UC Irvine
UC Irvine Electronic Theses and Dissertations

Title
Code Search using Code2Seq

Permalink
https://escholarship.org/uc/item/8418w1c9

Author
Nagar, Aishwariya Rao

Publication Date
2021

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

Code Search using Code2Seq

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Software Engineering

by

Aishwariya Rao Nagar

THESIS Committee:
Professor Cristina V. Lopes, Chair
Assistant Professor Iftekhar Ahmed
Associate Professor James A.Jones

2021

© 2021 Aishwariya Rao Nagar

TABLE OF CONTENTS

Page

LIST OF FIGURES **iv**

LIST OF TABLES **v**

ACKNOWLEDGMENTS **vi**

ABSTRACT OF THE THESIS **vii**

1 Introduction **1**
1.1 Motivation 1
1.2 Approaches for Code Search 2
1.3 Research Questions 4
1.4 Contributions and Chapters Overview 5

2 Background and Related Work **7**
2.1 Background 7
2.2 Related Work 8
2.2.1 Information Retrieval Based Approaches 9
2.2.2 Deep Learning Based Approaches 11

3 Design and Implementation **15**
3.1 Dataset 15
3.1.1 Data Collection 15
3.1.2 Data Preprocessing 16
3.2 System Overview 16
3.3 Comment Generation using Code2Seq 17
3.3.1 Abstract Syntax Trees 17
3.3.2 Feature Extraction 19
3.3.3 Model Architecture 20
3.3.4 Model Hyperparameters and Training 21
3.4 Information Retrieval 22
3.4.1 Indexing Source Code 22
3.4.2 Querying Source Code 23

4 Results **24**
4.1 Evaluation Metrics 24
4.1.1 Automatic Comment Generation Metrics 25
4.1.2 Precision Study 26
4.2 Experimental Results 28
4.2.1 Comment Generation 28
4.2.2 Information Retrieval Precision Study 28
4.2.3 Analysis of Results 30

5 Discussion **34**
5.1 Summary 34
5.2 Research Questions 35
5.3 Challenges and Threats to Validity 36

6 Conclusion and Future Work **39**
6.1 Conclusion 39
6.2 Future Work 40
6.2.1 Dataset Experimentation 40
6.2.2 Programming Languages 41
6.2.3 Improvements to Information Retrieval Technique 41
6.2.4 Comprehensive Evaluation of Code Search 42

Bibliography **43**

LIST OF FIGURES

Page

3.1 System Pipeline for Semantic Code Search 17
3.2 Code Snippet to Calculate Sum of All Elements in an Array using For Loop 18
3.3 Code Snippet to Calculate Sum of All Elements in an Array using While Loop 18
3.4 Code Snippet Example for Calculating the Sum of Two Numbers 19
3.5 AST Representation of figure 3.4 19
3.6 Code2Seq Model Architecture 21

LIST OF TABLES

Page

4.1 Table of Queries Used in Precision Study 27
4.2 Precision, Recall and F1-Score for Automatic Comment Generation 28
4.3 ROUGE Scores for Automatic Comment Generation 28
4.4 Comparison of Ground Truth Comments and Comments Predicted by Code2Seq 29
4.5 Comparison of Ground Truth Comments and Comments Predicted by Code2Seq 29
4.6 Top-5 Retrieved Code Snippets with Corresponding Cosine Similarities for the "copy paste data from keyboard" User Query 31
4.7 Top-5 Retrieved Code Snippets with Corresponding Cosine Similarities for the "remove problem marker from resource" User Query 32

ACKNOWLEDGMENTS

I would like to thank my thesis advisor and committee chair, Professor Cristina V. Lopes, for being my advisor and mentor throughout my Master's degree. Her guidance, knowledge, and unwavering support proved to be invaluable. I am also extremely grateful for Farima's insights and valuable feedback along the way. I would also like to thank the members of my committee, Associate Professor James A. Jones and Assistant Professor Iftekhar Ahmed for their patient advice, encouragement and counsel.

ABSTRACT OF THE THESIS

Code Search using Code2Seq

By

Aishwariya Rao Nagar

MASTER OF SCIENCE in Software Engineering

University of California, Irvine, 2021

Professor Cristina V. Lopes, Chair

The rapid development of software has led to the existence of a number of large, complex and swiftly growing codebases consisting of thousands of source code files. Therefore, the process of searching for code that performs a particular function has become an inherent part of the process of software development today. Developers often use general purpose search engines like Google or Q&A sites such as Quora and StackOverflow to search for relevant examples, which are not dedicated specifically to code search. In addition to this, code that is proprietary to a particular company or organization will not be available on these public platforms. In order to address these challenges, various semantic code search approaches based on information retrieval and deep learning techniques have been proposed which allows a user to search a code repository using natural language queries. However, information retrieval based code search systems rely on keywords and may not return relevant results if the query keyword is not present in the search documents. Deep learning approaches are able to retrieve code snippets that are similar to the user query even if the exact keywords aren't present but they treat source code as natural language and do not take into account the intrinsic semantic and syntactic information of source code. In this thesis, I aim to develop a hybrid semantic code search system that combines a neural model which leverages the syntactic properties of software artifacts to generate comments that automatically summarizes the function of the code snippet with information retrieval that returns methods that are

similar to the user query by computing the cosine similarity between the query input vector and the machine generated comment vectors present in the search corpus. Specifically, I use Code2Seq which represents a code snippet as an aggregation of individual paths in its Abstract Syntax Tree and learns the relevance of paths using attention in order to generate the target comment sequence. The code snippets along with the automatically generated comments form the IR search space which is used to find and retrieve code snippets that are relevant to the user query. The dataset, preprocessing steps involved, system design and implementation details are discussed in depth. Finally, the proposed system is evaluated through a precision study which shows that the top result returned is relevant to the user query 40% of the time.

Chapter 1

Introduction

1.1 Motivation

The existence of large, complex and rapidly growing code repositories both in open-source communities as well as organizations contain a plethora of syntactic and semantic information that can be used to develop a range of tools to boost developer productivity. The process of searching for similar code snippets or examples of how to execute a particular coding task is much more efficient than going through the documentation and source code of various APIs before using them. Developers often rely on search engines such as Google, Q&A platforms such as Quora and StackOverflow as well as technical blog posts for comprehension and assistance. These public resources cater to novice as well as experienced programmers who have questions pertaining to how a certain API is used, a programming language's syntax, examples of how to write code that performs a particular task in a particular programming language etc. These resources are extremely helpful as somebody else might have already faced the same road-block before and found a solution and written a blog post about it or posted a question on one of these platforms that was answered by someone else. However,

as useful as these resources are, they do not always have the answer that the developer is looking for. Additionally, code repositories of organizations are particularly challenging as the APIs and libraries used may be proprietary to the company and cannot be discussed on public platforms and an internal, company-specific Q&A forum might not be available.

Since code repositories of complex projects usually contain thousands if not millions of lines of code, code snippets that closely match the developer's intention are likely to exist somewhere within the source code. The objective of code search is to retrieve code snippets from a large code repository that is relevant to what the programmer plans to do, where the intention or goal of the developer is expressed in natural language. [26]. In theory, these code repositories should be well documented with clear, accurate and succinct comments which allow a developer to correctly ascertain what the program is doing without having to go through the entire source code. These comments can then be used to develop an application that is able to search for code snippets that closely match the developer's natural language query quickly and easily. However in practice, such summaries are too laborious, expensive and time consuming to produce manually. As a result, comments are often missing, inaccurate or out-of-date. In addition to this, if a code snippet is well documented, the comments need to be updated and maintained whenever the code snippet is modified in any way. The challenge here is to find and retrieve snippets in raw, unannotated or inaccurately annotated source code in an efficient manner.

1.2 Approaches for Code Search

Existing code search systems use annotated code corpora combined with information retrieval or deep learning based techniques to learn and compute the semantic similarity between code snippets and text query inputs. Information retrieval based search engines work by treating code snippets as text documents and indexing them in a database using keywords. This

database is searched to find indexes that match the text query and the corresponding code snippets are returned as ranked results. One of the key issues with code search engines based on Information Retrieval is that user queries often contain terms that do not exist in the source code document corpus. Thus, queries that do not contain the exact keywords present in the corpus will not have any matches in the database and no results will be returned even if they exist in the database.

