is to be judged on a scale of 0-3 with 0 meaning not relevant, 3 meaning highly relevant, and 1 and 2 meaning "somewhere in between". For the documents ordered by the ranking algorithm as

$$D_1, D_2, D_3, D_4, D_5, D_6$$

the user provides the following relevance scores:

$$3, 2, 3, 0, 1, 2$$

That is: document 1 has a relevance of 3, document 2 has a relevance of 2, etc. The Cumulative Gain of this search result listing is:

$$\mathrm{CG_6} = \sum_{i=1}^{6} rel_i = 3 + 2 + 3 + 0 + 1 + 2 = 11$$

Changing the order of any two documents does not affect the CG measure. If D_3 and D_4 are switched, the CG remains the same, 11. DCG is used to emphasize highly relevant documents appearing early in the result list. Using the logarithmic scale for reduction, the DCG for each result in order is calculated and shown in Table 3.2.

So the DCG_6 of this ranking is:

$$\mathrm{DCG}_6 = \sum_{i=1}^{6} \frac{rel_i}{\log_2(i+1)} = 3 + 1.262 + 1.5 + 0 + 0.387 + 0.712 = 6.861$$

Now a switch of D_3 and D_4 results in a reduced DCG because a less relevant document is placed higher in the ranking; that is, a more relevant document is discounted more by being placed in a lower rank.

The performance of this query to another is incomparable in this form since the other query may have more results, resulting in a larger overall DCG which may not necessarily be better. In order to compare, the DCG values must be normalized.

To normalize DCG values, an ideal ordering for the given query is needed. For this example, that ordering would be the monotonically decreasing sort of all known relevance judgments. In addition to the six from this experiment, suppose we also know there is a document D_7 with relevance grade 3 to the same query and a document D_8 with relevance grade 2 to that query. Then the ideal ordering is:

$$3, 3, 3, 2, 2, 2, 1, 0$$

Without the D7 and D8, the ideal ordering is:

$$3, 3, 2, 2, 1, 0$$

The DCG of this ideal ordering, or IDCG (Ideal DCG) , is computed to rank 6:

$$\mathrm{IDCG}_6 = 7.141$$

And so the NDCG for this query is given as:

$$\mathrm{nDCG}_6 = \frac{DCG_6}{IDCG_6} = \frac{6.861}{7.141} = 0.961$$

As mentioned before, BLEU and NDCG are the two metrics we used to evaluate the performance of our model. In the next chapter, we will discuss the results of our experiments.

Chapter 4

Results

4.1 Results from Initial Study

The initial study was applying seq2seq [40] model to code search task. We performed the task using two different datasets: GitHub function-docstring pairs [15], CoNaLa stackoverflow snippet-intent pairs [50]. Some of results showing the code summarization generated by the model on these two different datasets is shown in Table 4.1. The BLEU score for models that were trained on Github code-docstring pairs and CoNaLa snippet-intent pairs were 0.071 and 0.0013 respectively. It is important to note that, the training datasets used were different in size. GitHub dataset contained about 2 million function-docstring pairs where CoNaLa had 600 thousand snippet-intent pairs [50].

Code summarization is a problem in itself and is being actively studied by software engineering research community. Our objective for doing this study was to understand the process of using neural models for semantic code search. Some of the study's findings were promising and encouraged us to find and experiment with new neural models to do semantic search. In this particular study, the model performance was difficult to be evaluated as the dependency towards effective code summarization is enormous. Training data available for code summarization was limited and quantitatively evaluating the search was difficult as there was no reliable human annotated ground truth dataset available.

Table 4.1: Comparison of seq2seq model results

Original Title	**Predictions using Github dataset**	**Predictions using CoNaLa dataset**
get the enriched events related to a module	return the rich data for the item	how to mock a tornado coroutine function using mock know
test refresh identities	refresh the identities list	how to decide the language from cookies headers session
get elasticsearch mapping	get elastic mappings from elasticsearch	elasticsearch bulk index in chunks using pyes
fix <null >values in some jira parameters .	if value is none return none	doctest involving escape characters

4.2 Code2vec with Benchmark Baselines

This section will showcase results obtained from applying Code2Vec to semantic code search alongside other neural benchmark models available in the CodeSearchNet challenge. Results include MRR and NDCG scores obtained during model evaluation.

4.2.1 MRR baselines

Table 4.2 shows the comparison of Mean Reciprocal rank across all the benchmark baseline models along with the code2vec model. As it can be observed, Self-Attention model performs best during training.

The columns text and code refer to the representation models used for documentation and source code snippets respectively. The columns Validation MRR and Test MRR refer to the MRR results obtained for validation set and test set respectively. Column FuncName Test MRR is MRR calculated on test set but with the function names as queries instead of docstrings. All the models performed substantially better with function names as queries. This is expected, actual queries might not have method names in them. Thus, when the method names are supplied as queries, the code snippets that have the same method name is being ranked higher.

