UNIVERSITY OF CALIFORNIA,
IRVINE

Code Clone Detection using Code2Vec

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Software Engineering

by

Anupriya Prasad

Thesis Committee:
Professor Cristina V. Lopes, Chair
Associate Professor James A. Jones
Professor David Redmiles

2020

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

Page

ACKNOWLEDGEMENTS

I would like to express the deepest appreciation to my committee chair and thesis advisor, Professor Cristina V. Lopes, for her invaluable guidance, immense knowledge, and persistent support.

I would also like to express my sincere appreciation to Farima for her support and insights along the way.

I would like to thank my committee members, Associate Professor James A. Jones and Professor David Redmiles, for their valuable feedback, insights, and time.

ABSTRACT OF THE THESIS

Code Clone Detection using Code2Vec

by

Anupriya Prasad

Master of Science in Software Engineering

University of California, Irvine, 2020

Professor Cristina V. Lopes, Irvine, Chair

Code Clone detection is important in software engineering as it aims at solving various problems like code maintenance, identification, code reuse, scalability, and plagiarism. Software development revolves around implementing logic using tools and technologies where every developer has a different coding style and logical approach to reach required goals. However, the end result of many implementations can be the same. This is where the need for code maintainability, reusability, and optimization arises. Code clone detection can help to leverage the immensely large source codes available on the web to attenuate code writing time by reusing sources available online. Clone detection in source code is based on the similarity of the program content or similarity in the program functionality. There are many techniques that have been tried and tested in the literature. However, these naïve approaches do not perform adequately for higher level clones. In this thesis I am exploring deep learning based technique Code2Vec.

In order to identify, compare, and reuse an existing piece of code, deep learning techniques can help to predict if a similar implementation source code exists in a code base or a dataset of codes. In this thesis, the approach of representing code in the form of vectors and applying Natural Language Processing for code clone detection has been discussed. The scope of the thesis is to devise an approach for the detection of similar functional methods in Java GitHub code repositories, expanding on the Code2Vec model [1]. I demonstrate the capabilities of applying Code2Vec model to Java source code in order to determine the path vectors for method similarity detection. Furthermore, I discuss the design, architecture, and usage of the model. An in-depth analysis of preprocessing mechanisms, data collection, data preprocessing is also highlighted. In this thesis, I apply a normalization technique to update the variable names in Code2Vec approach and compare the baseline Code2Vec model with the normalized model. The comparison shows improved precision results in normalized code clone detection approach. Finally, the benefits of my approach and a detailed analysis of results on dataset with Java methods is presented. The results are evaluated on the basis of Recall and Precision. I evaluate the recall with the help of BigCloneBench and precision using InspectorClone and MeasurePrecision open-source tools.

Chapter 1

Introduction

1.1 Motivation

In larger code bases with multiple developers implementing different tasks, code duplication can be difficult to detect. This affects code maintainability as the complexity of code base increases with time. With the advent of IDEs, the call for code clone detection in working repositories is the need of the hour. Code recommendation capability will help developers write efficient code, promote code abstraction on either class or method level, and improve the code development process in programming in the large environments [23]. Traditionally, general purpose similarity detection in source codes was an adopted approach with the benefit of ensuring code semantic standardization for easier code reviews. Similarly, applications with support to identify the requirements of a developer based on the logical flow of code to detect the same logical functionality in the codebase can be helpful. This will promote the reuse of local code and open the possibility to search for code on open-source platforms. There are various other applications as well where code clone detection offers to

better the software development and software engineering practices and offers as the motivation in this Thesis.

1.2 Code Clone Types

Code clones are segregated into four types [17] ranging from Type-1 to Type-4 types of clones based on complexity and level of similarity. Type-1 clones can be easily detected because they are based on textual similarity, where identical pieces of code with very little difference especially in white-spaces, comments, and layouts. These can be refined with some effort in preprocessing/cleaning and preparing of the dataset. Typically, Type-2 clones are code pieces that have a token-based or lexical similarity. These are a little difficult to detect in comparison to Type-1 as the complexity increases. Type-3 clones are clones that are syntactically similar i.e., source code that is different at the statement level. The sections of code have statements modified, appended, or removed with respect to their clones, in addition to Type-1 and Type-2 differences. Finally, Type-4 code clones are clones based on syntactic similarity, where dissimilar fragments of code with the same logical functionality are considered clones.

1.3 Approaches for clone detection

I have described the different types of code clones that exist, there are several approaches to deal with different clone categories. One approach to detect clones in a dataset

is based on surface text duplication detection. In this approach, the code is considered as a stream of tokens and it involves matching code text/words. This is a naïve approach as it deals with direct matching and can miss out on clones easily, hampering the precision of code clone detection. To improve the clone detection in code, approaches exist which are based on control flow consideration. This requires expertise in the programming language for which the tool needs to detect clones. Another approach to solve this problem is the approach of Abstract Syntax Trees. ASTs are a common way of representing a program. AST paths from node to node in a code, capture some of the semantics using code syntax. This approach focuses on Type-1 to Type-4 clones.

In this thesis, the approach taken involves the application of deep learning model-Code2Vec [1] to Java GitHub code repositories in order to predict code clones. Deep Learning has been applied to various text/code based natural language problems in the past and has produced great results. Researchers have found it effective to learn the semantic meaning of the text using deep learning. Similarly, code fragments can be considered to have syntactic properties, which can be treated well with previous approaches. This approach is motivated by Mikolov et. al. work on Word2Vec [28], a natural language model that treats natural language as a continuous bag of words or tokens. The approach in the thesis is based on code path embeddings, which are abstract syntax trees-based paths of methods in the source code. This path-based representation is useful in semantic as well as flow based on ASTs. This is explained further in the background literature review section of the thesis.

1.4 Research questions

RQ1: Is Code2Vec capable of finding the code clones in a repository?

Code2Vec [1] is based on path embeddings in a code snippet. It captures the syntactical essence of a java method as a vector. There has been existing research where clones have been computed based on their Abstract Syntax Trees, and subtrees. Ira D Baxter et al. [16] have presented their approach for Code clone detection using AST and the challenges along with applications. Yes, Code2Vec can capture the syntactic nature of the code and can be used to find clones.

Q2: How does performance of code2vec change based on normalization?

The baseline Code2Vec model is based on top rated (five star rated) projects on GitHub. I have performed experiments to observe that when the variable names are changed in the code to randomly chosen names, the predictions for method names become poor. This observation is presented in Chapter 4-Normalization, where I discuss the motivation to add normalization to baseline Code2Vec. Normalization means renaming variables in the codes to standard variable names in order to reduce the impact of variable names in code clone detection. In Chapter 5-Observations and Results, it is shown that the performance computed in terms of precision is better for the normalized Code2Vec in comparison to baseline Code2Vec. Therefore, yes, the performance of Code2Vec changed after normalization.

1.5 Contribution and Chapters Overview

The first contribution of this thesis is developing upon the Code2Vec model [1] in order to find code clones in an existing dataset. The second contribution is to evaluate the effectiveness of removing the bias of variable names in Abstract Syntax Tree creation, used as a preprocessing step in the neural network. This is presented as comparisons between baseline Code2Vec and normalized Code2Vec.

The thesis presents the approach for computing the code clones for several Java methods using the Code2Vec model. The thesis is divided into the following chapters. In chapter 2, the information regarding relevant literature with proper background survey on Code2Vec, deep learning for source code, precision and recall measurements, code clone detection research is discussed in depth based on previous papers and research. Additionally, this section provides information on resources referenced in order to make progress in this thesis with a focus on research questions that have motivated me in my thesis. In chapter 3, the Code2Vec model is described in great detail. I have discussed about the initial study. Moreover, I have presented an established understanding of datasets used, data preprocessing techniques and approaches in chapter 4. The experimental design, hardware and hyperparameter specifications are also mentioned in chapter 4. In chapter 5, results and implications are discussed. Finally, results are evaluated, and the thesis is concluded in chapter 6 with analysis and future work.

Chapter 2

Background and Related Work

This thesis is based upon research work on Code2Vec [1], SourcererCC [14], InspectorClone [17], which have been discussed in detail. I have evaluated the Code2Vec model on Java source code downloaded from code2vec website. Code Clone detection has been a vital problem and engineers have investigated and performed case studies on how developers search code fragments for years. Google Researchers have a study on the topic that indicates that developers look for logical code and do not use already existing similar code in working repositories more often during software development. Various literature was explored during this thesis on clone detection, NLP, recall, and precision.