In order to address the drawbacks of IR based code search engines and improve their performance, various deep learning approaches were proposed. The main idea behind these approaches is the concept of embedding code snippets and comments in a shared vector space such that the corresponding code-comment vectors are mapped close together and calculating the similarities between the code snippet - comment vectors and the query vector using a similarity metric such as cosine distance and returning the most similar code snippets to the user. This allows the system to learn the semantic similarity between the comments and queries and is able to find relevant code snippets even if the exact keywords aren't used in the input query.

While this is useful, both these approaches do not account for mismatches between code and its corresponding documentation. Also, existing approaches treat code as natural language and do not leverage the rich syntactic information present in it. In addition to this, unannotated code snippets present in the source code cannot be searched and retrieved since there is little to no overlap between the keyword search terms and source code tokens.

In this thesis, I address these problems by combining deep learning with information retrieval in order to develop a semantic code search system. A deep learning based neural network model is used to automatically generate high-level natural language descriptions that summarize the overall function the code snippet directly from the source code. Specifically, I use Code2Seq [1] created by Alon et al. which leverages the syntactic information present in source code by taking an aggregated bag of paths over the code snippet's Abstract Syntax

Tree as input instead of treating source code as a bag of tokens. The descriptions generated by Code2Seq are then used to develop an information retrieval based code search engine that is able to find and retrieve code snippets that are documented to be performing a task that closely matches the developer's query input using cosine distance.

1.3 Research Questions

The main goal of this thesis is investigate if the semantic and syntactic information present in source code can be leveraged to improve upon existing semantic code search systems.

RQ1: *Can automatically generated comments for code snippets be used to retrieve relevant code snippets?*

Existing deep learning approaches for code search use corpora that contain code-comment pairs in order to train a model to learn the semantic similarity between source code and natural language in order to retrieve relevant results. However, these models rely on annotated source code and cannot search for relevant code snippets in undocumented code. Iyer et al. [17] present their approach for code search using automatic comment generation along with their results. This approach treats code as natural language and does not leverage the syntactic structure of programming languages. The hypothesis behind this research question is that the comments generated for code snippets by a neural model will be accurate and up to date can be used to retrieve all relevant code snippets.

RQ2: *Will the performance of the code search system be affected by the size of the retrieved set of code snippets?*

The code search system presented in this thesis works in two stages to retrieve code snippets that are relevant to a user query - automatic comment generation for code snippets and similarity matching between the query input and all the machine-generated comments in the

database. Since the relevance of a code snippet for a particular task is subjective and can vary according to the number candidate code snippets present in the database along with the machine generated comments, I evaluate the Top-1, Top-2 and Top-5 precision of the system through a precision study.

RQ3: *Will the performance of this code search system that leverages the syntactic and structural information present in source code be comparable to state-of-the-art methods that treat code snippets as natural language?*
Deep learning models that are used for code search are designed for NLP tasks and treat source code as natural language. However, natural language and programming language differ in many ways. While natural language tends to be ambiguous and unstructured, programming languages have their own respective grammars and source code written in them are unambiguous and structured. Also, natural language corpora usually consist of a limited vocabulary of a few thousand words that have the highest frequency. Words outside of this vocabulary are marked as UNK. This is an effective strategy because words outside this vocabulary are too rare to be considered. However, in the case of code corpora, the vocabulary consists of keywords, operators, and identifiers. A codebase that is used to build a probabilistic model will likely have a vast amount of out-of-vocabulary identifiers and alot of important information is lost. Therefore, a model that is able to take advantage of the semantic and syntactic information present in source code by representing a code snippet using its Abstract Syntax Tree has the potential to perform better than existing models that treat code as a bag of tokens.

1.4 Contributions and Chapters Overview

The first contribution of this thesis is adapting the existing Code2Seq [1] model for the task of automatically generating comments that summarize source code fragments.The second

contribution is building a semantic code search engine using the comments generated by the Code2Seq model. The third contribution is evaluating the relevance of the top N code snippets retrieved by the system through a precision study.

The thesis is divided into the following chapters. In chapter 2, the background and existing approaches for code search has been discussed. Chapter 3 contains details of the dataset and preprocessing steps followed, the overview of the system pipeline along with implementation details of the two stages in the code search system namely automatic comment generation using Code2Seq and information retrieval. In chapters 4 and 5, the results, analysis and threats to validity are discussed. Finally, the thesis is concluded in Chapter 6 with potential directions for future work.

Chapter 2

Background and Related Work

2.1 Background

This thesis is a follow-up work to Code2Seq [1] which is a deep learning model that predicts appropriate method names for code snippets and the dataset provided by the authors of the DeepCom paper [14] which have been discussed in detail in later sections. I evaluated the application of Code2Seq for the task of automatically generating comments for Java source code methods in order to develop a semantic code search system that utilizes these machine generated comments to find and retrieve code snippets that are relevant to a natural language query input.

Alon et al. created Code2Seq [1], a neural model that uses the syntactic information present in source code to produce better code embeddings. They achieve this by representing a code snippet as a collection of paths in its abstract syntax tree and using attention to select which paths to decode, one at a time. The authors use this model to predict method names for a given code snippet based on the notion that code snippets that perform similar tasks will have similar code embeddings. They evaluate their model by training it on 3 distinct

datasets consisting of 700K, 4M and 16M examples respectively and show that the model is able to predict appropriate method names for examples that were not present in the training set with an F1-score of 59.19 for the largest dataset. While they only used code2seq for the prediction of method names, the authors state that it can be used for a range of applications such as code documentation, summarization etc.

Prior to Code2Seq, the same authors designed Code2Vec [2] which employs a neural network to produce a single vector that represents the entire code fragment. They do this by extracting the Abstract Syntax Tree (AST) of a given code snippet, representing each path in the AST as a distributed vector and finally aggregating all the path vectors using attention to produce a continuous distributed representation of the code fragment body. This system was used to predict the probability of each target method name given the code vector.

2.2 Related Work

Various code search approaches have been proposed. These can broadly classified into two categories-
(1) Information Retrieval based techniques that preprocess and index source code in databases to facilitate efficient exploration and retrieval of relevant code snippets.

(2) Deep learning based approaches that embed code along with their corresponding natural language descriptions into vectors and then use a vector similarity measure such as cosine similarity to find and retrieve code snippets that have a semantic correlation with the query sentence.

2.2.1 Information Retrieval Based Approaches

Chatterjee et al. [8] describe a flexible code search system called SNIFF (SNIppet for Free-Form queries) which retrieves a small set of code snippets that match a free-form English text query. SNIFF utilizes the documentation present in the source code of APIs and library methods to annotate publicly available Java code. Each API or library method call in a code snippet is appended with the corresponding Javadoc description which adds relevant and meaningful comments to otherwise undocumented source code. This annotated source code is then indexed in a database. A free-form query to SNIFF searches the database and retrieves chunks of code that are closely related to the query input. The system finally constructs the most pertinent code snippet using type-based intersection to retain the most important and common parts of the candidate code chunks.

JSearch[27] is a scalable code search tool that supports various types of query searches over thousands of source code files in a repository. Java source code files are first preprocessed by replacing all the variable names with class names and then parsing them using an AST parser to extract syntactic elements such as classes, method names, comments, imported libraries, return types etc. Indexes are then created for each of these fields using Lucene to facilitate flexibility in searches. Given a query input string, the tool returns results ranked according to their relevance based on keyword matches between the text string query and source code indexes.

Bajracharya et al. [3] developed a code search engine that extracts and stores structural information present in source code available in open-source repositories to rank results and allow for flexible search queries at the function class, algorithm or property level. The infrastructure was built using a relational database that stores program entities and indexed keys along with Lucene to maintain a mapping between the keys and entities. Keywords present in a search query are matched to keywords stored in Lucene and the set of matching

keys along with the corresponding entities are used to fetch the source code files which are finally ranked using a novel ranking scheme and returned to the user.

JIRiSS [23] (short for Information Retrieval based Software Search for Java) is a code search tool that allows users to search for fragments or code at the class or method level using natural language queries. A corpus containing structural information, comments and identifiers is first created by decomposing the source code of a project along with associated documentation into text documents. This corpus is then projected onto a semantic search space using Latent Semantic Indexing. The user query input is mapped as a document into the same semantic search space and the similarity between the query document and every document in the search space is computed. The ranked results of the query are returned as methods or classes. This system also supports fragment-based searh, automatic spell checking of input queries and word suggestions if the query word is not present in the software vocabulary.