Encoder		CodeSearchNet Java Corpus		
Text	**Code**	**Validation MRR**	**FuncName Test MRR**	**Test MRR**
1D-CNN	1D-CNN	0.493	0.724	0.527
biRNN	biRNN	0.271	0.503	0.286
SelfAttn	**SelfAttn**	**0.551**	**0.815**	**0.587**
NBoW	NBoW	0.497	0.616	0.514
NBoW	**Code2Vec**	**0.500**	**0.661**	**0.516**
CBoW	Code2Vec	0.2952	0.494	0.3121
1D-CNN	Code2Vec	0.0108	0.01418	0.01194
biRNN	Code2Vec	0.08922	0.08309	0.07113
SelfAttn	Code2Vec	0.0809	0.0902	0.07

Table 4.2: Comparison of MRR across baseline models

It is obvious from the training results that, only NBoW model works reasonably well alongside Code2Vec during training. We experimented Code2Vec with other query (documentation) encoders provided by the benchmark. However, these models were ineffective and essentially yielded poor results during search. It should also be noted that the predictions generated by these models did not contain highly relevant snippets in the top 100 and resulted in NDCG scores that are negligible (below 0.1).

4.2.2 NDCG baslines

Table 4.3 shows the Normalized Discounted Cumulative Gain for all the basline models along with Code2Vec. It can be observed that Code2vec (along with NBoW) outperforms all the previous baselines developed in the original paper [19]. We consider this as a significant step forward in terms of the research problem addressed. We have demonstrated that it is possible to extract semantic meaning from redundant and syntactic structure of source code and achieve results.

The results also validate our hypothesis that complex neural models that works well for natural language (text) or other media types (images, videos) may not necessarily be as efficient when applied to programming language related tasks. For example, 1D-CNN model is typically used for image classification as it works well for image related machine

Encoder		CodeSearchNet Java Corpus
Text	**Code**	**NDCG Relevance Score**
1D-CNN	1D-CNN	0.1282
biRNN	biRNN	0.0623
SelfAttn	SelfAttn	0.1003
NBoW	NBoW	0.1207
NBoW	**Code2Vec**	**0.1484**

Table 4.3: Comparison of NDCG relevance across baseline models

learning tasks [20]. Bi-RNN and self attention is used extensively natural language tasks such as sentence classification [6]. These models have been outperformed by our hybrid model built with the help of Code2Vec. At the time of our submission to the CodeSearchNet challenge, our model stands in the top 10 results of the leaderboard. We have provided screenshots and other artifacts regarding our submission in chapter A.1 and A.2.

4.3 Reranking results

Our objective was to obtain the ideal or near-ideal ordering of the retrieved snippets through reranking. Results for reranking the model predictions generated by Code2Vec model using different metadata have been presented in Table 4.6. The key observation here is that the re-ranking (with only metadata), yields NDCG that is approximately 50% down from the original score of **0.1484**.

Tables 4.4, 4.5 show some results for reranking based on the repository's fork count. Column *Rank* is the original rank generated by the neural model and *Rerank* is the new rank obtained after re-ranking. As it can be observed, snippets that were ranked lower previously have been brought to the top as the snippet is originated from a heavily forked repository. The table also implies that the results obtained after reranking is likely to have more irrelevant or less relevant snippets in higher ranks than highly relevant snippets.

In order to eliminate re-ranking totally irrelevant snippets, we re-ranked the predictions by taking into account the prediction scores. As we have mentioned in the chapter 3.5.2, for any two snippets being compared within the window size, if the scores differ by more than 0.02, the re-ranking is prevented. For this re-ranking, we obtained the NDCG score

Function name	Rank	Rerank	Repository	Forks
BitType.convertToString	1	22	VoltDB/voltdb	390
TextTable.convertToString	2	39	skadistats/clarity	98
DataTypeUtils.convertFromString	3	96	ivanceras/orm	0
ConstantsScanListener.convertString	4	89	jbundle/jbundle	1
JavaStringConverter.convertFromJavaString	5	46	eclipse/xtext-core	72

Table 4.4: Before reranking based on fork count for the query "convert int to string"

Function name	Rank	Rerank	Repository	Forks
UTF8String.fromString	77	1	apache/spark	21208
TypeConversion.convertString	54	2	deeplearning4j/deeplearning4j	4803
StringUtil.bytesToString	38	3	knightliao/disconf	2269
StringUtil.longToString	56	4	knightliao/disconf	2269
KeyValueFormat.parseStringDouble	90	5	SonarSource/sonarqube	1223

Table 4.5: After reranking based on fork count for the query "convert int to string"

of **0.1286**. It was evident that the score improved but it was still below the original submission without any re-ranking.

We performed experiments by changing the number of snippets retrieved as well. We limited our model predictions to the top 10 and discarded the rest of snippets to obtain a submission score of 0.108566551. The reasoning to discard snippets with rank more than 10 is to reduce the number of irrelevant snippets that have strong metadata during re-ranking. With this new score as a baseline, we performed the various re-ranking discussed above. The results for those submissions were still below this new score of 0.108566551.

Reranking method	Re-ranked NDCG
Rerank top 100 by fork count	0.077261896
Rerank top 100 by repo size	0.074158342
Rerank top 100 by code token length	0.075949323
Rerank only top 10 by fork count	0.084801173

Table 4.6: NDCG after reranking predictions using only metadata

We wanted to ensure this observation is consistent across various neural models submitted to CodeSearchNet leaderboard. We downloaded the predictions file for the leading model from the leaderboard. We applied the above mentioned re-ranking methods to the leading models and found that the observation was similar. Re-ranking based on metadata performed worse than the original predictions. The lesson we learnt from this re-ranking is that, metadata such as popularity and token length does not impact the relevance in a positive way. In other words, we cannot rerank code snippets based on the popularity of the projects.