2.1 Code Clone detection literature

Cristina V Lopes et al. have developed SourcererCC [17], the most scalable tool so far for detecting Type-3 clones. Type-3 clones are the ones that differ at the statement level. The tool focuses on the evaluation of precision, recall, scalability, and execution time for inputs

of various domains and sizes, including the large inter-project software repository IJaDataset 2.0 [4] which includes 25K projects with 250 million lines of code and 3 million examples. The tool showcased results with good execution time without any scalability issues, even on a normal hardware i.e., machine with 3.5GHz quad-core i7 CPU with 12GB RAM. SourcererCC [17] compared code fragments through the bag-of-tokens technique. Since it is a token-based accurate near-miss clone detection tool, that uses an optimized partial index and altering parameters to achieve largescale clone detection on a machine with simple average hardware. The authors measure the recall of this tool using two state-of-the-art clone detection benchmarks, the Mutation Framework and BigCloneBench [5]. SourcererCC [17] produces great results and is highly competitive with even the best of the state-of-the-art Type-3 clone detectors. A lot of motivation has been taken from this tool in this thesis for results evaluation.

In the research paper on Oreo [6] by Cristina V. Lopes et al., a source code clone detection technique and tool have been proposed that identifies clones in the twilight zone. Oreo is a combination of information retrieval, machine learning, and metric-based approaches. The technique includes an Action filter and a two-level input partitioning strategy, which reduces the number of candidates. Despite this, it maintains good value for recall in the results. A neural network with Siamese architecture was introduced in this research. Oreo is a scalable, accurate, and performs significantly better than other four state-of-the-art tools that Oreo was compared with. It performs great clone detection and outshines in harder-to-detect clones in the twilight zone.

CCLearner [11] is another approach that leverages deep learning to identify clone and non-clone methods in source code fragments. The authors have leveraged the usage of terms in source code such as identifiers and reserved words to create a feature vector for each source code method. The results are verified using BigCloneBench [5] for training a classifier and the rest for testing the model. The tool is scalable to a dataset of 3.6 MLOC.

InspectorClone [17] is used in this thesis for clone results verification. This tool uses an approach designed to facilitate precision studies for code clone detectors. InspectorClone helps in evaluating precision of clone detectors and automatically resolves clone pairs as true positive or false positive, removing the need of manual inspection.

2.2 Deep learning for sourcecode literature

Alon et al.[1] have proposed Code2Vec, an approach based on learning the distributed representation of source code [2] similar to Word2Vec [28]. In their work, they have created a natural language processing based neural network model which represents fragments of code in the form of continuous distributed vectors known as source code embeddings. These embeddings are used to predict the semantic representation of a code fragment. The authors have performed this representation by decomposing the source code into a bag of paths in the code fragment's abstract syntax tree. The neural network is trained by learning the atomic representation of each path embedding. They make use of the embedding similarity of similar code fragments to predict method names based on trained examples. Furthermore,

this approach has been validated by training a model on a dataset of 14M Java methods. It was observed from the results that the neural net model can predict correct method names for examples that were unobserved. This model has a great prediction rate and is 100 times faster than other competitive approaches. While the authors used the Code2Vec model for method name prediction, they mentioned that this model can be extended and used for various other programming language processing tasks such as semantic clone detection, semantic search, etc. Before Code2Vec, the authors had devised an approach to predict code properties using a path-based code representation, which further leads to Code2Vec. PathMiner was created by Kovalenko et al. [3] to extract these path-based representations for several programming languages.

White Martin White et al. [7] presented an unsupervised deep learning approach to detect clones at the method and file levels. They explored the technique by automatically learning different features of source code fragments. Their approach had 93% precision. Similarly, in Wei and Li's approach [8], a Long-Short-Term-Memory network is applied to learn representations of code fragments and code parameters. A hash function is used that computes the hamming distance between hash code of code clone pairs. It is ensured that this hamming distance is kept as minimum as possible. They have also verified their results on BigCloneBench [5] and OJClone.

Sheaneamer and Kalita [9] use Abstract Syntax Trees and Program Dependence Graph based techniques for semantic and syntactic features to train their neural network model. They use semantic features and simple ensemble methods to train their model.

CODEnn is a similar study presented by Gu et al. based on a deep neural network for searching code in which they demonstrate an application of a neural net model. The results are evaluated against traditional information retrieval systems like Apache Lucene. The results are a great improvement over IR based tools.

Hu et al. [12] performed a study to create comments and code documentation for source code and named the tool DeepCom. DeepCom works with Java methods and applies natural language processing techniques to learn from huge code corpus. The authors disintegrate the code in a neural net to analyze structural information of Java methods for generating comments. They used 9714 GitHub open-source projects and verified the results on machine translation metric.

There has been research on the usage of Code2Vec [1] for different applications. One such application is built by Briem et al. [13]. They have used Code2Vec for bug detection in code and detect one-by-one errors in Java code snippets. It is the form of a binary classification problem. In order to create bugs in the correct code, they have manually prepared the dataset by introducing problems in the correct code and creating false positive examples. These examples are used while training and validating their results.

Chapter 3

Code2Vec

In this section, I discuss the design of the code clone system. In order to understand the system, I explain the foundational techniques used like Abstract Syntax Trees, Code2Vec [1] model, dataset specifications, hyperparameters. An overview of the process is described in Figure.

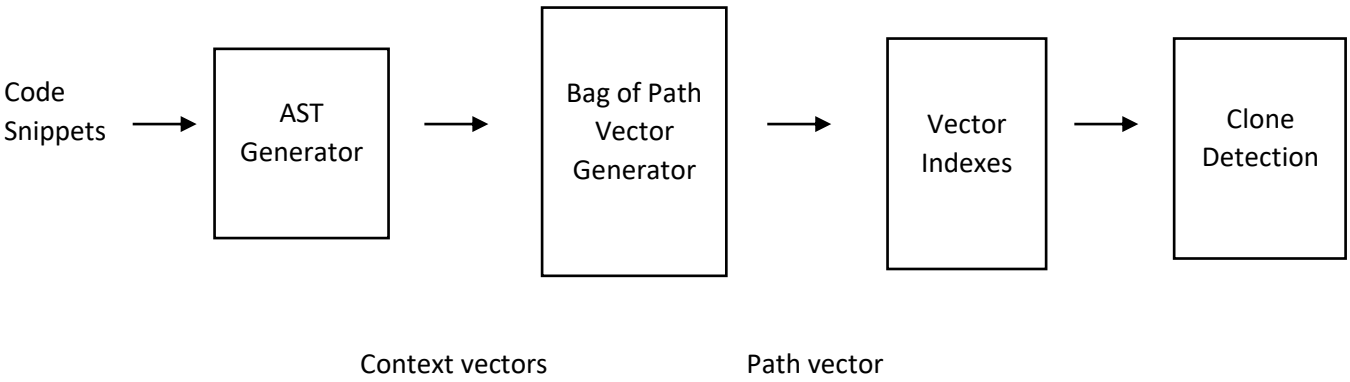


Figure 3.1: Pipeline process for clone detection

3.1 Abstract Syntax Trees

Abstract Syntax Tree used in this thesis is a basic tree representation of the abstract syntactic structure of the Java program. The formal definition is referenced from the Code2Vec model [1].

Abstract Syntax Tree for a source code snippet can be represented in the form of a tuple expression i.e., <N,T,X,s,δ,ϕ>, where N is a set of non-terminal nodes, T is a set of terminal nodes, X is a set of values, $s \in N$ is the root node, $\delta : N \rightarrow (N \cup T)$ is a function that links a non-terminal node to all of its children, and $\phi : T \rightarrow X$ is a function that links a terminal node to a particular associated value.

Abstract Syntax Tree Path is a representation of a path in a tree i.e., way to join two nodes in a tree. It is used to represent the path embedding in Code2Vec model [1] and is defined as a path between two nodes starting from one terminal node ending in another terminal node, with an intermediate node called non-terminal node in the path which is the common ancestor of both terminal nodes.