Holmes et al. [13] describe the Strathcona tool which retrieves code snippets present in a repository that match the structure of the code being written. The client portion of the system automatically extracts structural information of the code being developed by the user in order to form the query when the user requests relevant example code snippets to be fetched from a repository. The client sends this structural context description to the server which stores the repository that needs to be explored. The server then searches the repository using a set of heuristics in order to match the structure of the query code snippet and returns the code examples with the best structural context matches to the user.

Jiang et al [18] combined information retrieval and supervised learning to propose a system called ROSF (Recommending cOde SNippets with multi-aspect Features). First, a code snippet corpus is created from a large collection of Android projects. When a free-form text query is provided to the system, it creates a candidate set with as many relevant code snippets as possible by searching the code snippet corpus using an information retrieval technique. In the next stage, the candidate set of code snippets, the probabilities for different relevance

scores between the candidate code snippets and the query text are predicted by a learned multi-class logistic regression model. The candidate code snippets are re-ranked according to these probability values and the top K results are returned to the user.

2.2.2 Deep Learning Based Approaches

Sachdev et al.[26] proposed a code search tool for large code repositories called NCS (Neural Code Search) that finds and retrieves code snippets that are related to the search query directly from source code. Their tool employs an unsupervised technique for learning vector representations which only uses embeddings derived from code examples. The model treats the code snippet as natural language and generates token embeddings for code snippets as well as query input using fastText.[5] The code snippet embedding is then formed by summing over these unique code token embeddings using TF-IDF weights. The search query embedding is formed by summing over the individual query token embeddings. Finally, the code snippet vectors with the smallest cosine distance to the search query vector are retrieved.

Cambronero et al.[6] extended the existing NCS model by adding a supervised learning component to it. The training data consists of code snippet - docstring pairs which are are tokenized and embedded using fastText [5]. The code and docstring tokens are then combined using attention to generate the code snippet and docstring sentence embeddings respectively. The query input embedding is generated by computing an average over the query token embeddings. Lastly, the code vectors that have the highest cosine similarity to the query input vector are fetched. Their findings show that a simple model with supervision outperforms NCS as well as more sophisticated deep learning models. Their experimental results also prove that an idealized training corpus with supervision can deliver great results.

Iyer et al. [17] employ a deep neural network for supervised learning. Their domain agnostic CODEnn model, short for Code Description Embedding Neural Network, uses an LSTM

network with attention on the code snippets to model the conditional distribution of the natural language summary and generate it one word at a time. This model is trained on a dataset consisting of C# and SQL code snippets they collected from StackOverflow. CODEnn first extracts the method name, API call sequence and a bag of code tokens from the raw code snippet. The method name and API call sequence are fed into two separate biLISTM networks while the bag of code tokens is fed into a feed forward network. The final code embedding is obtained by combining these three vectors which is then fed as input into a dense neural network which generates the natural language summary, one word at a time. The bag of docstring tokens is used to generate the query input embedding which is then compared with all the summary vectors present in the retrieval set. The code snippets are ranked according to the cosine distance between the query input vector and the summary vectors and the top ranked code snippets are retrieved.

Husain et al. [15] developed another code search tool called SCS (short for Semantic Code Search) by embedding code and natural language in a shared vector space. Their approach is divided into four distinct steps. In the first step, python code snippets and their corresponding docstrings are fed as input into a seq2seq model which generates code summaries. Secondly, a separate language model is trained to embed query input tokens. The seq2seq model used to generate code summaries is then fine-tuned to generate summary embeddings that are mapped to the same vector space as the code embeddings such that the distance between the code vector and summary vector for a particular code snippet is minimal. Finally, a search index of code vectors is created which is used to retrieve the nearest neighbors to the query input vector in the shared code-summary vector space.

Heyman et al. [12] present an approach for code search which retrieves code snippets annotated with succinct intent descriptions using natural language query inputs. Two discrete embedding models are trained independently to learn the similarity between the intent descriptions present in the code snippet and query inputs as well as between the code snippets

and queries. The similarity between queries and descriptions are learnt by the Universal Sentence Encoder (USE) [7] while the similarity between code snippets and queries is learnt by the NCS model [26]. The code corpora was augmented by inserting description word tokens in the middle of source code tokens as well as appending the description word tokens to the code tokens. This was done to force the code tokens and description words that appear in the same code snippet closer together in the training corpus in order to make their embeddings more similar. Finally, an ensemble model computes the similarity between the query input and code snippets as a linear combination of the cosine similarities of the Universal Sentence Encoder and Neural Code Search model outputs.

Husain et al. [16] followed earlier work and implemented a code search system by jointly embedding code and natural language vectors in a single vector space. Their model architecture consists of two distinct encoders, one for code and one for natural language respectively which are trained to map the inputs into a shared vector space such that the code and corresponding natural language descriptions are close to each other. Finally, an embedding is generated for the query input and the code snippet neighbours that are closest to the query vector are returned. Code search experiments were also run on a popular search engine called ElasticSearch by creating an index using the function name and code snippet for every function in the code corpus. The authors also contributed a dataset called CodeSearchNet which was created from open-source projects from GitHub and consists of 6 million functions across 6 programming languages to enable researchers to develop and evaluate novel approaches in code search and retrieval systems.

Yao et al. [32] developed a reinforcement learning based framework called CoaCor (short for Code Annotation Code Retrieval) where a model is trained to generate natural language annotations for code snippets which can then be used by a code retrieval model to find and fetch relevant code snippets. First, a code retrieval model is trained on code snippet - natural language comment pairs. A code annotation model is then trained to predict the annotation

for a code snippet as a sequence of natural language tokens and receives a reward from the code retrieval model based on how effectively the machine generated code annotation is able to discern the corresponding code snippet from a candidate set. Finally, another code retrieval model is trained on the code snippet - machine generated annotation pairs. At test time, given a natural language query, the code snippets are ranked by combining the scores of both the code retrieval models. Their results show that machine generated annotation for code can substantially boost the performance of code search systems.

Feng et al. [10] implemented a bimodal pre-trained model called CodeBERT which learns the semantic relationship natural language and programming languages such as Java, PHP, Go, Python, Javascript and Ruby. A multi-layer bidirectional Transformer model is trained in a manner that is similar to multilingual BERT where a one pre-trained model is trained on the CodeSearchNet dataset [16] consisting of code snippets across 6 different programming languages without any indication of the programming language for the code snippet inputs. CodeBERT is trained with biomodal data-points which consist of natural language-code snippet pairs as well as unimodal data-points consisting of only function level natural language comments and only code snippets. For code search, the language model is fine-tuned for each programming language and the aggregated hidden representations of the code snippet - comment pair is used to compute the semantic similarity between the code and natural language query input.

Chapter 3

Design and Implementation

3.1 Dataset

3.1.1 Data Collection

The dataset from DeepCom [14] is used for training and evaluation. It is a Java corpus built from 9,714 open source projects from GitHub. The training set consists of 470,486 examples while the test and validation sets consist of 58,811 examples each. The dataset contains 3 json files, for train, test and validation respectively.

This dataset is especially challenging for two reasons:
(1) the Javadoc comments are extracted from the code using Eclipse's JDT compiler and they were not checked by humans to ensure its correctness.
(2) The dataset is also orders of magnitude smaller than the dataset used for the task of predicting method names by Alon et al. [1].

3.1.2 Data Preprocessing

The json files containing the code-comment pairs are first converted to individual raw .java files in order to extract the AST. The Javadoc comment associated with each method is written to a text file with the file name of the corresponding code snippet. The Javadoc comment text files are then preprocessed to remove HTML tags and converted to lowercase. Comments with 0 tokens or greater than 40 tokens are excluded since Code2Seq is trained to predict short method names and cannot learn extremely long sequences of text. Code snippets with duplicate method names are removed because the comment for each code snippet is mapped to the AST using the method name and each comment had to be associated with a unique method name in order to be correctly mapped.