Another strategy we employed during re-ranking was to use different types of similarity search in vector space, essentially trying to obtain different prediction scores. We tried hamming distance, dot distance, and, euclidean distance between the vectors provided by the Annoy [38] vector search library. We were not surprised to find out that the scores were not significantly different and the observations were consistent with what we had gathered previously. The underlying reason was still the same, re-ranking based on metadata was not the best way to recommend code snippets.

Thus, we conclude the results chapter. In the next chapter, we will discuss the answers to our research questions.

Chapter 5

Discussion

Although our quantitative results validate the performance of neural models towards code search task, our models are still a work-in-progress. In this section, we will discuss the findings of our experiments and answer our research questions mentioned in Chapter 3.

5.1 Research Questions

RQ1: *Can bag-of-paths model be successfully applied to semantically search code snippets?* **Yes.** We were able to successfully implement a hybrid model that encodes source code in a different way than typical neural models encode text. We had to use the other benchmark models to encode text as Code2Vec is designed to extract meaning from source code. We were able to successfully use artifacts from previous research [3, 19] and tensorflow migration scripts to develop this fully functioning hybrid model.

RQ2: *How effective is the resulting model, compared to previous benchmarks?* The hybrid model we designed was able to beat the previous benchmarks developed in the original CodeSearchNet paper [19]. We made a pull request to the CodeSearchNet repository and got it accepted. As a result of which, our model currently stands in the top 10 results of the leaderboard. However, it is important to note that this challenge is being actively solved by researchers and developers around the world.

RQ3: *Does the popularity of a repository impact the relevance of the retrieved code snippets?* **No.** In the context of semantic search, the result we have obtained through various re-ranking strategies show evidence that the metadata such as popularity of repository, repository size and token length of snippets do not impact the relevance of retrieved

Query	Sensible result	Contradicting result
convert int to string	NumberType.convertToString	Number.convertStringToInt
string to date	DateUtils.stringToDate	Util.dateToString
convert decimal to hex	NumericUtil.toHexShortString	SpatialiteReader.hexToInt
convert int to bool	LangUtils.convertToBoolean	BooleanUtils.toIntegerObject

Table 5.1: Directionality problem with neural models

snippets. Using such metadata for code recommendation through re-ranking will yield poor results.

5.2 Neural models are not perfect

Although our model in comparison with other neural models shows substantial improvement in results, neural models still face some challenges. Table 5.1 shows one of the important challenges we need to solve in order to improve the models: the directionality problem. For the type of queries that involve any form of conversions, neural models struggle to understand the intended direction of the query. For example, "convert from int to bool" query returns snippets that does the exact opposite. This remains an open challenge and we hope to solve this problem in the future by actively improving the model.

Another major shortcoming was that none of the complex neural models such as Self attention [42], biRNN [6], Conv NN [20] performed well when combined with Code2Vec [3]. These neural models have complex model architecture and have higher capacities compared to models such as NBoW. Understanding why these complex models struggle to learn features when combined with Code2Vec could potentially help changing these models to work well for semantic code search.

5.3 Contributions

We have presented the contributions of our study as follows:

- We have demonstrated that it is possible to integrate neural models into CodeSearchNet [19] that are specifically designed to extract features from source code (such as

Code2Vec). We have built an hybrid model that treats text and source code differently and still understand the semantics between them.

- We have provided a new baseline model to the CodeSearchNet benchmark by submitting our results to the challenge, enabling the research community to further research and improve upon.
- We have open-sourced all the preprocessing scripts, model source code (upgraded to Tensorflow 2.0) and other artifacts we developed or modified from prior research [16]. We believe this will help researchers in replicating the results and encourage them start building new models on top of neural models tailored for understanding source code.
- We have found that reranking a retrieved result set based on metadata such as popularity of the project, code token length, repository size would result poor code recommendation. In other words, popular projects do not result in more relevant search results.

5.4 Threats to Validity

- Documentation is fundamentally different from user queries. Queries are typically questions and documentation is a general description of a code snippet along with parameters and return type. Is it ideal to use them as a feature to learn the semantics? Moreover, documentation becomes outdated with time. At the moment, researchers (and developers) do not have any other option but to learn semantics from documentation as it bridges the gap between natural language (query) and code snippets (source gap).
- The relevance annotations used to evaluate the neural models were not fully utilized. Developers did not annotate the entire CodeSearchNet corpus as it takes a lot of manual effort and infeasible. Only a total approximately 4000 snippets were annotated. This means that a lot of code snippets which may be relevant and retrieved but not annotated. Thus, NDCG may not reflect the full capacity of the neural models but rather an approximation.
- Since the relevance annotations were hidden from the challenge in order to avoid over fitting the models, we were not able to answer interesting questions such as, what type of queries the neural models are capable of solving? How can the models achieve the ideal ordering of code snippets through re-ranking after retrieval?

- We have only applied Code2Vec to the Java corpus. There are five more programming languages that have evaluation datasets. Would Code2Vec be effective for any other language other than Java?
- Queries used for evaluation are relatively short and general purpose queries. We have not tested the models against project specific queries and queries that are long.
- We only used Github dataset and preprocessing artifacts provided by CodeSearchNet framework. How well these observations and model performance will translate to different datasets is an intriguing question yet to be answered.