AST path is represented as $n_1 d_1 \ldots\ldots n_k d_k\, n_{k+1} d_{k+1}$, where

n_1 and $n_{k+1} \in T$ i.e. the terminal nodes, for $i \in [2..k]$,

$n_i \in N$ i.e. the nonterminal nodes for $i \in [1..k]$,

$d_i \in \{\{\uparrow, \downarrow\}$ are movement directions i.e., up and down depending on the traversal in the tree to create the path. If $d_i = \uparrow$, then: $n_i \in \delta(n_{i+1})$; if $d_i = \downarrow$, then: $n_{i+1} \in \delta(n_i)$. For an AST-path p, we

use start (p) to denote n_1 - the starting terminal of p; and end (p) to denote n_{k+1} - its final terminal. Example paths in Figure 3.3:

1. (MultByX, Method Declaration ↓ BlockStmt ↓ ReturnStmt ↓Binary Expr:times , x) denoted by the color green in Figure 3.3.
2. (x, parameter ↑ Method Declaration ↓ BlockStmt ↓ ReturnStmt ↓ Binary Expr:times , x) denoted by the color red in Figure 3.3.

A path-context is defined as a tuple of an AST path and the values associated with its terminals. The Code2Vec [1] model uses this as a definition and mentions that given an AST Path p, its path-context is a triplet ⟨xs, p, xt⟩ where xs =ϕ (start (p)) and xt =ϕ (end (p)) are the values associated with the start and end terminals of p. That is, a path-context describes two actual tokens with the syntactic path between them. This path is used to form the path embedding that is used in training the neural network model. The embedding is chosen to limit the size of training data and reduce the sparsity of the tree that is generated. If the AST is deeper, it will impact the results as the tree will be less sparse. Different applications can deal with different ways to represent a tree into path embedding for the neural net. There is some work where the paths are limited to a maximum length depending on child nodes and their index and if they are used in any other path. These are experimental approaches based on applications.

3.2 Code2Vec Model

3.2.1 Model Overview

This thesis is based on a bag-of-paths neural model Code2Vec [1]. The Code2Vec model uses pure TensorFlow and TensFlow's Keras and is trained on 14 M source code examples. It is a model that represents code as continuous distributed vectors known as code embeddings. The model used in this thesis can be trained further on Java code snippets. In this approach, a code snippet is composed of a bag of contexts, and each context is represented by a vector. The values of this vector capture two pieces of information. First, the semantic meaning of this context. Second, the amount of attention this context should receive. These vectors are further used in the learning part of the process as the model decomposes the code snippets into a collection of path embeddings using Abstract Syntax Trees and learns the atomic representations of aggregated paths. Therefore, code is represented as a Bag of Path-Contexts.

3.2.2 Feature Extraction

The script for feature extraction is developed by Code2Vec authors Alon et al. [1], based on an open-source AST miner called PathMiner from JetBrains research. For example, a line of code "x=7" can be represented as a path through AST as <x, NameExpr ↑ AssignExpr ↓ IntegerLiteralExpr), 7⟩. I will explain how the path is extracted from source code through ASTs using the below example in Figure 3.2.

For a small program to compute the x times of any input number, the code is as follows:

```
int MultByX(int x) {
        return x*x;
}
```

Figure 3.2: Code snippet for Multiply by x program

The above code computes x times any number that is provided to the function. This code snippet is parsed to create an Abstract Syntax Tree which is shown as below in Figure 3.3. AST is traverse to find the path between terminal nodes i.e., leaves in the tree and this is shown in the form of a string representation with links shown as up and down arrows.

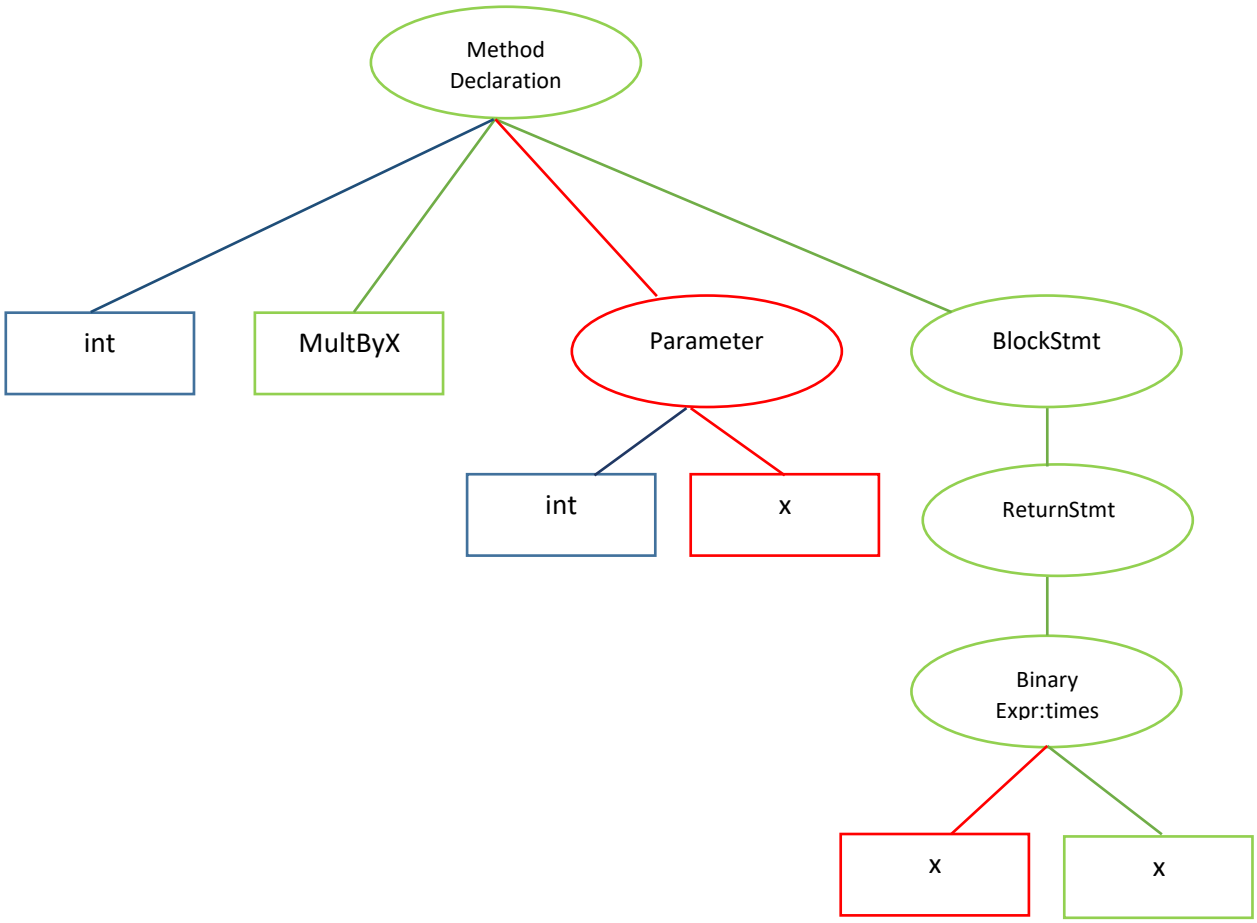


Figure 3.3: Abstract Syntax Tree representation of code snippet in Figure 3.2

Example paths from AST in Figure 3.3 can be:

1. (MultByX, Method Declaration ↓ BlockStmt ↓ ReturnStmt ↓Binary Expr:times , x) denoted by the color green in Figure 3.3.
2. (x, parameter ↑ Method Declaration ↓ BlockStmt ↓ ReturnStmt ↓ Binary Expr:times , x) denoted by the color red in Figure 3.3.

3.3 Attention Model

3.3.1 Distributed Representation of Context

Once the code snippets are formulated into Abstract Syntax Tree form and the paths are developed, a range of distributed representation of context is formed. Each of the path and leaf values of path-context is mapped to its corresponding real-valued vector representation called path embedding.

3.3.2 Path-Attention network

The Path-Attention network aggregates multiple path-context embeddings into a single vector that represents the entire essence of the method body. Attention is the mechanism that is used to learn to score each AST path-context, so that the most important path gets the maximum weightage while assigning importance. Multiple context vectors are aggregated using the attention scores that are allotted to every vector during learning into a

single code. Attention helps with two goals. First, it explains which path should get how much attention. Secondly, it captures the semantic meaning of path-context. Most importantly, the dot product of learned attention vector with path contexts after normalization gives learned path-vector weighted average, which is used to predict results. In this thesis, the AST paths for Java methods are given attention weightage which is used to capture the essence of the methods, further used to predict clones.