3.2 System Overview

This thesis focuses on developing a semantic code search system for code snippets by combining an encoder-decoder based neural model (Code2Seq [1]) with traditional information retrieval techniques in order to fetch relevant code snippets that are closest to the free-form text query input. The system can be divided into two distinct parts. First, the code2seq model generates comments for all the code snippets in a repository. Secondly, the task of code search is treated as an information retrieval problem where the candidate code snippets along with the corresponding machine generated comments are stored in a database and code snippets whose comments are semantically similar to the natural language query are retrieved. This step is described in detail in section 3.4.

Section 3.3 describes the first part of the system- comment generation using Code2Seq [1] in detail. This section contains foundational concepts required in order to comprehend how the Code2Seq [1] model is used for this task - Abstract Syntax Trees, feature extraction,

model architecture, model training and hyperparameters used. Section 3.4 comprehensively describes how code snippets relevant to the text query are retrieved using traditional information retrieval techniques. An overview of the system pipeline is provided in Figure 3.1.

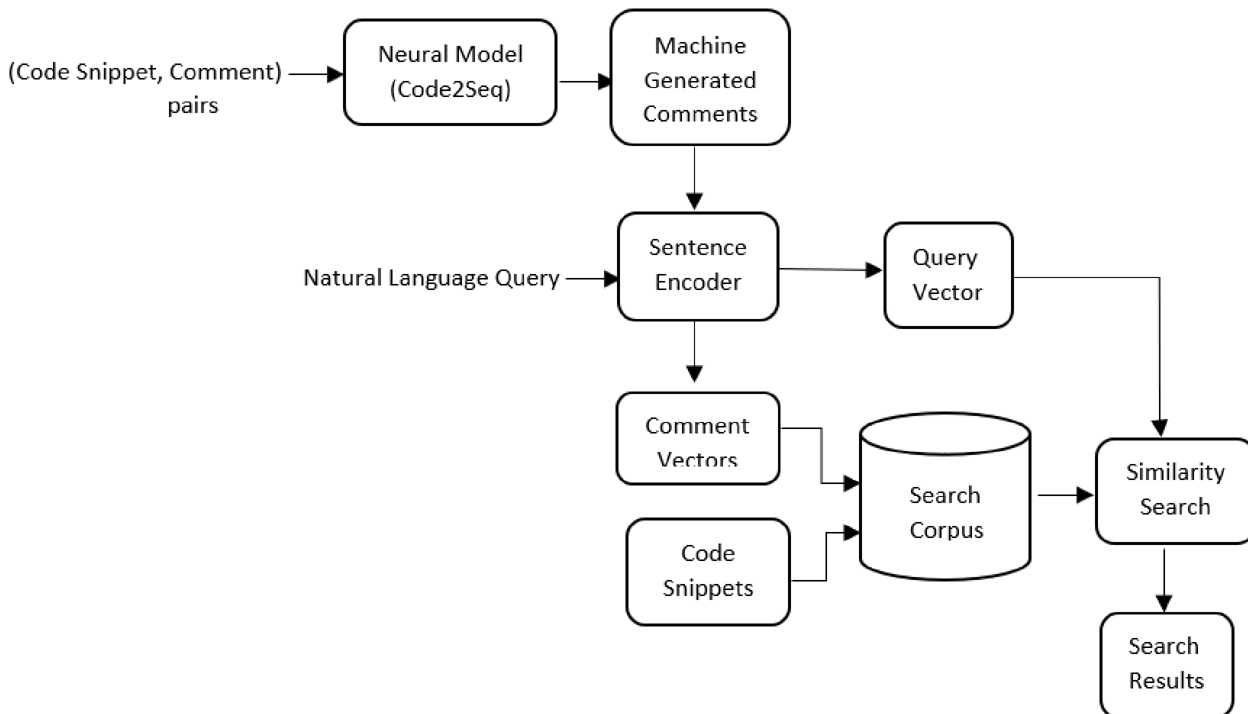

Figure 3.1: System Pipeline for Semantic Code Search

3.3 Comment Generation using Code2Seq

3.3.1 Abstract Syntax Trees

In this thesis, an abstract syntax tree (AST) refers to the basic tree representation of the abstract syntactic structure of source code written in the Java programming language [31]. The leaf nodes in an AST are called terminals and usually refer to user-defined identifiers and names (such as float, num etc.) in the code snippet. Syntactic structures in the code such

```
static int f(int[] arr)
{
   int sum = 0;
   int i;
   for (i = 0; i < arr.length; i++)
     sum +=  arr[i];

   return sum;
}
```

Figure 3.2: Code Snippet to Calculate Sum of All Elements in an Array using For Loop

```
static int f()
{
   int sum = 0;
   int i=0;
   while(i < arr.length){
     sum +=  arr[i];
     i++;
  }
   return sum;
}
```

Figure 3.3: Code Snippet to Calculate Sum of All Elements in an Array using While Loop

as loops (ForStmt denotes a for loop) and variable declarations (VarDec) are represented by non-leaf terminals and are called non-terminals.

An AST is a very effective way of encoding source code. Consider the two Java methods in Figure 3.2 and 3.3. Both these methods find the sum of all the elements in an array. These methods are implemented differently but perform the same function. If the vector representations for these code snippets are generated using a bag of tokens, the recurring pattern between the two methods that suggests a common method name won't be considered. However, when the syntactic structure of the two methods are observed, it can be seen that the ASTs only differ in a single node - a while statement is used in one versus a for loop in the other. Therefore, an AST is able to normalize a lot of variance that occurs in source code and a model that is able to leverage this can generalize much better to examples not seen during training.

The Code2Seq model decomposes code snippets to a set of paths over their ASTs and learns the representation of the code snippet using these path embeddings. This is described in detail with an example in Section 3.3.2.

3.3.2 Feature Extraction

After the preprocessing steps in section 3.1 were performed, the AST for each code example is extracted using the Java extractor provided by Alon et al. [1]. The methodology used to extract the paths from a code snippet's AST is explained below with an example.

```
static int f(int m, int n)
    {
      return m+n;
    }
```

Figure 3.4: Code Snippet Example for Calculating the Sum of Two Numbers

Consider a code snippet (figure 3.4) that calculates the sum of two integers m and n whose AST (shown in Figure 3.5) is constructed by the Java extractor. Then, every pairwise path between two terminal leaf tokens is traversed and represented as a string of sequences, where each sequence contains the AST nodes, connected by up and down arrows which denote the up or down link between connected nodes in the tree.

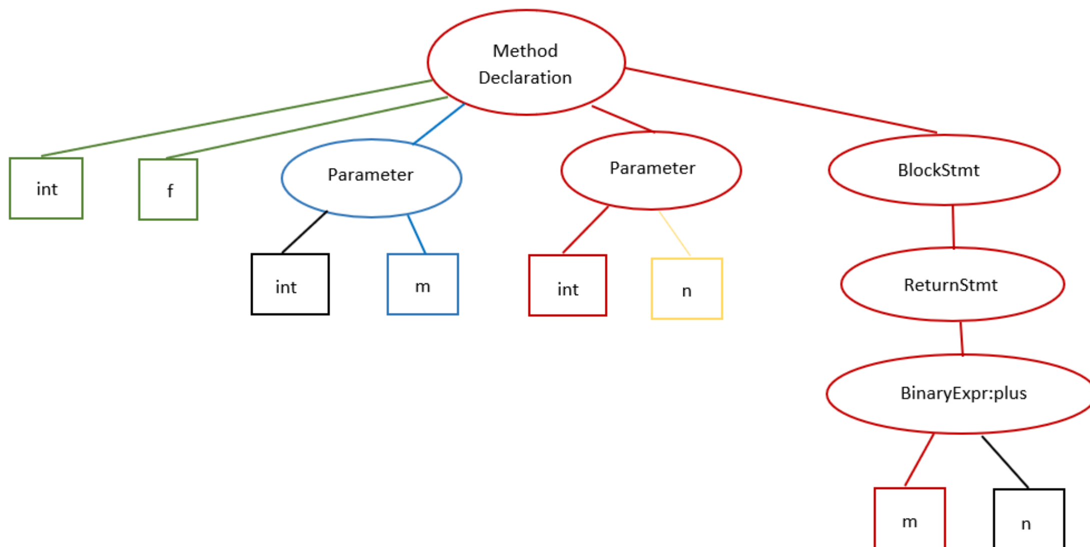

Figure 3.5: AST Representation of figure 3.4

Examples of some paths extracted from the above AST would be:

- (int, parameter ↑ Method Declaration ↓ BlockStmt ↓ ReturnStmt ↓ Binary Expr:plus, m) denoted by the colour red in Figure number 3.5.
- (int, Method Declaration, f) denoted by the colour green in Figure number 3.5.