CodeSearchNet challenge is inspired and designed similar in structure to data science challenges such as ImageNet [8] challenge. ImageNet challenge was a crucial step towards solving the challenges in image classification research. Like ImageNet, CodeSearchNet could be the next big step [43] in solving semantic search. Our model Code2Vec is a demonstration that building models specific to learn source code is also crucial towards semantic search.

All the benchmark models developed so far were not able to perform better than Elasticsearch [10] baseline with an NDCG score of **0.204**. However, semantic code search is a rapidly evolving research area. With the advent of CodeSearchNet, researchers are able to rapidly make improvements and advance in this particular area of research. When this thesis was being written, Feng et al [11] have developed a new model based on a natural language model called BERT [9] that made a huge impact in the field of NLP recently. This model, CodeBERT [11] named after BERT, has now outperformed all the neural models so far submitted to CodeSearchNet. This model is a bimodal pre-trained model for programming language (PL) and natural language (NL), in other words, it has learned to utilize both NL-PL data together using a single model architecture. This is different from our hybrid model which uses two different architectures (Code2Vec and NBoW). Such studies emphasize the rate at which the field is growing and potentially help finding the next "big thing" for code search.

Chapter 6

Conclusions

In our thesis, we have built a hybrid neural model that learns semantics from code snippets as well as natural language text. We have demonstrated that building neural models this way can improve the performance in searching code snippets. We also evaluated our model against previous benchmark models and outperformed those models in terms of relevance scores. We hope that this would encourage more researchers to represent and learn semantics from the syntactic nature of code snippets for semantic search and as well as other problems that deal with source code. We have also showed how re-ranking based on only metadata can negatively affect the results. Our study was heavily influenced and inspired by Code2Vec and the CodeSearchNet challenge and we hope to improve our solution in the future through iterative development of the designed model.

6.1 Future Work

- As we have mentioned before, we performed our experiments only on systems written in the Java programming language. We are interested in applying the same experiments on other languages. This would also mean learning to extract bag of paths by constructing ASTs for each of the language. We plan to use PathMiner [22], which has provided reusable artifacts, to mine path based representations for different programming languages.

- Replicating the model implementation to work for different datasets can be a great learning experience and it opens up the possibility of creating more training data for

existing models as well. Neural models have the ability to improve performance with more data.

- Representing source code in the form of directed acyclic graphs instead of paths could also lead to a neural model and improve our understanding of doing semantic search.
- Natural language processing is a rapidly growing area of research as new and improved neural models are discovered frequently. This also means new opportunities to represent documentation (text) through such improved models.

References

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[2] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*, pages 2091–2100, 2016.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[4] Hudson Borges, Andre Hora, and Marco Tulio Valente. Understanding the factors that impact the popularity of github repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344. IEEE, 2016.

[5] Jón Arnar Briem, Jordi Smit, Hendrig Sellik, and Pavel Rapoport. Using distributed representation of code for bug detection. *arXiv preprint arXiv:1911.12863*, 2019.

[6] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[7] Nick Craswell. Mean reciprocal rank. *Encyclopedia of database systems*, 1703, 2009.

[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[10] Elasticsearch. Elasticsearch source code. https://github.com/elastic/elasticsearch, 2019.

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[12] GitHub. Github search could be better. https://github.com/isaacs/github/issues/908, 2017.

[13] GitHub. Github rest api. https://developer.github.com/v3/, 2019.

[14] GitHub. Queries used to evaluate the models. https://github.com/Laksh47/CodeSearchNet/blob/master/resources/queries.csv, 2019.

[15] GitHub. Semantic code search using sequence to sequence model. https://towardsdatascience.com/semantic-code-search-3cd6d244a39c, 2019.

[16] GitHub. Code2vec semantic search. https://github.com/Laksh47/CodeSearchNet/tree/YANCS, 2020.

[17] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944. IEEE, 2018.

[18] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, pages 200–210, 2018.

[19] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[20] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[21] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[22] Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli. Pathminer: a library for mining of path-based representations of code. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 13–17. IEEE, 2019.

[23] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 476–481. IEEE, 2015.

[24] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[25] Microsoft. Roslyn compiler for dotnet. https://github.com/dotnet/roslyn, 2019.

[26] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[27] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[28] Bilegsaikhan Naidan, Leonid Boytsov, Yury Malkov, and David Novak. Non-metric space library manual. *arXiv preprint arXiv:1508.05470*, 2015.

[29] Hacker News. Github search sucks. https://news.ycombinator.com/item?id=13935370, 2017.

[30] Haoran Niu, Iman Keivanloo, and Ying Zou. Learning to rank code examples for code search engines. *Empirical Software Engineering*, 22(1):259–291, 2017.

[31] Sheena Panthaplackel, Milos Gligoric, Raymond J Mooney, and Junyi Jessy Li. Associating natural language comment and source code entities. *arXiv preprint arXiv:1912.06728*, 2019.

[32] Md Masudur Rahman, Jed Barson, Sydney Paul, Joshua Kayani, Federico Andrés Lois, Sebastián Fernandez Quezada, Christopher Parnin, Kathryn T Stolee, and Baishakhi Ray. Evaluating how developers use general-purpose web-search for code retrieval. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 465–475, 2018.

[33] Xin Rong. word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*, 2014.

[34] Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. How developers search for code: a case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 191–201, 2015.

[35] Script. Manual analysis on the repositories. https://github.com/Laksh47/scripts_codesearch/blob/master/analyse_repo_data.ipynb, 2020.