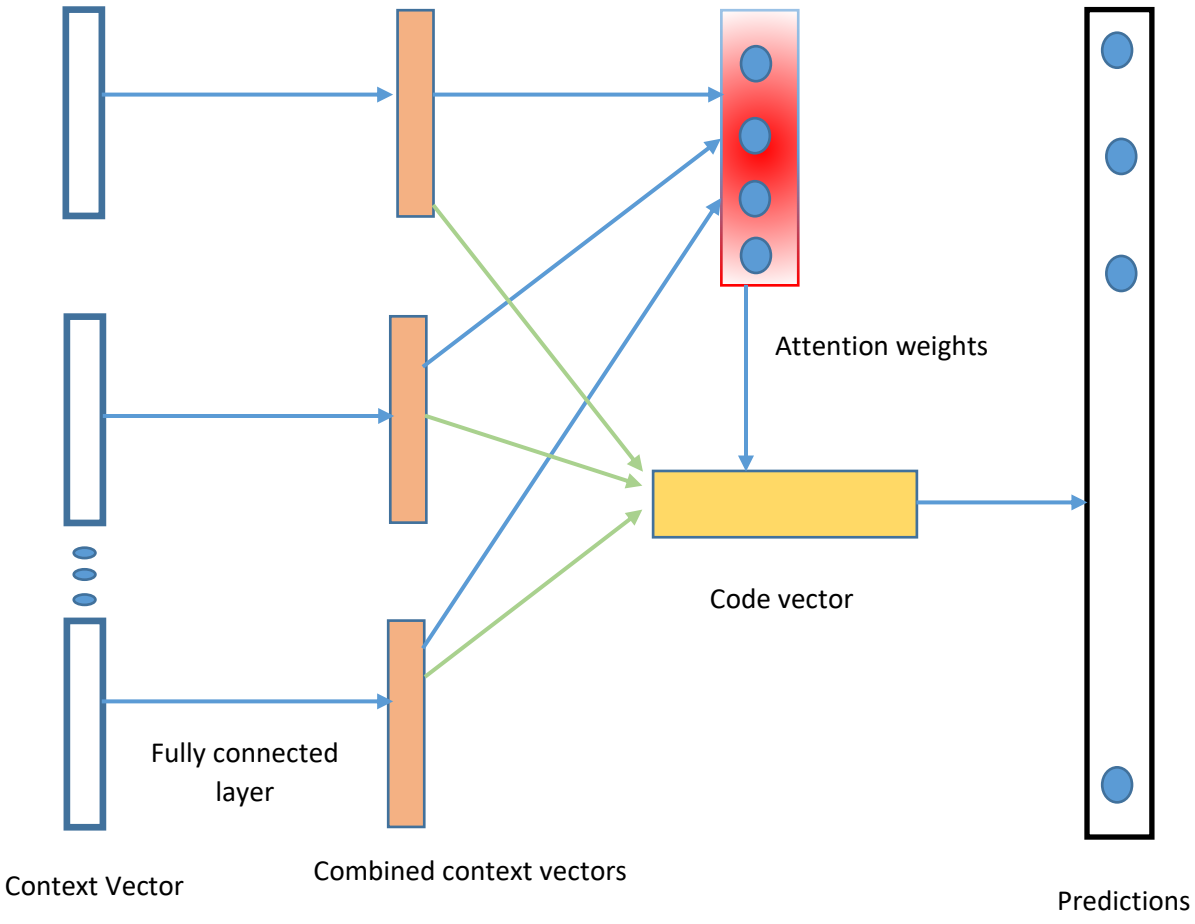


Figure 3.4: Path Attention network that gives weightage to some AST paths

Chapter 4

Experimental Design

4.1 Dataset

In this thesis, I have used two datasets for code clone detection task and experimentations. These datasets are as follows:

1. BigCloneBench IJaDataset
2. Code2Vec dataset of 10,072 Java Github projects

The model and approach were used and verified on the dataset provided by Code2Vec authors, a dataset of 10,072 Java GitHub repositories, originally introduced by Alon et al [1]. The results of this dataset are presented in Chapter 5 in detail.

Secondly, I used the standard BigCloneBench IJaDataset [5] dataset, which is a benchmark of known clones in the IJaDataset. The BigCloneBench IJaDataset is a large repository of over 25000 open-source Java projects, with over 3 million source files. It is divided into 3 sections i.e., default, sample, and selected. In BigCloneBench dataset, a code

fragment is a single method and there are over 8 million validated code clones in the dataset. In this thesis, the experiments use only a subset of code snippet pairs in the BigCloneBench IJaDataset for clone detection and validation. This is explained in greater detail in Chapter 5 – observations and results. The BigCloneBench benchmark helps to claim and compute only the recall value of a clone detection tool and not the precision figure for the tool.

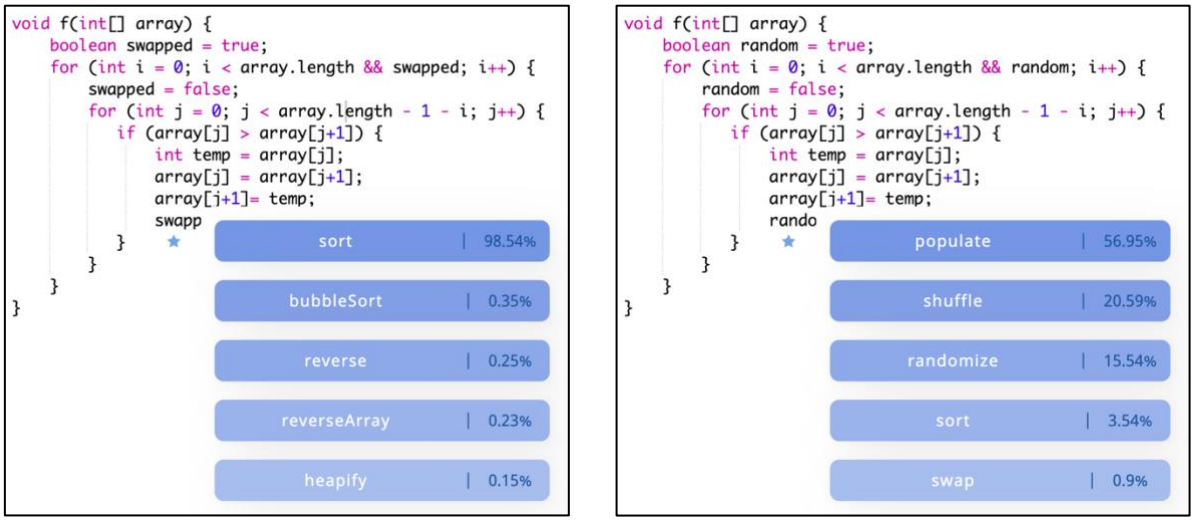
4.2 Experimental Design

One key strategy to scaling super liner analysis of large datasets is to preprocess the data for a machine learning model as much as possible. Preprocessing is a step that is vital for proper results. It is one linear scan of the input dataset with filtering and mutation operations performed with the goal of extracting features from it. These extracted features further help in better organization, optimization, and actual data processing.

4.2.1 Normalization

One important experimental design decision taken in this thesis is to update the AST path generator to ensure that the variable names are replaced with standard variable names to compute the paths. This decision is taken with an assumption that the code snippet representation as a path embedding in Code2Vec will provide more weightage to the factors other than variables in the code. We know that Type2 clones are easier to detect than Type3 and Type4 clones. The assumption is that in order to detect Type3 and Type4 clones, the focus has to be put on the logical flow of the code snippet which is captured in AST than the parameter value. Also, Code2Vec uses variable names as it is for node considerations with

an assumption that the top five stars rated GitHub repositories will have good variable naming conventions in the code. To understand the relation between variable name in Code2Vec results, I have presented the results of sort function in Figure 4.1 from code2vec website where it is shown that when the variable name is changed for the same method, the predictions are impacted. The sort function in Figure 4.1 (a) used the bool name as 'swapped' and shows predictions as sort (98.54% similarity), bubbleSort (0.35%), reverse (0.25%) and so on. When I changed the bool name 'swapped' to 'random' as shown in Figure 4.1 (b), the predictions changed to populate (56.95%), shuffle (20.59%), randomize (15.54%), and sort (3.54%). Both the methods still implement the sort functionality, but the similarity percentage in Figure 4.1 (a) shows 98.54% similarity with sort and Figure 4.1 (b) shows 3.54% similarity with sort. Therefore, it is noted that parameter names can result in poor predictions despite the method functionality being very similar.



a) Sort method with bool name 'swapped' b) sort method with updated bool name 'random'

Figure 4.1: Predictions of similar methods to given sort method with a) original bool name 'swapped' b) changed 'swapped' bool name to 'random'

Furthermore, I changed the variable ‘swapped’ to constant variable name as ‘var1’ in Figure 4.2, to understand how constant names impact the prediction. The similarity of sort to constant variable name is 72.92% and is a better prediction than sort code snippet with poor naming convention.