A tuple is then constructed for every pair of AST leaf tokens and the path connecting them where each token is split to subtokens using the "|" character. This tuple is called a path-context. Code snippets that could not be parsed and code examples with empty method bodies are dropped by the extractor. This step generated a single text file for train, test and validation respectively where each row is an example. Each example is a space-delimited list of all the path-contexts in a code snippet's AST separated by commas. The comments are finally inserted at the beginning of each example as a set of tokens, also separated by the "|" operator. The final training data consisted of 142,550 examples while the test and validation data consisted of 8325 and 9000 examples respectively.

3.3.3 Model Architecture

The model presented by Alon et al. [1] is used for this task. This model is based on the traditional Encoder-Decoder Architecture used in NMT problems.

A given code snippet is represented as a set of compositional paths over its Abstract Syntax Tree. Paths in the AST are sampled and encoded by a BiLSTM to create a vector representation for each path along with its values in the AST separately instead of reading the input as a flat sequence of tokens. The decoder then attends over the encoded AST paths while generating the target comment sequence. At each decoding step, the probability of the next target token depends on the previously generated token. A detailed explanation of the model can be found in the original paper. [1]

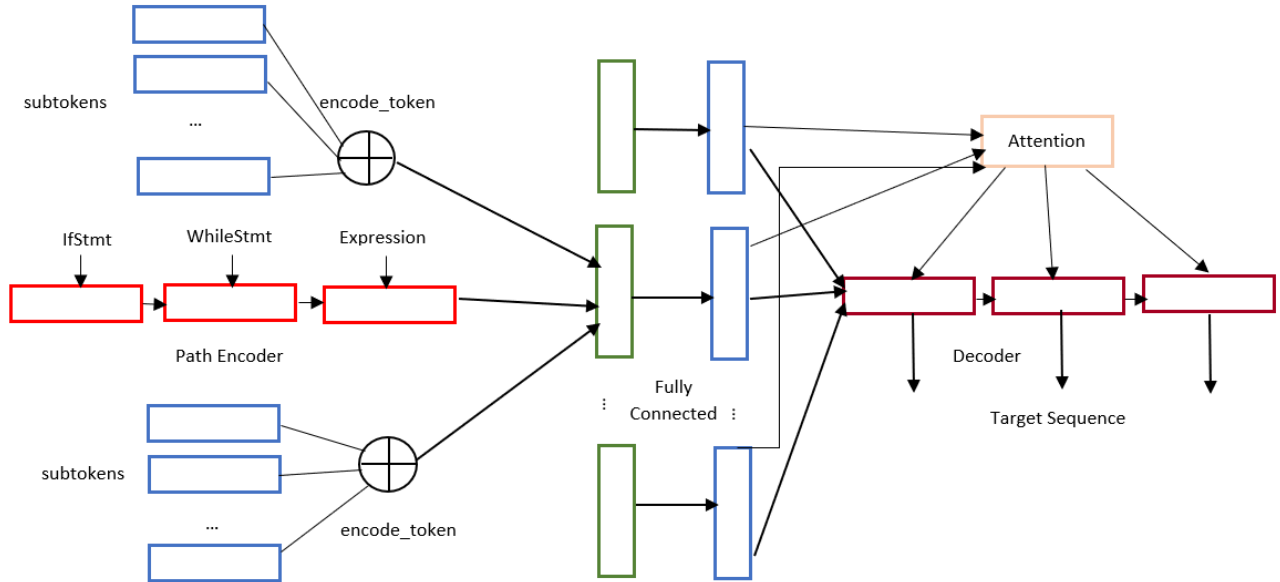

Figure 3.6: Code2Seq Model Architecture

3.3.4 Model Hyperparameters and Training

The Code2Seq model is trained on Google Colab Pro with high RAM setting. The batch size is reduced to 128 in order to fit into memory. The encoder BiLSTM that encodes the AST paths consists of 256 units and a recurrent dropout of 0.5 is applied on each LSTM. The decoder LSTM consists of 2 layers, each of size 512 in order to support the generation of longer target sequences. The maximum length of the target sequence is set to 40 in accordance with the maximum number of ground truth comment tokens in the training dataset. This configuration resulted in 69M trainable params. The model is trained for 39 epochs or until there is no improvement after 10 iterations.

3.4 Information Retrieval

3.4.1 Indexing Source Code

The test set consisting of 8300 code snippets along with the corresponding machine generated comments form the candidate set of source code fragments that is searched in order to find and retrieve code snippets that are relevant to the user query. In order to find methods that match the user's objective, the natural language comments in the candidate set along with the query input text need to be converted to vectors before they can be compared using a similarity measure such as cosine distance.

Popular text embedding models such as Word2vec [21] and Glove [22] convert individual words in a sentence to a vector. But while embedding a sentence, the words along with the context in which they are used needs to be captured by the vector which isn't possible using Word2Vec or Glove. In order to address this, Google proposed the "Universal Sentence Encoder" [7] which generates an a fixed-length 512-dimensional vector for each input sentence. The Universal Sentence Encoder model comes with two variations - one employs a transformer architecture while the other uses a deep averaging network (DAN). The two variants have a trade off between accuracy and amount of computational resources required. The transformer based model has higher accuracy but requires more computational resources while the DAN based model is computationally less intensive with lower accuracy. These models are pre-trained and can be downloaded from Tensorflow-Hub. In this thesis, the transformer encoder version is used to generate a sentence embedding for each comment in the candidate set.

The code snippets are indexed using the corresponding comment embedding and are stored in a Pandas dataframe [28] to simulate the indexed database that is used for efficient search and retrieval in IR based code search engines.

3.4.2 Querying Source Code

Before a search starts, all the code snippets in the candidate set are indexed using the corresponding comment embeddings. When a developer enters a search query such as "how to convert json object to csv", "how to open url in html browser", "prompt user to enter details" etc., it is expressed as a natural language sentence. The Universal Sentence Encoder model [7] is first used to embed the query sentence to generate a 512-dimensional vector that is of the same size as the candidate set comment embeddings. Then, the cosine similarity between the query sentence embedding and all the comment embeddings present in the candidate set is computed using equation 3.1. Finally, the top-K code snippets whose comment embeddings are most similar to the query input embedding are returned as results. The value of K is set to 5 in my experiments.

$$\cos(\mathbf{A}, \mathbf{B}) = \frac{\mathbf{A}.\mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} \tag{3.1}$$

Chapter 4

Results

4.1 Evaluation Metrics

This chapter showcases the experimental results of the semantic code search system when applied on the dataset obtained from the Deep Code Comment Generation paper [14].

Section 4.1 provides a brief overview of the evaluation metrics used in this thesis. Since the code search system first generates comments for code snippets which is then used for retrieval, the performance of the system heavily depends on how well the code2seq model performs on the task of automatic comment generation. Therefore, the comment generation model is first evaluated using precision and recall and then with ROUGE and BLEU which are metrics used to assess the performance of neural-machine translation problems in literature. The information retrieval based code-search system is then evaluated through a precision study which is described in detail in section 4.1.2.

4.1.1 Automatic Comment Generation Metrics

Ideally, the machine generated comments should be manually evaluated by human annotators, but given that manual evaluation is very difficult to scale, I adopted the measures used in literature - precision, recall, Rouge and BLEU which is measured over case-insensitive tokens. The basic idea behind these metrics is that the quality of the machine generated comments mainly depends on the words used to compose it along with the word overlap between the predicted and ground truth comments.