[36] Imran Sheikh, Irina Illina, Dominique Fohr, and Georges Linares. Learning word importance with the neural bag-of-words model. In *Proceedings of the 1st Workshop on Representation Learning for NLP*, pages 222–229, 2016.

[37] Yusuxke Shibata, Takuya Kida, Shuichi Fukamachi, Masayuki Takeda, Ayumi Shinohara, Takeshi Shinohara, and Setsuo Arikawa. Byte pair encoding: A text compression scheme that accelerates pattern matching. Technical report, Technical Report DOI-TR-161, Department of Informatics, Kyushu University, 1999.

[38] Spotify. Annoy approximate nearest neighbours. https://github.com/spotify/annoy, 2019.

[39] Spotify. Spotify. https://github.com/spotify, 2019.

[40] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[41] TensorFlow. Tensorflow migration guide. https://www.tensorflow.org/guide/migrate, 2019.

[42] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[43] WandB. Imagenet for source code search. https://www.wandb.com/articles/codesearchnet, 2019.

[44] WandB. Weights and biases. https://app.wandb.ai/github/CodeSearchNet/benchmark/leaderboard, 2019.

[45] Yining Wang, Liwei Wang, Yuanzhi Li, Di He, and Tie-Yan Liu. A theoretical analysis of ndcg type ranking measures. In *Conference on Learning Theory*, pages 25–54, 2013.

[46] Wikipedia. Google code search. https://en.wikipedia.org/wiki/Google_Code_Search, 2017.

[47] Wikipedia. Bilingual evaluation understudy. https://en.wikipedia.org/wiki/BLEU, 2019.

[48] Wikipedia. Normalized discounted cumulative gain. https://en.wikipedia.org/wiki/Discounted_cumulative_gain, 2019.

[49] Wikipedia. Triplet loss function. https://en.wikipedia.org/wiki/Triplet_loss, 2019.

[50] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 476–486. IEEE, 2018.

[51] ChengXiang Zhai. Statistical language models for information retrieval. *Synthesis Lectures on Human Language Technologies*, 1(1):1–141, 2008.

Appendix A

APPENDICES

A.1 Our Tools, Artifacts, Results

Links to our code, pre-processing & reranking scripts, benchamrk submissions, and results are as follows:

- GitHub link to Code2Vec integrated model
- Scripts to crawl GitHub metadata and re-ranking strategies
- Benchmark submission of the model performance with logs
- Re-ranking submissions
- Link to Leaderboard
- Original CodeSearchNet datasets and source code
- Original Code2Vec source code

