Predicting code time complexity using code2vec

Narendra Kumar

Final Thesis Report

January 2021

# TABLE OF CONTENTS

ABSTRACT

LIST OF FIGURES

CHAPTER 1: INTRODUCTION……………………………………………………………...1

1.1 Background of the Study……………………………………………………………..1

1.2 Problem Statement…………………………………………………………………...2

1.3 Aim and Objectives…………………………………………………………………..3

1.4 Scope of the Study……………………………………………………………………3

1.5 Significance of the Study…………………………………………………………….3

1.6 Structure of the Study………………………………………………………………...4

CHAPTER 2: LITERATURE REVIEW………………………………………………………5

2.1 Introduction…………………………………………………………………………..5

2.2 Asymptotic analysis……………………………………………………………….....5

2.2.1 What is asymptotic code analysis………………………………………………..5

2.2.2 Why time complexity…………………………………………………………….6

2.2.3 Types of time complexity………………………………………………………..6

2.2.4 General functions with Big-O notation…………………………………………..6

2.2.5 Applications of code analysis……………………………………………………8

2.3 Code datasets……………………………………………………………………….11

2.3.1 GitHub Archive…………………………………………………………………12

2.3.2 Scraping coding competitions website………………………………………….12

2.3.3 Challenges………………………………………………………………………12

2.4 Embeddings…………………………………………………………………………12

2.4.1 Word embeddings……………………………………………………………...13

2.4.2 Word2vec………………………………………………………………………13

2.4.3 Code embedding……………………………………………………………….14

2.4.3.1 Using word2vec with source codes……………………………………....14

2.4.3.2 Code embeddings using other methods………………………………….15

CHAPTER 3: RESEARCH METHODOLOGY……………………………………………..25

3.1 Introduction…………………………………………………………………………25

3.2 Research Approach…………………………………………………………………25

3.2.1 Data Selection………………………………………………………………….25

3.2.2 Data Collection………………………………………………………………...26

3.2.3 Data Preprocessing……………………………………………………………..28

3.2.4 Data Transformation…………………………………………………………...28

3.2.4.1 Why code2vec…………………………………………………………….28

3.2.5 Proposed Approach…………………………………………………………….30

3.2.5.1 Selection Of Classifier Algorithms……………………………………...32

3.2.6 Evaluation……………………………………………………………………...32

3.3 Tools………………………………………………………………………………...33

3.3.1 Python 3.7………………………………………………………………………33

3.3.2 TensorFlow 2.0…………………………………………………………………33

3.3.3 GPU CUDA 10.0……………………………………………………………….33

3.3.4 Jupyter Notebook 6.0.1…………………………………………………………33

3.3.5 Astminer………………………………………………………………………...33

3.3.6 Scikit-learn 0.21.2………………………………………………………………33

3.4 Summary 33

CHAPTER 4: ANALYSIS AND DESIGN 35

4.1 Introduction 35

4.2 Dataset Description 35

4.3 Dataset Distribution 35

4.4 Exploatory Data Analysis 36

4.5 Data Preparation 41

4.5.1 Data Split 41

4.5.1 Using astminer library 42

4.6 Data Preprocessing 43

4.6 Model Architecture 45

4.6 Model Configuration 46

4.6 Model Training 47

4.6 Model Hyperparameters 48

4.6 Summary 49

CHAPTER 5: RESULTS AND DISCUSSIONS 50

5.1 Introduction 50

5.2 Output 50

5.3 Evaluation Metrics And Results 50

5.3.1 Using TensorFlow 51

5.3.2 Using keras 52

5.4 Inetractive Prediction 53

5.5 Comparing Model Performance 55

5.6 Summary 56

CHAPTER 6: CONCLUSION AND RECOMMENDATIONS 57

4.1 Introduction 57

4.2 Discussion and Conclusion 57

4.3 Contribution To Knowledge 58

4.4 Future Recommendations 58

REFERENCES 59

APPENDIX A: RESEARCH PROPOSAL 66

**Abstract**

Estimating time complexity of any program is a difficult task. It needs an experienced programmer to be able to correctly judge the asymptotic upper bound of any given code. Determining this is a very subtle process and needs comprehensible knowledge of different data structures and the way they are implemented in a given programming language. Turing Halting problem states that its practically impossible to estimate running time complexity. Hence an approximated analysis of code gives programmers an overview of how their code performs on any given machine of any configuration. Such intrinsic nature of this problem makes it suitable for machine learning implementation. This paper uses CoRCod dataset, discussed in this paper(Sikka et al., 2020) for training a code2vec model, which is expected to learn from the data and predict time complexity of any unseen program. Solutions like these can be very useful for any automatic grading of online programming contests, static analysis of code in IDE and others. This research studies only the upper bound of any program, also known as worst case time complexity and denoted by O(Big Oh) notation.