BLEU (bilingual evaluation understudy) [29] is a popular metric that is used to evaluate the quality of output text in neural machine translation problems. In this thesis, the translation of a code snippet written in the Java programming language to a natural language sentence (prediction of a comment that summarizes the function of the code snippet) is being evaluated. The idea behind BLEU is that the quality of a machine generated translation depends on how close it is to a professional human translation. This is done by using a modified form of precision to compare the predicted translation against the reference text. An n-gram is a sequence of words within a window of size n. BLEU compares the n-gram of the predicted translation with the n-gram of the reference text to count the number of matches, regardless of the positions at which these matches occur. Higher the number of matches between the predicted text and reference text, better is the machine translation. Generally, BLEU scores are based on an average of unigram, bigram, trigram and 4-gram precision [25]. However, BLEU is problematic for a number of reasons which is discussion in detail in section 4.4. This metric is used to compare the output of the code2seq model to the benchmark model described in the Deep Code Comment Generation [14] paper that performs the same task on the same dataset. The BLEU score was computed using SacreBLEU. [24] which is an open-source tool that can be used to predict a corpus level BLEU score between the reference file and predicted file.

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) [30] is a modification of BLEU that focuses on recall rather than precision.It consists of a set of metrics rather than just one. ROUGE-N looks at how many n-grams in the reference text show up in the predicted text. ROUGE reports the precision, recall and F-1 score for each value of N under consideration. In this thesis, ROUGE-1 and ROUGE-2 scores are reported. Additionally, ROUGE-L calculates the longest shared sub-sequence of tokens between the reference text and predicted output as a longer common sub-sequence indicates a higher similarity between the two.

BLEU and ROUGE scores complement each other and therefore, both were used in this thesis in order to evaluate the model in a comprehensive manner.

4.1.2 Precision Study

The evaluation of the retrieval of code snippets that are relevant to the user's query input is a challenging task as it is laborious and requires specialized manual effort. Popular information retrieval metrics such as Mean Reciprocal Rank (MRR) and Normalized Discounted Cumulative Gain (NCDG) cannot be applied to this task since ground truth data is required in order to determine the precision of the code retrieval system which can only be provided by humans.

In this thesis, an experiment-driven analysis of the system is conducted through a precision study involving three developers in order to determine the relevance of the retrieved code snippets. This is performed using 10 natural language queries obtained from the paper written by Bajracharya et al. [4]. The search space is a small corpus of 8300 code snippets obtained after pre-processing the test set [14]. The search queries used in this study are presented in table 4.1.

The top-5 queries with the highest cosine similarity values are first retrieved for each query.

Sl. No.	User Queries
1	copy paste data from clipboard
2	open url in html browser
3	track mouse hover
4	open file in external editor
5	prompt user to select directory
6	open dialog and ask yes no question
7	parse source string ast node
8	run job in ui thread
9	open external file
10	remove problem marker from resource

Table 4.1: Table of Queries Used in Precision Study

Each developer then manually examined these top-5 code snippets and answered "Yes" or "No" depending on whether they found the code snippet to be pertinent to the objective described in the query or not. After this, the answers provided by the three developers is examined and aggregated using the following rule - the answer to whether a code snippet returned by the system is relevant is "Yes" or "Y" if and only if 2 or more developers out of 3 answered "Y". Otherwise, the answer is "No" or "N".

The precision of the system is also determined by the aggregation rule followed. Farmahini-farahani et al. [9] evaluated the precision of eight different code clone detection tools through a precision study involving three judges. They reported the results obtained by taking the majority vote as well as by taking the unanimous vote. They observed a drop in precision for certain tools when it was calculated using the unanimous voting method. This gap indicates that the judges disagreed on the classification of some code clones and shows that precision experiments are highly dependent on what the judge perceives to be a clone.

As discussed earlier, the precision of the code retrieval system is calculated by aggregating the answers using the majority voting method. The results of this study are presented in section 4.2.2.

4.2 Experimental Results

4.2.1 Comment Generation

Table 4.2 shows the results for automatic comment generation using the Code2Seq model when evaluated on the test set consisting of 8300 code snippets unseen during training.

Evaluation Metric	Percentage Value
Precision	30.65
Recall	35.77
F1-Score	33.01

Table 4.2: Precision, Recall and F1-Score for Automatic Comment Generation

The model was also evaluated using ROUGE scores on the test set. The results are reported in table 4.3.

ROUGE Metric	Precision %	Recall %	F1-Score %
ROUGE-1	21.07	26.59	20.73
ROUGE-2	5.71	7.36	5.66
ROUGE-L	25.255	24.16	22.33

Table 4.3: ROUGE Scores for Automatic Comment Generation

In addition to this, the BLEU score for this model is 3.64 when evaluated on the test set.

Table 4.4 presents some of the results of the automatic comment generation task by comparing the ground truth comments for the test set with the comments predicted by Code2Seq.

4.2.2 Information Retrieval Precision Study

Table 4.5 provides the results for the precision study conducted in order to manually evaluate the precision of the code retrieval system. Columns labelled 1 through 5 indicate the order

Ground Truth Comment	Predicted Comment
accept role assignment event	handle role assignment request event
acquires the write lock if the lock is not available then the current thread becomes disabled for thread scheduling purposes and lies dormant until the lock has been acquired	write locks to all of the set of threads
add new module	adds a module to be used in the template
add text part without background from resources	create text builder
a version suitable for our viewholders	create a tag for a view id
a method to add a bookmark	create a new bookmark bookmark and user
a check to be used as method reference	returns true if this is a message

Table 4.4: Comparison of Ground Truth Comments and Comments Predicted by Code2Seq

Sl. No.	User Query	1	2	3	4	5	Top-1	Top-2	Top-5
1	copy paste data from clipboard	Y	Y	Y	N	N	Y	Y	Y
2	open url in html browser	Y	Y	N	N	Y	Y	Y	Y
3	track mouse hover	Y	N	Y	N	N	Y	Y	Y
4	open file in external editor	N	N	N	Y	N	N	N	Y
5	prompt user to select directory	Y	N	N	Y	N	Y	Y	Y
6	open dialog and ask yes no question	N	N	Y	N	N	N	N	Y
7	parse source string ast node	N	N	N	N	Y	N	N	Y
8	run job in ui thread	N	Y	N	N	N	N	Y	Y
9	open external file	N	N	N	Y	N	N	N	Y
10	remove problem marker from resource	N	N	N	N	Y	N	N	Y

Table 4.5: Comparison of Ground Truth Comments and Comments Predicted by Code2Seq

in which a code snippet was retrieved i.e 1 refers to the top code snippet with the highest cosine similarity to the query vector that was retrieved by the system, 2 refers the code snippet with the second highest cosine similarity, 3 refers to the code snippet with the third highest cosine similarity and so on.

The Top-1 column refers to whether the top code snippet retrieved is relevant to a particular query. ("Y" in column 1). The Top-2 column indicates whether atleast one out of the top two code snippets returned by the system is relevant or not. ("Y" in either column 1 or column 2 or both). The Top-5 column specifies whether one or more code snippets out of the top-5 results returned by the system was relevant to the user query or not. ("Y" in one

or more columns from column 1 through column 5).

From the table, it can be seen that the Top-1 precison of the system is 40% , Top-2 precision is 50% and Top-5 precision is 100% when calculated using the majority voting method.

4.2.3 Analysis of Results

This section qualitatively analyses the results from section 4.2.2 using two examples.

Consider the query "copy paste data from clipboard". Table 4.5 shows that the top-3 code snippets returned by the system were considered relevant by atleast 2 out of the three developers involved in the precision study. The top-5 retrieved code snippets are presented in table 4.6.

The top-3 code snippets that were considered relevant all have one common characteristic - they all open a file in write mode. The task of opening a file and writing data can be considered to be similar to copy pasting data from the clipboard to a text file. A text file needs to be opened before the data from the clipboard can be written to it and a code example that opens a text file is relevant to the task the developer is trying to accomplish.

Now consider the query "remove problem marker from resource". Table 4.5 shows that the top-4 code snippets returned by the system were all irrelevant whereas the 5th code snippet was considered relevant. The top-5 retrieved code snippets are presented in table 4.7.