A.2 Figures

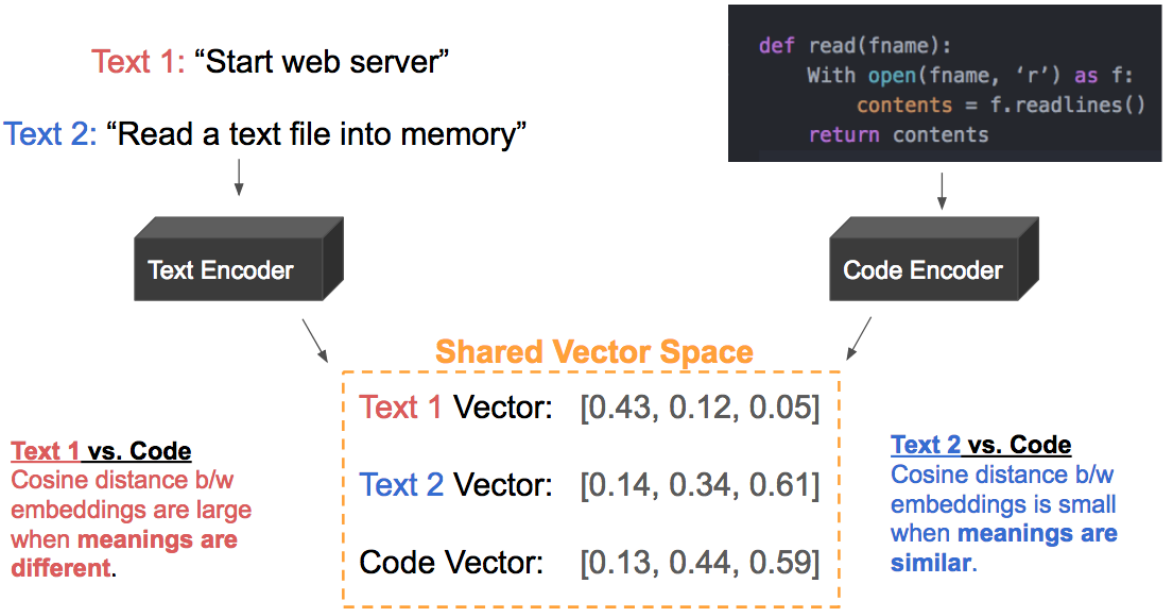

Figure A.1: code-documentation to vector representation

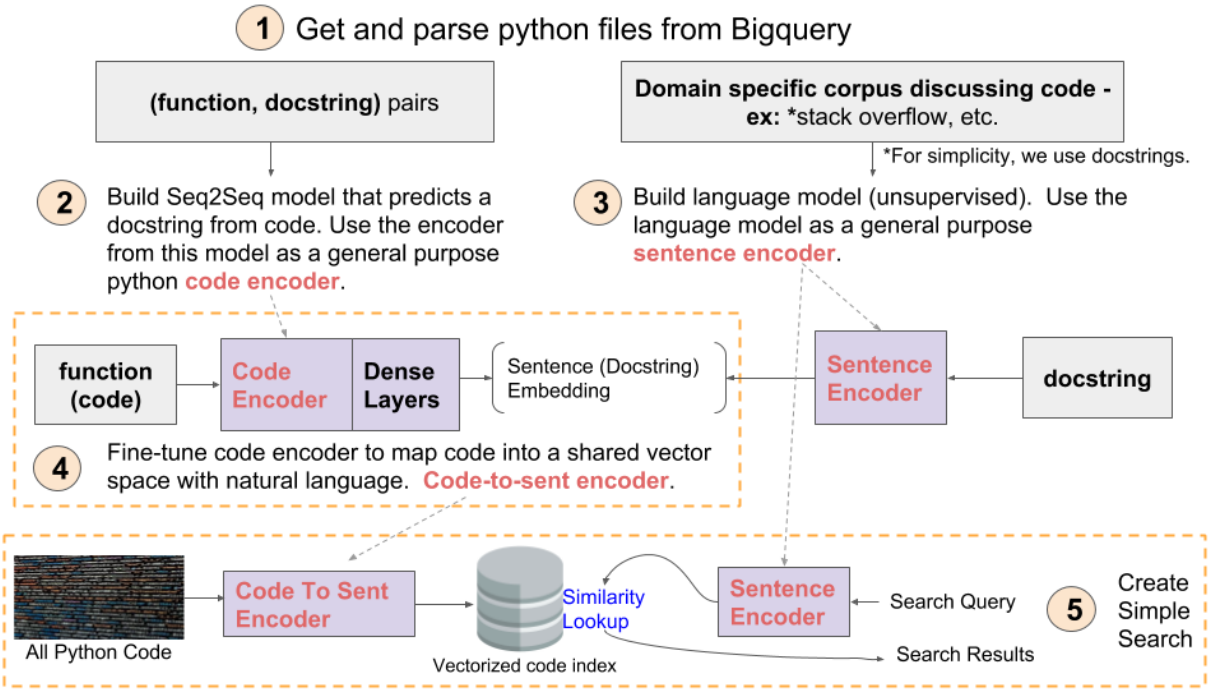

Figure A.2: Pipeline used for the initial study

```
{
  "question_id": 36875258,
  "intent": "copying one file's contents to another in python",
  "rewritten_intent": "copy the content of file 'file.txt' to file
  ↪  'file2.txt'",
  "snippet": "shutil.copy('file.txt', 'file2.txt')",
}
```