**LIST OF FIGURES**

Figure 2.1 Big-Oh complexity chart…………………………………………………………...7

Figure 2.2 Function and time comparison……………………………………………………...8

Figure 2.3 Execution time of functions………………………………………………………...9

Figure 2.4 Word embeddings……………………………………………………....................13

Figure 2.5 Code and control flow graphs……………………………………………………..14

Figure 2.6 Word2vec embeddings example…………………………………………………..15

Figure 2.7 Code summary and caption………………………………………………………..16

Figure 2.8 Code and AST example………………………………………………...................16

Figure 2.9 Method name example…………………………………………………………….17

Figure 2.10 Similar method distinction……………………………………………………….18

Figure 2.11 AST with attention mechanism…………………………………………………..18

Figure 2.12 Bad method name example………………………………………………………19

Figure 2.13 Func2vec………………………………………………………………………...19

Figure 2.14 Hyperbolic function embeddings……………………………………...................20

Figure 2.14 Bug Prediction Example…………………………………………………………21

Figure 2.16 DeepAPI…………………………………………………………………………22

Figure 2.17 DeepAPI architecture……………………………………………………………22

Figure 2.18 DeepBugs………………………………………………………………………..23

Figure 2.19 Inst2vec………………………………………………………………………….23

Figure 3.1 Data asset………………………………………………………………………….25

Figure 3.2 AST example……………………………………………………………………...27

Figure 3.3 ASTVisitor code example…………………………………………………………27

Figure 3.4 Flowchart of data processing……………………………………………………...28

Figure 3.5 AST with attention scores…………………………………………………………29

Figure 3.6 Code2vec architecture…………………………………………………………….30

Figure 4.1 Features from code samples……………………………………………………….35

Figure 4.2 Data distribution pie chart…………………………………………………………36

Figure 4.3 No of if’s versus Complexity……………………………………………………...37

Figure 4.4 hash map versus complexity………………………………………………………37

Figure 4.5 No of jumps versus Complexity…………………………………………………...38

Figure 4.6 No of methods versus complexity…………………………………………………39

Figure 4.7 Recursion versus complexity……………………………………………………..39

Figure 4.8 Nested loop depth versus complexity……………………………………………...40

Figure 4.9 Data split architecture……………………………………………………………..41

Figure 4.10 Processing pipeline architecture…………………………………………………43

Figure 4.11 Attention framework architecture……………………………………………......45

Figure 4.12 Training Layer architecture……………………………………………...............46

Figure 5.1 Input 1……………………………………………………………………………..53

Figure 5.2 Keras model output ……………………………………………………………….53

Figure 5.3 TensorFlow output………………………………………………………………..54

Figure 5.4 Input 2……………………………………………………………………………..54

Figure 5.5 Output 2…………………………………………………………………………...55

**CHAPTER 1**

**INTRODUCTION**

* 1. **Background of the Study**

Time complexity of an algorithm is the amount of runtime, an algorithm needs to finish execution without any error. Formally it is measured as an order of growth of a function of the given input size. This research studies worst case time complexity, which is also known as the upper bound time complexity and are represented by O(Big Oh) notation. For example shows the asymptotic upper bound of an algorithm, where n is the input size.

Measuring time complexity of an algorithm is an arduous task, since it requires significant amount of effort to figure them out. There are several manual methods to compute them, but either they are very problem specific or are error prone. For example Master’s theorem(Bentley et al., 1980) is limited to the divide and conquer algorithms. Cyclomatic complexity(McCabe, 2020) uses call graphs to determine the logic strength of an algorithm but doesn’t account for how many times a specific path in the graph is called. This leaves a question with its robustness while analysing different kind of algorithms.