The top-4 code snippets with the highest cosine similarities to the query vector are all irrelevant. These results are interesting as the top-2 code snippets are both performing resource management. The top code snippet is removing the resource itself while the second code snippet is returning the location of the resource. However, the task the the developer

Sl. No.	Code Snippet	Cosine Sim%	Relevance
1	public static Document openDocument(OutputStream outputStream){ return openDocument(outputStream, PageSize.A4); }	82.89	Yes
2	public static void writeProtoTextToFile(File outputFile, Message proto) throws IOException { try (BufferedWriter outWriter = Files.newWriter(outputFile, StandardCharsets.UTF_8)) {TextFormat.print(proto, outWriter);} }	79.96	Yes
3	public static PrintWriter openFileForPrintWriter(final File file) throws IOException { if (file.getName().endsWith(".gz")) { return new PrintWriter(openFileForWriting(file));} else {return new PrintWriter(file);} }	79.41	Yes
4	protected TimelineReader loadAudioTimeline(String fileName) throws IOException, MaryConfigurationException { return new TimelineReader(fileName); }	78.85	No
5	private static String[] showChoices(Component parent, Document doc, String classification) { final boolean doTaxonomy = classification.equalsIgnoreCase (Classification.Taxonomy); ... choice1.getCurrentText(false), choice2.getCurrentText(false) };} return null; }	78.33	No

Table 4.6: Top-5 Retrieved Code Snippets with Corresponding Cosine Similarities for the "copy paste data from keyboard" User Query

Sl. No.	Code Snippet	Cosine Sim%	Relevance
1	void discardResource() { R old = resource; lifecycle.onRemoval(resourceKey .getKey(),old);}	81.59	No
2	public ResourceLazyLoadingScript script(String resourceLocation) { this.resourceLocation = resourceLocation; return this; }	71.59	No
3	public static Map¡String, Object¿ createUserPrefMap(GenericValue rec) throws GeneralException { return addPrefToMap(rec, new LinkedHashMap¡String, Object¿()); }	71.12	No
4	public boolean removeEdge(Edge e) { if (!edges.remove(e)) return false; srcMap.remove(e.getSrc(), e); tgtMap.remove(e.getTgt(), e); unitMap.remove(e.srcUnit(), e); return true;}	70.95	No
5	public static void removeMarkers(JTextComponent component, SimpleMarker marker) { Highlighter hilite = component.getHighlighter(); Highlighter.Highlight[] hilites = hilite.getHighlights(); for (int i = 0; i ¡ hilites.length; i++) { if (hilites[i].getPainter() instanceof SimpleMarker) { SimpleMarker hMarker = (SimpleMarker) hilites[i].getPainter(); if (marker == null —— hMarker.equals(marker)){ hilite.removeHighlight(hilites[i]); } } } }	70.86	Yes

Table 4.7: Top-5 Retrieved Code Snippets with Corresponding Cosine Similarities for the ”remove problem marker from resource” User Query

is trying to accomplish is removing a problematic marker from the resource and therefore, these two methods aren't useful. The third code snippet uses hash maps which is definitely irrelevant. A potential reason as to why it was retrieved by the system might be because the comment generated by the neural model is incorrect. Code snippet 4 assigns more importance to the keyword 'remove' but this example removes code snippets in a map and is unhelpful. The 5th code snippet is returned since it contains the keywords 'remove' and 'marker' and is relevant because it is similar to the objective of the developer. One key observation that could potentially have implications on future work is the reliance of information retrieval techniques on the presence of keywords in the search corpus. However, it is difficult to determine the relative importance of each word in the query without an attention based neural model.

This concludes the results chapter. The next chapter discusses answers to the research questions mentioned in section 1.3 along with the summary of the results and threats to validity.

Chapter 5

Discussion

Overall, the preliminary quantitative results are promising and validate the combination of neural models with information retrieval for the code search task, the system is still a work-in-progress with lots of room for improvement. This section includes a summary of the results, answers to research questions and threats to validity.

5.1 Summary

As stated in the previous section, the performance of information retrieval is heavily dependent on the quality of comments generated by the neural model. Upon analysing the results, it can be concluded that the task of automatic comment generation is best modelled as a neural machine translation problem where the code written in a programming language has to be translated to natural language rather than as an information retrieval problem. This is because information retrieval based approaches rely on an exhaustive set of keywords to generate comments whereas this approach can generate sequences unseen in the training data. Retaining the tree-like structure of the AST represents the code bet-

ter rather than treating source code as natural language. This allows the model to learn underlying patterns in code and enable it to predict similar comments for code snippets that have different implementations but perform similar tasks. Treating code search as a 2 stage problem also has some advantages. Firstly, the output predictions of the neural model can be manually examined before performing the retrieval and incorrect predictions can be modified for better retrieval results. Secondly, most existing neural code search models work by mapping the code snippets and corresponding comments in a shared vector space such that the code-snippet pairs are pushed close together. These approaches do not leverage the syntactic information present in source code, while the approach followed in this thesis is able to utilize this information in an effective manner.

However, semantic code search is a rapidly evolving area of research with researchers swiftly making improvements and advances to existing systems using pre-trained models, reinforcement learning and other hybrid techniques. Recently, Feng et al. [10] developed a single model architecture that is able to utilize both natural language as well as programming language data together. This is a huge break-through in the field of NLP, especially for the task of code search. This highlights the rate at which the interest in this field is growing and all the research that is being done to potentially make the lives of developers easier.

5.2 Research Questions

RQ1: *Can automatically generated comments for code snippets be used to retrieve relevant code snippets?*
Yes. This thesis demonstrates that comments generated by a machine can successfully be used to implement a hybrid code search system that utilizes deep learning as well as information retrieval to retrieve relevant code snippets from a search corpus.

RQ2: *Will the performance of the code search system be affected by the size of the retrieved set of code snippets?*

Yes. The search corpus in this thesis consisted of 8300 code snippets. The results show that the top result returned by the system was relevant to the user query 40% of the time. This can be attributed to the fact that the search space was small compared to other code search systems. However, the experimental results also show that users found at least one retrieved code snippet among the top-5 search results that was relevant to the search query. This shows that the hybrid code search system presented in this thesis performs well even with small code corpora.

RQ3: *Will the performance of this code search system that leverages the syntactic and structural information present in source code be be comparable to state-of-the-art methods that treat code snippets as natural language?*

Yes. The primary advantage of a model that effectively utilizes the syntactic and structural information present in source code is that it is able to learn patterns in code snippets that have similar functions that might be overlooked in models that treat code as natural language. This means that code snippets that perform the same function with different implementations will be retrieved by this system and the developer can choose which implementation is best suited for the task at hand.

5.3 Challenges and Threats to Validity

1. Lack of good datasets for auto-comment generation: The training data corpus used to train the code2seq model for the task of automatic comment generation consisted of approximately 150k examples which isn't sufficient. In order to get viable results, the model will have to be trained on millions of examples. Existing datasets are small and noisy and this problem will need to be addressed by either creating a new dataset altogether or combining existing

datasets if possible to generate a sufficient amount training data.

2. Code documentation and user queries are fundamentally disparate: The dataset used in this thesis collected the comments for Java methods from the first sentence of Javadocs leading to the presence of mismatched and incomplete comments in the dataset. Documentation also provides a general description of the code snippet and becomes outdated as time passes whereas a user query is usually structured as a question. It isn't ideal to use code documentation in order to learn the semantic meaning behind source code but currently, it is the only option since it bridges the gap between the source code (written in a programming language) and the query input (written in natural language).

3. Evaluation metrics for automatic comment generation: The gap between machine generated comments and human-written comments is evaluated using the Rouge or BLEU score. These metrics are used to reduce the the subjectivity of manual evaluation. However, these metrics measure the number of words that overlap between the comments that were written by humans and those generated by the machine. The automatically generated and manual comments may describe similar functionalities but with different words or order. This reduces the overall BLEU score portraying that the model has poor performance which may not be the case. In addition to this, the BLEU score tends to favour short predictions, which are usually not informative, especially for the task of automatic comment generation. It also does not take into account the underlying meaning or grammatical correctness of the translated text output.

4. Code snippet search corpus: The corpus of 8300 code snippets used to create the search space for information retrieval is extremely small in comparison to thousands of Java projects available on GitHub. Plans for future work include running experiments to evaluate the effectiveness and scalability of this code search system on code corpora of different sizes.

5. Quantitative evaluation of code search: The relevance of code snippets depends on the

developer, what task they're trying to achieve and their level of experience with the programming language. In addition to this, more than one code snippet might be relevant to the user query. Evaluating the search results quantatively using popular information retrieval techniques wasn't a feasible option as no dependable human annotated ground truth dataset that was similar to the training corpus was available. Therefore, a qualitative evaluation was performed through a precision study. However, the precision study had to be small and restricted to ten queries because the number of participants were limited in number.