Figure A.3: A sample pair from CoNaLa dataset

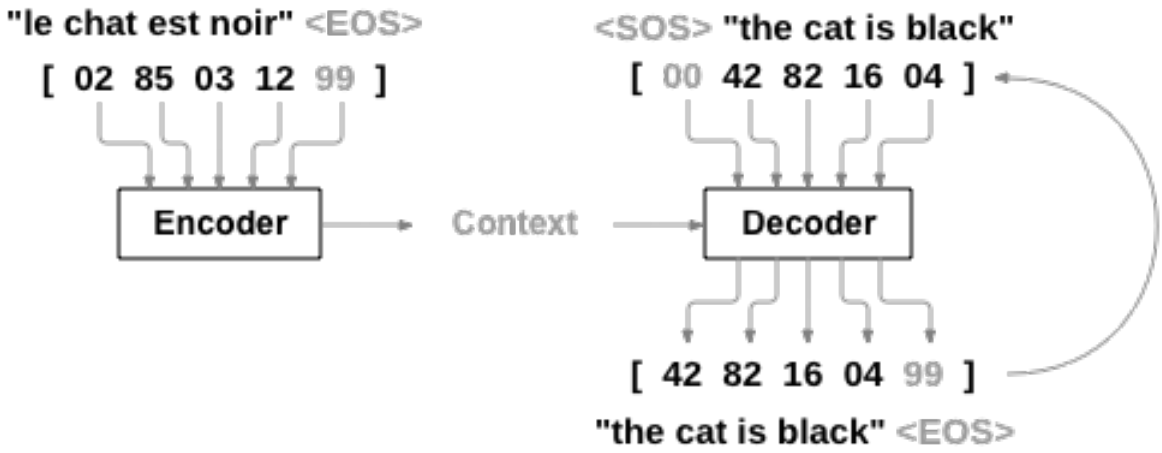

Figure A.4: Example of seq2seq model

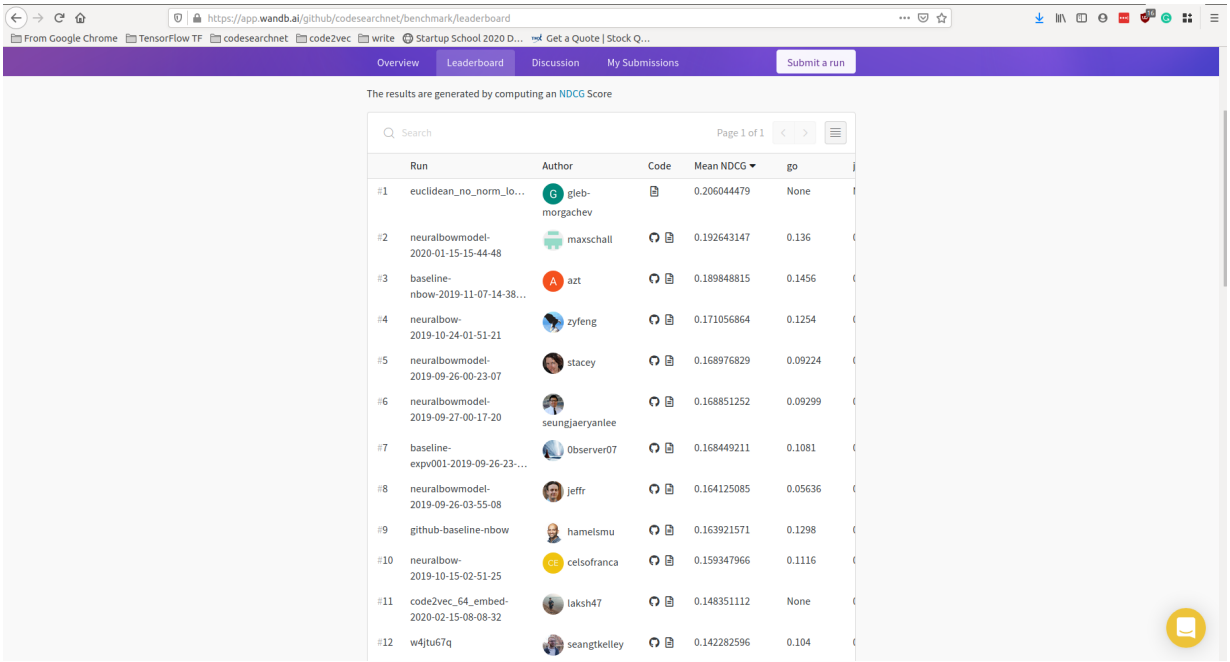

Figure A.5: Our model stands 11th in the CodeSearchNet Leaderboard

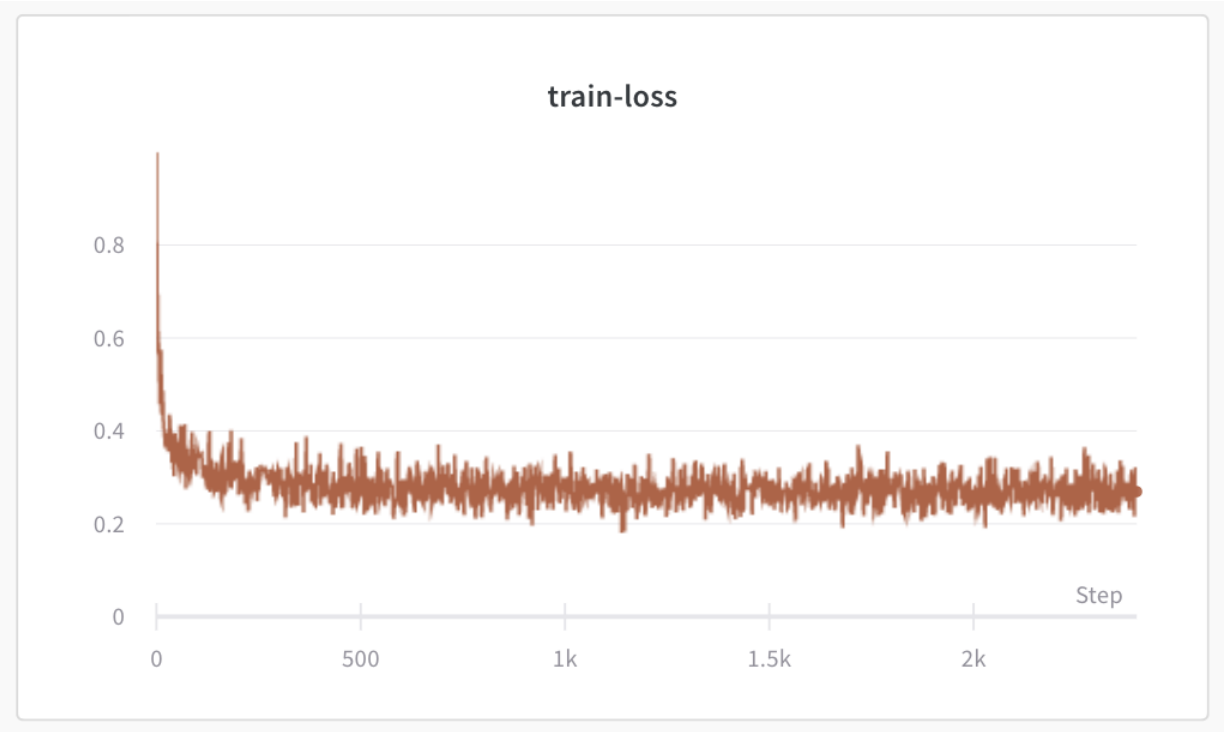