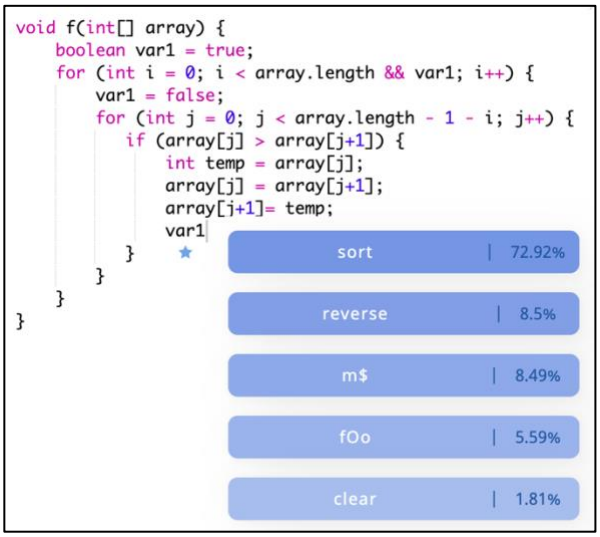


Figure 4.2: Predictions for sort method with bool variable name 'var1'

4.2.3 Normalization implementation

In this thesis, the approach is taken for renaming the variables in the path representation by standard variable naming convention. The variable encountered first is renamed as var1 and the next variable encountered is renamed as var2, so on so forth. The AST path in this case will have a starting node Node1 and a terminal node Node2 with a path between 2 nodes. This approach is referenced from opensource AST miner called PathMiner [3] from JetBrains research. This is done using a script with pseudocode described in Figure 4.3. Each code repository is filtered, and methods are extracted. Furthermore, the variables

from Java methods are renamed as “vari”, where ‘i’ ∈ [1..n], n is the number of variables in the method. This approach ensures that there is no bias on the basis of poor or good naming conventions adopted by developers of GitHub repositories that are used as a dataset in the model.

```
AST Update Algorithm Pseudocode:
for code in corpus:
        extract methods and store all the methods
        for method in methods:
                parse method and store integer ‘i’
                if current component is a variable:
                        rename the variable as “var”+i
                        increment i
```

Figure 4.3: Pseudocode for Abstract Syntax tree update from Code2Vec model

An experiment is performed that compares the baseline Code2Vec results for clone detection and the results with updated variable names AST to compare the accuracy. This experiment is done on Java methods selected from code2vec website. The true positives and false positive are computed for baseline Code2Vec without variable name change and with variable name change. These results are presented in Chapter 5-Observations and Results in detail.

4.3 Hyperparameters

In this thesis, the computations and training are done on a normal desktop machine engine with i7, 8GB memory. The hyperparameters to train the models are mentioned in this section. The max number of epochs to train the model is 20. The training batch size is 1024. The top K words considered during prediction is updated continually to check the results for precision and recall, with the initial value of 10. The path embedding size is considered similar to the Code2Vec model [1] that is the embedding size of the token. The dropout rate used during training is 0.75 and the learning rate as 0.01.

Chapter 5

Observations and Results

This chapter will showcase results obtained from the application of Code2Vec [1] to Java repositories of GitHub projects and BigCloneBench IJaDataset [5] dataset.

5.1 Precision

To measure the precision, I developed a script that runs on ten methods and predicts the top10 clones. The ten methods are selected from Code2Vec [1] interactive demo site http://code2vec.org. To predict the clones, I use the Gensim Python Package. Gensim package is a python library for topic modeling, document indexing, and similarity retrieval with large corpora. It is widely used in Natural Language Processing and Information Retrieval related work. The 'most_similar' function of the Gensim package inspects and retrieves the clones for Java methods with a value of prediction rate or similarity percentage. These clones are listed in the Table 5.2.

5.2 Recall

To measure the recall, I developed scripts that transformed the BigCloneBench IJaDataset [5] data in a directory structure that was accepted by Code2Vec to train and predict clone. BigCloneBench IJaDataset [5] dataset has tagged pairs for clones. I wrote logic in scripts that parses all the code and finds the method name from the first tagged clone in the tagged pair. Then Code2Vec model is run on the methods extracted from tagged clone pairs, Furthermore, the top1 prediction of the model is stored in a map format of the form Clone1 -> Clone Predicted. The script then finds the predicted clone in the BigCloneBench dataset and matches if the starting line and ending line of the predicted clone is the same as that of the second code in the tagged clone pair. This computes the recall value of the tool and the clone predictions.

5.3 Experimental Results

5.3.1 Baseline Code2Vec Clones

Table 5.1 shows the results for code clone detection using baseline Code2Vec, without any kind of changes. These results are presented in order to be compared further with results of application of design decision in section 5.3.2 and 5.3.3, where the variable names are updated as constant variable names. The table shows the similarity of top 10 predicted clones for 10 Java methods that are selected from code2vec website. The results are marked as True Positive or False Positive based on InspectorClone or MeasurePrecision jar file. Finally, these results are used in section 5.3.3 where the precision is computed.

Table 5-1: Results of Baseline Code2Vec design 10 Java methods (selected from https://code2vec.org) and their Top10 predicted clones with InspectorClone and MeasurePrecision values of TP (true positive) and FP (false positives)

				InspectorClone	
	Function	**Top10 Clones**	**Similarity %**	**TP**	**FP**
1	sort	sortdefinition	61.11	8	2
		currentsorts	57.27		
		nativenumbersort	55.76		
		sortx	54.44		
		numperslot	54.51		
		sorted	53.09		
		sorttrianglesinternal	50.75		
		dissolveproperties	49.99		
		mpackagenametoactivity map	49.42		
2	contains	containskey	65.5	2	8
		screentoclient	52.4		
		isinaabb	48.07		
		leafarray	48.04		
		wrongvaluepi	44.24		
		xlogicoperator	44.01		
		flatteningpathiterator	43.96		
		containscachedcontact	42.15		
		ibmthai	40.21		
		normalizedspan	40.17		
3	getindex	indexof	59.31	7	3
		findex	51.63		
		indexeclass	51.27		
		getindexforname	49.4		
		index	48.23		
		searchindexwithindexing progress	47.53		
		prefetchcrls	47.51		
		gettypeparameterindex	47.20		
		accessibletablecelllistener	42.11		
		carty	42.09		
4	add	myadd	64.73	8	2
		addelement	62.99		
		addcontentchangelistener	56.83		
		adding	55.17		
		getoffsettoadd	52.51		

		personw	51.63		
		put	49.73		
		addchild	49.34		
		addbefore	49.30		
		supercs	47.32		
5	reverse	reversed	53.01	7	3
		endlow	51.33		
		fronthigh	50.21		
		routeleg	49.34		
		reversedreaction	49.33		
		generatereverse	49.32		
		generalizedlemmatransformation	49.30		
		revline	49.21		
		dbdatanode	49.15		
		currentplaytime	49.02		
6	factorial	factorial	63.69	4	6
		factarray	62.19		
		longmath	61.33		
		factorials	61.31		
		productbits	55.57		
		xmlview	47.77		
		topchosen	46.32		
		isclientonlyservice	46.17		
		osfamilyversionbit	41.07		
		islookright	40.04		
7	download	isdownloadauthorized	62.02	8	2
		subscriptiondownloadlistener	67.29		
		githubdownload	60.10		
		handlecommandexitstatus	56.008		
		isdownload	52.58		
		downloadlocationfile	48.49		
		downloadfile	47.78		
		readthefile	47.34		
		authelem	47.04		
		ftpproxyset	43.03		
8	postrequest	post	75.59	3	7
		senddate	65.65		
		gettotalrevenue	41.97		
		updatecheckincomment	41.67		
		directaction	41.53		
		spotifyapiexception	41.22		

		createwriternotcalledtime	40.71		
		mcthreadlock	40.19		
		maxaltitudefeet	39.67		
		disabledbankaccountexception	38.11		
9	equals	equalsignorecase	79.11	5	5
		equal	73.23		
		setdailysubtotal	59.72		
		dateresattr	56.43		
		rioret	51.74		
		equalsm	49.74		
		aclmemberentity	45.11		
		fieldlayouts	41.01		
		gradientpainttransformtype	39.00		
		repaymentty	39.98		
10	getCount	supercount	72.69	7	3
		count	71.35		
		parentconfigurationnotapplicableexception	58.65		
		size	58.37		
		istoolargetobeconvertedtoanint	48.36		
		uniqueresult	44.77		
		modifiedfids	41.55		
		msectioncache	41.19		
		tertiary	42.71		
		gettotalcount	42.73		