6. Generalizability of Results: The code search system has only been applied to a Java corpus [14]. The effectiveness of this system for languages other than Java and how generalizable the experimental results and model performance reported in this thesis are when applied to different datasets is a fascinating question yet to be answered.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis formulates the task of code search as a hybrid two stage problem comprising of automatic comment generation and information retrieval. The task of automatic comment generation for code is formulated as a neural machine translation problem which translates source code written in a programming language to comments written in natural language. An attention-based code2Seq model that is able to learn semantic and syntactic information from code snippets is used to generate comments for Java methods. These code snippets along with the corresponding machine generated comments are then used to form a database which is searched in order to find and retrieve methods that are relevant to a user query based on the cosine similarity between the query and the machine generated comments for the methods in the search corpus. I have demonstrated that neural models that leverage the syntactic structure of source code can be combined with traditional information retrieval techniques in order to develop a robust semantic code search system that is able to fetch relevant code snippets even if the query doesn't contain the exact keywords present in the

search corpus. I hope that this thesis encourages more researchers to build code search systems that can learn and utilize the semantics from the syntactic nature of source code. I hope to build upon my solution in the future by improving and refining it through iterative development of the designed model.

6.2 Future Work

Plans for future work include improving the performance of the model by collecting copious amounts of training data making the model more generalizable by training it on code snippets from other languages; running experiments with different datasets and improving the information retrieval technique used in this thesis in order to make it more efficient and scalable to large datasets.

6.2.1 Dataset Experimentation

The results reported in this thesis are values obtained when the model is trained on 140k examples. NLP models, especially ones used for neural machine translation problems are data hungry and require millions of examples in order learn the semantics between the two languages. Running experiments with different datasets and more training data will improve the model's predictions on unseen code snippets as it will have more data to learn from. NLP is also a rapidly growing field and there is a possibility of replacing the Code2Seq model used in this thesis with an improved model that is able to generate better documentation for source code.

6.2.2 Programming Languages

Source code of large,complex software systems consist of thousands of lines written in different programming languages such as Java, Python, Scala, Javascript etc. In this thesis, the model was trained on Java methods and can therefore only generate comments for Java code. In order for it to be useful in the industry, this work needs to be made more generalizable by training the model on code snippets written in different languages so that the entire codebase can be maintained searched instead of just the parts written in Java. This goal of language extension can be achieved by modifying the model and script to construct Abstract Syntax Trees and extract paths from code snippets written in additional languages. PathMiner [19] is an an open-source library that can be used to mine path-based code representations for different programming languages in order to extend this thesis to other languages.

6.2.3 Improvements to Information Retrieval Technique

The current implementation performs information retrieval by storing the code snippets and the corresponding machine generated comment embeddings in a Pandas dataframe. [28] and computing the cosine similarity between the query embedding with all the embeddings in the search corpus and retrieving the code snippets with the highest cosine similarity to the query embedding. While this works efficiently for a small dataset, it isn't very efficient when searching through millions of code snippets. Thus, future work includes making the storage and search of the code snippets more efficient and scalable by using techniques such as inverted indices to reduce memory usage, storing it in a relational database instead of a dataframe, storing the location of the code snippets in a repository and retrieving it when required instead of storing the actual code snippet in memory, clustering similar user queries together and hashing the results so that the cosine similarity between a query and the search corpus need not be computed everytime.

6.2.4 Comprehensive Evaluation of Code Search

This thesis evaluated the semantic code search system through a precision study consisting of three participants and ten queries. A more thorough evaluation involving more developers and more queries is required in order to assess the impact code search has on developer productivity. As this is hard to do in an academic setting, another possible direction is to evaluate the system using a reliable human annotated ground-truth dataset which takes into account the subjectivity involved in determining the relevance of a code snippet.

Bibliography

[1] U. Alon, S. Brody, O. Levy, and E. Yahav. code2seq: Generating sequences from structured representations of code, 2019.

[2] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning distributed representations of code, 2018.

[3] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: A search engine for open source code supporting structure-based search. volume 2006, pages 681–682, 10 2006.

[4] S. Bajracharya, J. Ossher, and C. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *FSE '10*, 2010.

[5] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*, 2016.

[6] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra. When deep learning met code search. ESEC/FSE 2019, page 964–974, New York, NY, USA, 2019. Association for Computing Machinery.

[7] D. Cer, Y. Yang, S. yi Kong, N. Hua, N. Limtiaco, R. S. John, N. Constant, M. Guajardo-Cespedes, S. Yuan, C. Tar, Y.-H. Sung, B. Strope, and R. Kurzweil. Universal sentence encoder, 2018.

[8] S. Chatterjee, S. Juvekar, and K. Sen. Sniff: A search engine for java using free-form queries. FASE '09, page 385–400, Berlin, Heidelberg, 2009. Springer-Verlag.

[9] F. Farmahinifarahani, V. Saini, D. Yang, H. Sajnani, and C. Lopes. On precision of code clone detection tools. pages 84–94, 02 2019.

[10] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[11] X. Gu, H. Zhang, and S. Kim. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 933–944, New York, NY, USA, 2018. Association for Computing Machinery.

[12] G. Heyman and T. V. Cutsem. Neural code search revisited: Enhancing code snippet retrieval through natural language intent, 2020.

[13] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, page 117–125, New York, NY, USA, 2005. Association for Computing Machinery.

[14] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, page 200–210, New York, NY, USA, 2018. Association for Computing Machinery.

[15] H. Husain and H.-H. Wu. How to create natural language semantic search for arbitrary objects with deep learning, 2018.

[16] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search, 2020.

[17] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Summarizing source code using a neural attention model. In *ACL*, 2016.

[18] H. Jiang, L. Nie, Z. Sun, Z. Ren, W. Kong, T. Zhang, and X. Luo. Rosf: Leveraging information retrieval and supervised learning for recommending code snippets. *IEEE Transactions on Services Computing*, 12:34–46, 2019.

[19] V. Kovalenko, E. Bogomolov, T. Bryksin, and A. 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, 2019.

[20] S. Mahapatra. Use-cases of google's universal sentence encoder in production. `https://towardsdatascience.com/use-cases-of-googles-universal-sentence-encoder-in-production-dd5aaab4fc15`, 2019.

[21] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space, 2013.

[22] J. Pennington, R. Socher, and C. Manning. Glove: Global vectors for word representation. volume 14, pages 1532–1543, 01 2014.

[23] D. Poshyvanyk, A. Marcus, and Y. Dong. Jiriss - an eclipse plug-in for source code exploration. volume 2006, pages 252– 255, 07 2006.

[24] M. Post. A call for clarity in reporting BLEU scores. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, pages 186–191, Belgium, Brussels, Oct. 2018. Association for Computational Linguistics.

[25] Rachel Tatman, Towards Data Science. Evaluating text output in nlp at your own risk. `https://towardsdatascience.com/evaluating-text-output-in-nlp-bleu-at-your-own-risk-e8609665a213`.

[26] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra. Retrieval on source code: A neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, page 31–41, New York, NY, USA, 2018. Association for Computing Machinery.

[27] R. Sindhgatta. Using an information retrieval system to retrieve source code samples. ICSE '06, page 905–908, New York, NY, USA, 2006. Association for Computing Machinery.

[28] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.

[29] Wikipedia. Bleu. `https://en.wikipedia.org/wiki/BLEU`.

[30] Wikipedia. Rouge (metric). `https://en.wikipedia.org/w/index.php?title=ROUGE_(metric)&oldid=913825951`, 2019.

[31] Wikipedia. Abstract syntax tree. `https://en.wikipedia.org/w/index.php?title=Abstract_syntax_tree&oldid=1016693387`, 2021.

[32] Z. Yao, J. R. Peddamail, and H. Sun. Coacor: Code annotation for code retrieval with reinforcement learning. In *The World Wide Web Conference*, WWW '19, page 2203–2214, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Yasmin Moslem. Computing bleu scores for machine translation. `https://blog.machinetranslation.io/compute-bleu-score/`.

[34] M. Zhang. Searching for code? let a neural network do that for you!, 2019.