Figure A.6: Code2Vec model training loss during each train step

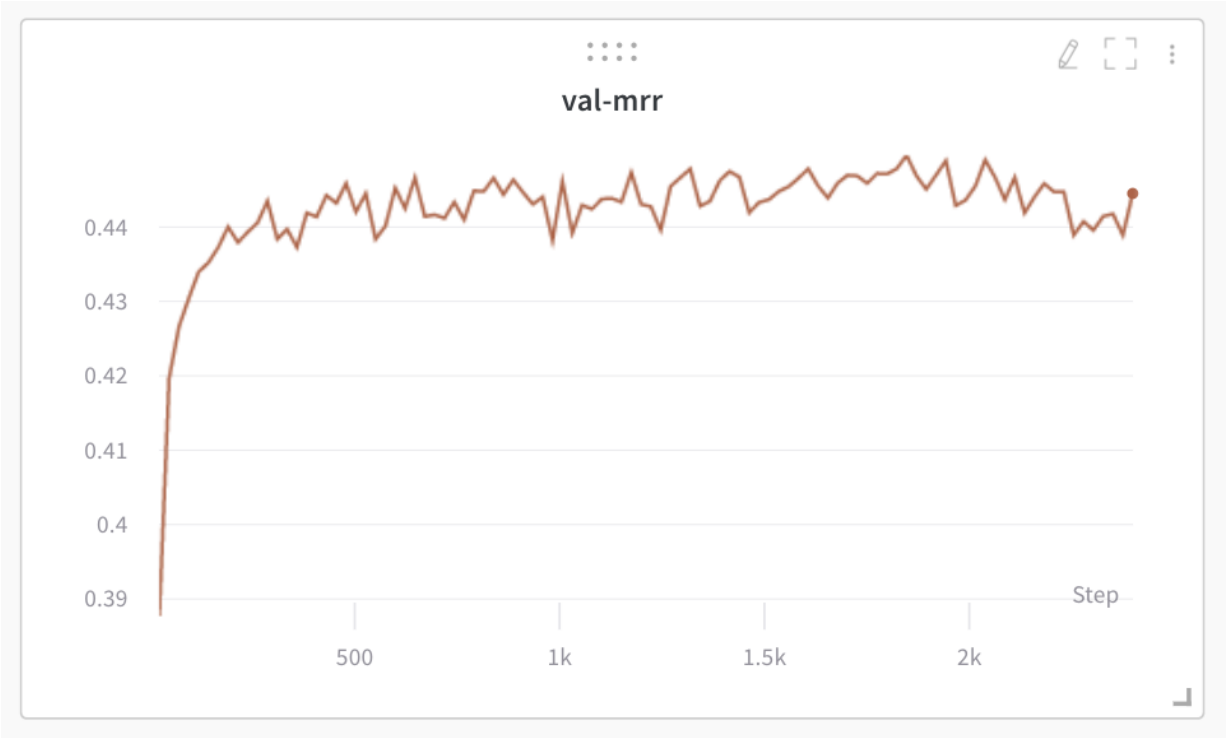

Figure A.7: Code2Vec validation MRR over each train step

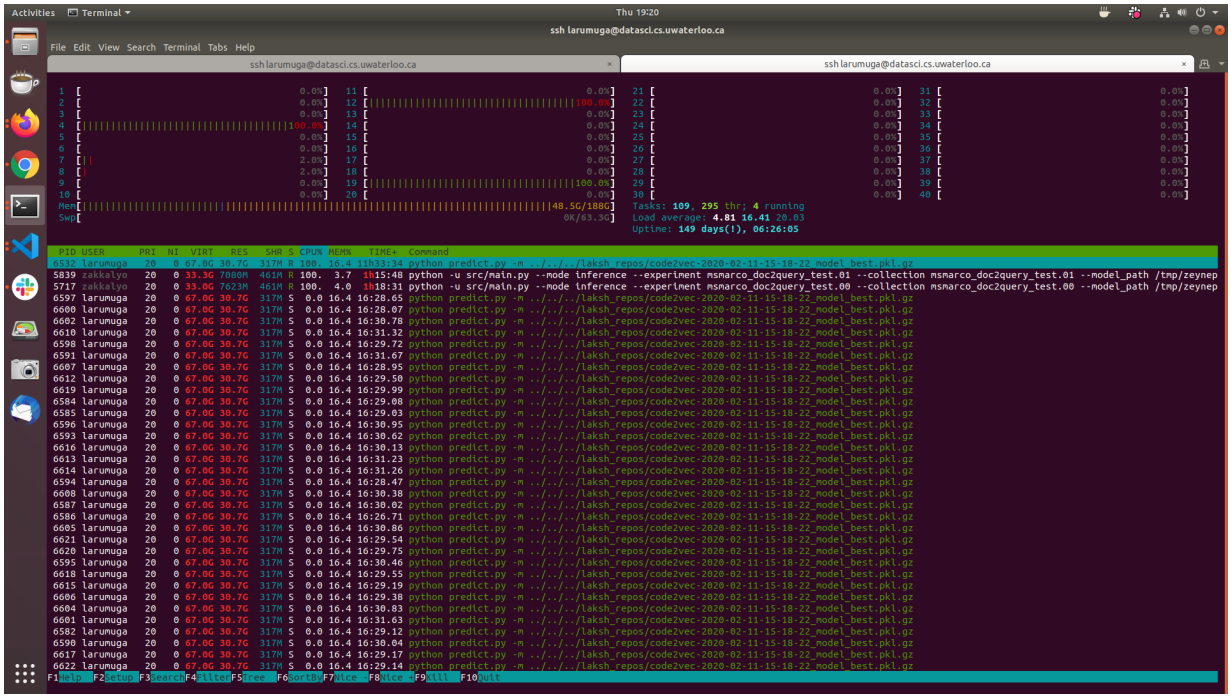

Figure A.8: Resource utilization during model predictions