5.3.2 Normalized Code2Vec clones

The result of the normalized Code2Vec code clone detection is presented in this section, which has been verified from InspectorClone [17]. Figure 5.1 represents the true positive clone and Figure 5.2 shows the false positive case detected using Code2Vec model from BigCloneBeanch dataset.

```
public static int getIndex(int arr[], int t)
    {

        // if array is Null
        if (arr == null) {
            return -1;
        }

        // find length of array
        int len = arr.length;
        int i = 0;


        // traverse in the array
        while (i < len) {

            // if the i-th element is t
            // then return the index
            if (arr[i] == t) {
                return i;
            }
            else {
                i = i + 1;
            }
        }
        return -1;
    }
```

```
public static int indexOf(int arr[], int t)
    {
        int len = arr.length;
        return IntStream.range(0, len)
            .filter(i -> t == arr[i])
            .findFirst() // first occurrence
            .orElse(-1); // No element found
    }
```

Figure 5.1: True Positive Clone detected by Code2Vec and verified manually

```
static void reverse(int a[], int n) {
        int[] b = new int[n];
        int j = n;
        for (int i = 0; i < n; i++) {
            b[j - 1] = a[i];
            j = j - 1;
                                    }
}
```

```
static void reversedreaction (int origReaction[], int k) {
        double[] b = new double[k];
        for (int i = 0; i < k; i++) {
            b[i] = 1/origReaction[i];
        }
}
```

Figure 5.2: False positive Clone detected for reverse method

Table 5-2: Results of normalized Code2Vec for 10 Java methods (selected from https://code2vec.org) and their Top10 predicted clones with both manual examination and InspectorClone and MeasurePrecision values of TP (true positive) and FP (false positives).

				Manual Test		**InspectorClone**	
	Function	**Top10 Clones**	**Similarity %**	**TP**	**FP**	**TP**	**FP**
1	sort	sortdefinition	61.11	10	0	10	0
		currentsorts	57.27				
		nativenumbersort	55.76				
		sortx	54.44				
		numperslot	54.51				
		sorted	53.09				
		blockcomparator	52.89				
		introsort	51.64				
		msort	50.96				
2	contains	containskey	65.5	4	6	4	6
		screentoclient	52.4				
		isinaabb	48.07				
		leafarray	48.04				
		containsgroup	46.33				
		stackobject	45.59				
		flatteningpathiterator	45.59				
		rectcrossings	45.50				
		mpartialpolyline	45.11				
		janesmithcontains	45.02				
3	getindex	indexof	59.31	10	0	10	0

		findex	51.63				
		indexeclass	51.27				
		getindexforname	49.4				
		index	48.23				
		searchindexwithindexingprogress	47.53				
		getindextree	47.24				
		gettypeparameterindex	47.20				
		binarysearch	47.04				
		indexlarq	46.44				
4	add	myadd	64.73	8	2	9	1
		addelement	62.99				
		addcontentchangelistener	56.83				
		localadd	55.26				
		adding	55.17				
		getoffsettoadd	52.51				
		personw	51.63				
		put	49.73				
		addchild	49.34				
		addbefore	49.30				
5	reverse	reversed	53.01	7	3	7	3
		endlow	51.33				
		fronthigh	50.21				
		routeleg	49.34				
		reversedreaction	49.33				
		generatereverse	49.32				
		generalizedlemmatransformation	49.30				
		revline	49.21				
		dbdatanode	49.15				
		currentplaytime	49.02				
6	factorial	factorial	63.69	6	4	6	4
		factarray	62.19				
		longmath	61.33				
		factorials	61.31				
		productbits	59.57				
		plainbest	58.77				
		topchosen	56.32				
		isclientonlyservice	46.17				
		packageide	43.1				
		lgamma	43.0				

7	download	isdownloadauthorized	52.02	10	0	10	0
		subscriptiondownloadlistener	50.29				
		githubdownload	50.10				
		handlecommandexitstatus	49.008				
		isdownload	48.58				
		downloadlocationfile	48.49				
		downloadfile	47.78				
		readthefile	47.34				
		deviceofflinedownload	47.04				
		downloadableartifacts	47.03				
8	postrequest	post	65.59	1	9	3	7
		senddate	65.65				
		gettotalrevenue	42.97				
		updatecheckincomment	42.67				
		directaction	42.03				
		spotifyapiexception	41.22				
		createwriternotcalledtime	40.71				
		mcthreadlock	40.19				
		setpagedlistinfo	39.76				
		disabledbankaccountexception	39.55				
9	equals	equalsignorecase	79.72	6	4	6	4
		equal	78.23				
		setdailysubtotal	59.72				
		dateresattr	53.01				
		rioret	52.76				
		eq	52.74				
		equalfilters	52.11				
		fieldlayouts	52.01				
		gradientpainttransformtype	52				
		repaymentty	51.98				
10	getCount	supercount	57.09	7	3	8	2
		count	51.35				
		parentconfigurationnotapplicableexception	48.65				

		size	48.37				
		istoolargetobeconvertedtoanint	48.36				
		mshowfooterview	47.77				
		mhistorycount	47.56				
		msectioncache	47.16				
		maxcasts	46.669				
		gettotalcount	46.665				

In Table 5.2 the results are shown for manually inspected clone results of the normalized Code2Vec model [1]. The results are verified using Measure precision. Measure precision is a java package that computes if two code snippets are clones or not. It shows the results in the format of a true positive or a false positive.

5.3.3 Normalization validation

In this section, I compare baseline Code2Vec i.e., without variable name change code clone results with the results obtained from normalized Code2Vec with variable name change implementation. This work is done to validate the design decision made in Chapter 4-Experimental Design based on example and observation on 'sort' Java method on code2vec website. Table 5.1 presents the results obtained on baseline Code2Vec for 10 Java methods selected from code2vec website. The same methods have been used to obtain results of Table 5.2, where the top 10 clones are shown for normalized Code2Vec model. The clones are then marked as true positive and false positive using MeasurePrecision jar file.

Table 5-3: Comparison of clone detection results for Baseline and Normalized Code2Vec for top10 predicted clones

Method	**Baseline Code2Vec**			**Normalized Code2Vec**		
	TP	**FP**	**Precision**	**TP**	**FP**	**Precision**
sort	8	2	80%	10	0	100%
contains	2	8	20%	4	6	40%
getIndex	7	3	70%	10	0	100%
add	8	2	80%	9	1	90%
reverse	7	3	70%	7	3	70%
factorial	4	6	40%	6	4	60%
download	8	2	80%	10	0	100%
postRequest	3	7	30%	3	7	30%
equals	5	5	50%	6	4	60%
getCount	7	3	70%	8	2	80%

Table 5.3 shows the comparison of precision for baseline Code2Vec and normalized Code2Vec for 10 Java methods previously examined. It is observed that the false positive rate is lower for normalized Code2Vec approach.

5.3.2 Effect of 'k' on precision

As we see that the value of 'K' in topk predictions, affects the precision and recall. I developed a script that does predictions for 100 methods with different values of 'K' to find the balance between precision and recall. Table 5.4 shows the results of this experiment.

Table 5-4: Relation of K value and Precision value over 100 examples

Value of K in topk prediction	**Precision Value average over 100 examples**
1	100%
5	80%
7	63%
10	57%

5.3.3 Recall

To measure recall, BigCloneBench dataset [5] is used. This dataset has manually tagged pairs. The IJaDataset is used with Code2Vec to find clones. If the Code2Vec top1 prediction is the same as BigCloneBench tagged method in clone pair, then it is considered successful clone detection and is marked as 'T' i.e., true, otherwise it is marked as 'F' i.e., false. Some examples from this experimentation are displayed in Table 5.5. This approach is still rudimentary and therefore needs more work, which will be focused on in the future work section. This will provide more information on the number of types of clones that were identified in the already manually tagged clone pairs. The future work will include running

Code2Vec for the first method in BigCloneBench tagged clone pair and checking if the top prediction matched the second method of tagged clone pair.

Table 5-5: Result of Code2Vec and BigCloneBench clone pair verification. C: True or False if top1 predicted clone is the second method in BigCloneBench tagged clone pair

BigCloneBench Tagged Pair Clone1			**BigCloneBench Tagged Pair Clone2**			**C**
method1_file	method1_startline	method1_endline	method2_file	method2_startline	method2_endline	
355856.java	598	615	355856.java	617	634	F
137408.java	346	377	118496.java	362	393	T
67675.java	606	619	44208.java	604	617	T
5593.java	208	227	97228.java	208	227	T
1090867.java	670	680	1321892.java	625	635	F
69184.java	184	198	85894.java	197	213	F
1484507.java	255	270	1484507.java	531	548	T

5.3.4 Clone threshold computation

Clone threshold is defined as the cutoff value of similarity between original function and predicted clone. The code clone above this threshold value is further checked if it is a true positive or a false positive. The threshold value is taken in a range of 0.4 to 0.9 to find the accuracy or precision of clone prediction. The pseudocode to compute the results is mentioned in Figure 5.3. Using Gensim python package, the most_similar code vectors are

fetched with the method name information and the similarity percentage value. Then the resultant code vectors are considered clones if the similarity percentage value is greater than or equal to the threshold value. These clones are further verified using MeasurePrecision jar file and the result is logged as true positive or false positive. This result is used to compute the Precision value. It is observed that as the threshold is increased, the true positive rate increases and false positive rate decreases, but the number of clones found is also reduced. This impacts the recall value of the system because not all clones are found due to higher similarity threshold cap. As the threshold value is reduced to 0.45, in some cases, false positive clones show up. Therefore, precision reduces at a very low threshold value. The bar graph comparing the manually inspected threshold value for three manually selected functions is shown in Figure 5.4. In order to scale the results to a greater number of examples, the graph shown in Figure 5.5 is drawn for 10 examples. The threshold value and precision results are inspected for all methods using similar computation as given in Table 5.6 for 3 example methods. Overall, the 10 methods extracted are from the dataset as mentioned in Table 5.2 previously. These are the methods mentioned on http://code2vec.org website. For different threshold values, the true and false positives for these method clones are checked and precision value is computed. Now, we have the false positives percentage at different threshold values for each of the 10 methods. To aggregate the results over the 10 methods, I take an average of false positive rate for each threshold value. These results are shown in the Figure 5.5.

The experiment on 10 functions with threshold value ranging from 0.4 to 0.9 show that clones detected with 0.7 threshold similarity value are true positives. The precision

value is 100% for this threshold. Computation of recall based on threshold values is a future work.

```
def find_clones(model, code_vectors, threshold):
        res = [item for item in model.most_similar(code_vector, topn=len(model.vectors)) if
        item[1] > threshold]
 return res
```

Figure 5.3: Pseudocode to filter clones based on threshold value

Table 5-6: Precision value for different clone similarity thresholds of different functions

Function	**Threshold**	**Total Clones**	**TP**	**FP**	**Precision**
1. sort	0.4	17	13	4	76.4%
	0.5	13	13	0	100%
2. contains	0.45	10	4	6	40%
	0.48	4	4	0	100%
	0.5	2	2	0	100%
3. reverse	0.49	10	7	3	70%
	0.493	7	6	1	85.71%
	0.5	3	3	0	100%

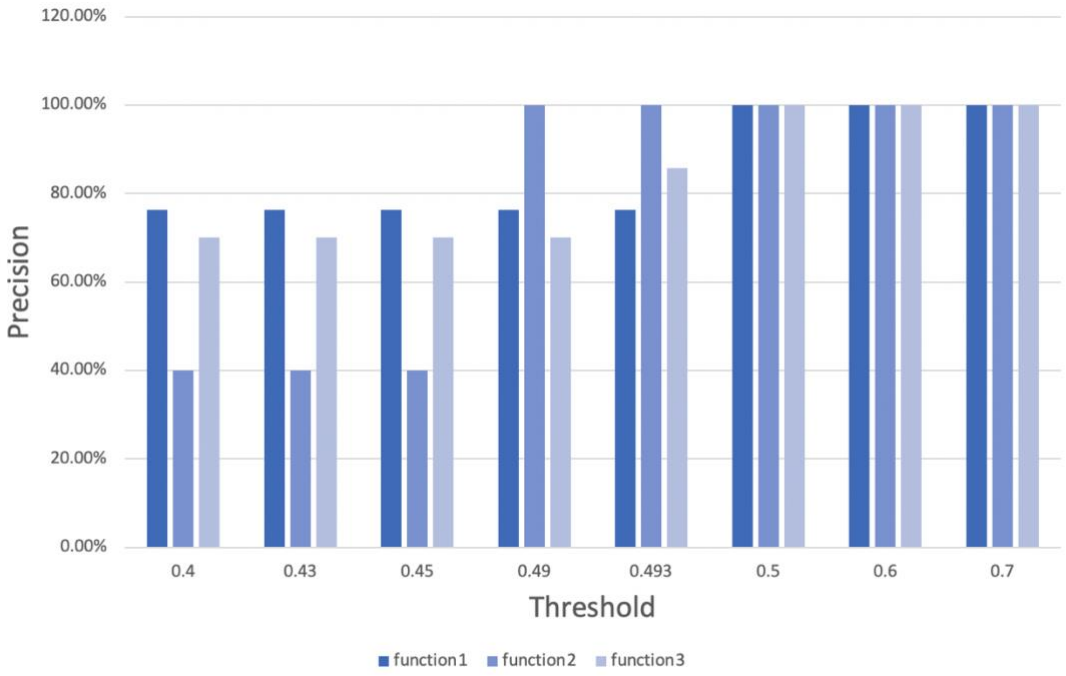


Figure 5.4: Result of Precision values for 3 functions with different thresholds

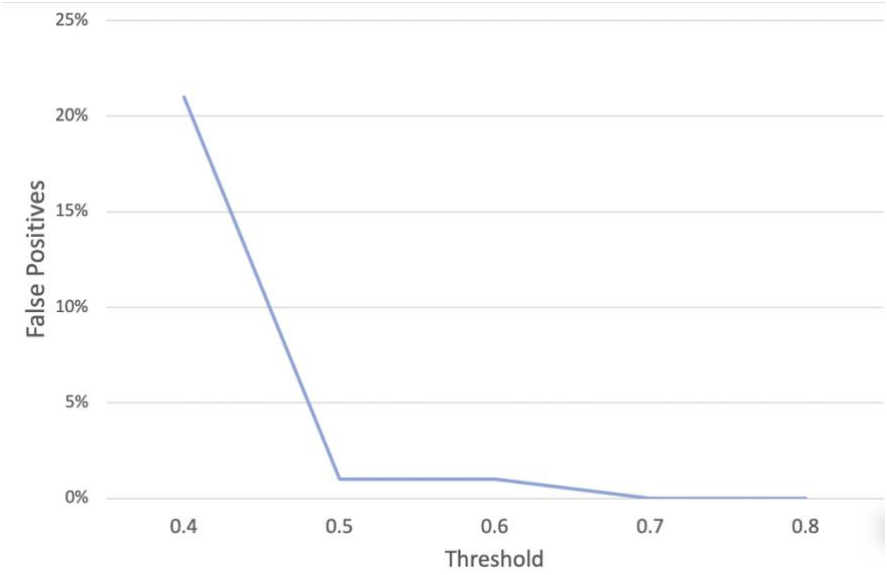


Figure 5.5: Averaged false positive for 10 manually selected methods (given in Table 5.2) inspected with different threshold values

5.4 Evaluation and Analysis

This chapter focuses on the evaluation of results showcased in the previous sections of this chapter. The effectiveness of clone detection tools is usually evaluated in terms of precision and recall. Precision is the percentage of true positives (clone pairs) within a set of code pieces identified by the tool as clones. The recall is the percentage of true positives that are retrieved by the tool within the complete set of known clones.

The measurement of precision and recall, in general, relies on the existence of labeled datasets. A well labeled dataset provides realistic data and authentic labels on all the constituents that should be detected by the analysis tool – in the case of clone detection, all clone pairs are labeled as such. The results were matched for manually tested methods and predicted clones. The evaluation was done using the Measure Precision jar file available open source. I searched for top10 clones using Measure Precision jar which takes clones and produces if the clones are a true positive result or a false positive result. The Measure Precision was used to verify results of InspectorClone [17] at inspectorclone.org, where the clones detected by InspectorClone were verified through MeasurePrecision as a true positive or a false positive.

I also did experiments on the threshold or cutoff value of similarity percentage beyond which a high precision is achieved, and all true positive clones are detected. It is observed with a set of preliminary results that at 0.7 threshold similarity value all true positive clones are detected. Therefore, the precision is 100% for 0.7 threshold.

It is also observed on the sample methods that the precision for normalized Code2Vec is higher than baseline Code2Vec. This is due to importance of variable names when a code snippet is represented as AST path to form a path embedding. If names used are not proper, then a wrong method gets predicted in baseline Code2Vec. This is analyzed using results presented in the section above.

Further Analysis of why false positives were detected by baseline Code2Vec is ongoing and will be a part of future work. This will require manual study of false positive examples and experiments on variable name. Also, this analysis will be concrete once it is extended to a larger dataset where consistency of the results is observed.

Chapter 6

Future Work and Conclusion

6.1 Summary

In this thesis, I have normalized the code2vec neural network model that learns semantics from code snippets to predict clones. I have demonstrated that normalized neural model Code2Vec can improve the performance in searching for clones in code snippets in comparison to baseline Code2Vec. I also evaluated the results based on previous benchmark models like InspectorClone [17] and BigCloneBench [5]. I have also showcased how precision and recall can be computed for Code2Vec results. The study was heavily influenced and inspired by Code2Vec [1], SourcererCC [14], InspectorClone [17]. I aim to improve my solution in the future through the iterative development of the designed model, and refining techniques.

6.2 Future Work

6.2.1 Programming Languages

In this thesis, the experiments are performed on system code GitHub repositories in Java Programming language. In the future, this has the scope of extending to other programming languages with modification in the model and the script. This would include extracting a bag of paths by constructing Abstract Syntax Trees for additional languages. Thus, language extension is one goal. PathMiner [3] has provided reusable artifacts to mine path-based code representation for different programming languages, which will be useful when extending on other languages.

6.2.2 Dataset Experimentation

Another future work includes computing the recall value of the complete IJaDataset. This will highlight what clone types are missing or not properly detected with Code2Vec bag of paths and AST based approach. This will help in learning how to improve the current representation of code or mutate the representation in a more effective way. Representing source code in the form of directed acyclic graphs instead of paths could also lead to a neural model and improve knowledge in the semantic extraction of code.

Experiments like replicating the model implementation to work with different datasets will offer more learning as it will increase the training data for existing models. Neural models improve the ability to predict when fed with more training data.

6.2.3 Analysis of False Positives

The accuracy of the results is impacted by false positives that are seen when the threshold is reduced to a lower range. In this section, I discuss what could be the plausible reasons for the occurrence of false positive clones above certain thresholds.

First reason is that the Code2Vec model [1] used for clone detection represents AST based path embeddings in a certain manner. There can be more efficient ways to create path embeddings which will be explored in future work. A detailed analysis of path vector representation can be a factor that will help understand the occurrence of false positive.

Secondly, the cosine similarity is a measure of similarity between two path vectors in a vector space. Alternative approaches to compare vectors will be explored in the future. This will provide some insight in scope of reducing false positive rate.

Thirdly, data preprocessing is a major factor to improve the accuracy of results. The code snippets for false positives will be manually tested and AST drawn to understand the reason behind prediction of such methods.

6.2.4 Application Design

As there is existing research on generating comments for code using the Code2Vec model [1], some engineering on this aspect can help create code from pseudocode. This will help advance the future of software development and programming in general. The

challenge here is the lack of training examples in the form of pseudocode and real code. Application of techniques of extracting features from code is a growing area of research and has ample of other opportunities to explore.

References

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

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A General Path-Based Representation for Predicting Program Properties. Proceedings of the ACM on Programming Languages, POPL. April 2018.

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

[4] Jeffrey Svajlenko and Chanchal Kumar Roy. Evaluating Clone Detection Tools with BigCloneBench. In Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME). 2015.

[5] Jeffrey Svajlenko and Chanchal K Roy. BigCloneEval: A clone detection tool evaluation framework with bigclonebench. In Proceedings of 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE 2016.

[6] Vaibhav Saini, Farima F., Cristina V. Lopes, Oreo: Detection of Clones in the Twilight Zone. Proceedings of 2018 ACM on European Software Engineering Symposium on Foundation of Software Engineering. Sept 2018.

[7] White Martin, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep Learning Code Fragments for Code Clone Detection. Proceedings of ASE'16 Singapore. Sept 2016

[8] Wenhan Wang, Ge Li, Bo Ma, Xin Xia,and Zhi Jin. Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree. Feb 2020.

[9] A. Sheneamer and J. Kalita. 2016. Semantic Clone Detection Using Machine Learning. In Proceedings of the 15th IEEE International Conference on Machine Learning and Applications (ICMLA). 2016.

[10] Abdullah Sheneamer and Jugal Kalita. 2016. A Survey of Software Clone Detection Techniques. International Journal of Computer Applications. 2016.

[11] Liquing Li, He Fend, Wenjie Zhuang, Na Meng. CCLearner: A Deep Learning-Based Clone Detection Approach. 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). Sept 2017.

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

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

[14] Hitesh Sajnaniy Vaibhav Sainiy Jeffrey Svajlenkoz Chanchal K. Royz Cristina V. Lopes. SourcererCC: Scaling Code Clone Detection to Big-Code. 2016 IEEE/ACM 38th IEEE International Conference on Software Engineering. 2016.

[15] http://mondego.ics.uci.edu/projects/jbf/

[16] Ira Baxter, Andrew Yahin, Leonardo de Moura, and Marcelo Sant'Anna. Clone Detection Using Abstract Syntax Trees. 10.1109/ICSM.1998.738528. Jan 1998.

[17] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu Di Yang, Pedro Martins, Hitesh Sajnani, Pierre Baldi, Cristina V. Lopes. InspectorClone: Towards Automating Precision Studies of Clone Detectors. arXiv:1812.05195v2 [cs.SE] 14 Dec 2018.
[18] T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, S. Kawaguchi, and H. Iida, "Shinobi: A real-time code clone detection tool for software maintenance," Nara Institute of Science and Technology, p. 26, 2008.
[19] Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Deanet. Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781. 2013.
[20] 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.
[21] Xin Rong. word2vec parameter learning explained. arXiv preprint arXiv:1411.2738. 2014.
[22] 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.
[23] Susan A. Dart, Robert J. Ellison, Peter H. Feiler, and A. Nico Habermann. Overview of Software Development Environments.
[24] 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.
[25] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. arXiv preprint arXiv:1711.00740, 2017.
[26] A.L. Maas, R.E. Daly, P.T. Pham, D. Huang, A.Y. Ng, and C. Potts. Learning word vectors for sentiment analysis. In Proceedings of ACL, 2011.
[27] Eric H. Huang, R. Socher, C. D. Manning and Andrew Y. Ng. Improving Word Representations via Global Context and Multiple Word Prototypes. In: Proc. Association for Computational Linguistics, 2012.
[28] Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Deanet, Ilya Sutskever. Distributed Representations of Words and Phrases and their Compositionality. arXiv preprint arXiv:1301.3781. 2013.
[29] Pedro Martins Rohan Achar Cristina V. Lopes. SourcererJBF: The Java Build Framework: Large Scale Compilation. UCI-ISR-18-3. April 2018.
[30] Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, Mauro Pezz., and Paolo Tonella. Search-based synthesis of equivalent method sequences. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2014.
[31] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering. 2002.
[32] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In Proceedings of 13th Working Conference on Reverse Engineering. 2006.
[33] Y. Zhang, R. Jin, and Z.-H. Zhou. Understanding bag-of-words model: a statistical framework. International Journal of Machine Learning and Cybernetics. 2010.
[34] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia. Problems creating task-relevant clone detection reference data. In Proceedings of WCRE. 2003.

[35] R. Collobert and J. Weston. A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning. In International Conference on Machine Learning, ICML. 2008.
[36] Seulbae Kim, Seunghoon Woo, Heejo Lee, Hakjoo Oh. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In proceedings of IEEE Symposium on Security and Privacy (SP). May 2017.