Turing halting problem proves that it is mathematically impossible to estimate code complexity. Rice’s theorem and similar studies have also shown that there can’t be any general mathematical equation, which can be formulated for any code with polynomial order complexity. These limitations make this problem ideal for machine learning. For data requirements there are many online sources like Public Git Archive, stack overflow, which hosts a large amount of dataset to be explored.

Research done by(Sikka et al., 2020) does an excellent work in terms of creating a dataset, which maps a program to its respective complexity. It achieves motivating results using traditional machine learning models, along with using graph2vec(Narayanan et al., 2017) for code embeddings. It also proposes results for all baseline classifier models, along with the study of features which has high effect on the computed complexity class. Data generated for learning took a lot of manual effort and hence the total number of samples generated are limited. Although the dataset is balanced and has little bias towards one class i.e. but overall it’s limited number makes it not suitable for exploring deep learning models.

Distributed code representations uses AST of the code to generate graphs and uses the same to generate a vector for training a machine learning model. Overall this study tries to contribute towards:

* Introducing another approach using code2vec model.
* Use code2vec(Alon et al., 2019) neural network for generating code embeddings and code vectors.
* Implement baseline neural network models and verify it’s performance.
  1. **Problem Statement**

Any software written nowadays goes through several stages of review process. Maintaining overall code quality is the sole responsibility of developer and his/her team. Compulsory peer review of the code is a must before getting approval for merging the codebase into the main repository. Most of this review process is manual today, where it presents a lot of scope of automation. CI/CD pipelines are configured for smooth flow of code being developed and shipped to the servers and clients.

Code review process generally takes a lot of time to finish, since almost all the processes are manual and also depends on the time availability of the person responsible for maintenance. Manual evaluation are also prone to human errors, where one can easily miss basic coding horrors like bad variable names, incorrect indentation, methods defined but never used etc. These can be taken care by using static code analysers but issues like memory leak in the code, badly optimized code are easy to miss and can only be done with the help of experienced developers. Measuring time complexity of the source code is one of those parameters, which is a highly manual effort at present. There are no tools available which can correctly predict the order by which a developer’s code can run in different environments. This thesis tries to fill in this void and improve on work done by (Sikka et al., 2020). This is hopefully a great start on building a tool which can straight away predict the order of magnitude by which execution time grows with respect to the input size. Few gaps identified in this paper like converting the code into AST nodes and then converting them into code embeddings can be improved using other code embedding techniques like code2vec etc. Code2vec gives a single vector considering various other factors, which are discussed in detail in literature review section.

**1.3 Aims and Objectives**

The main aim of this research is to propose a model which predicts time complexity of the software code based on its intrinsic features. These features can be number of if statements, number of for loops, nested loop depth among other common code constructs. Although these features give a lot of inner detail on what contributes more towards a particular time complexity but they don’t capture any semantics. For that purpose, code embeddings will be used. The goal of this research is to come up with a better performing model using code2vec(code2vec github, 2018), which is a neural network for learning distributed representations of code.

The research objectives are formulated based on the aim of study which are as follows:

* To analyse the important features contributing to code’s time complexity.
* To suggest a suitable data generation method that can be fed to code2vec model.
* To compare code2vec neural network’s performance on the data asset.
* To evaluate neural net machine learning models using TensorFlow and Keras.

**1.4 Scope of the Study**

Scope of this thesis is limited to predicting time complexity of the code. Code running time also depends on the configuration of system, it is running on. Systems having higher clock speed, better RAM are expected to execute the code instructions faster. Hence same algorithm can show different running times in different environment. Kernel processes can also change the runtime based on how prioritised the current process is. Code instructions having high priority instructions will be grabbed first than those having low priority. Basic assumptions like constant time lookup for a variable, insertion and deletion are inherently present in the current study. Other concepts like space complexity are not considered, instead main focus is on studying the order of growth based on the input size.

* 1. **Significance of the Study**

Even though the cost of computing resources have become relatively cheaper than few decades ago, one can never deny over the advantages of software running at its optimum level. Optimized software is easy to scale and overall maintainability of the code improves manifold. One of the metrics to evaluate a software code is to measure its time complexity or order of growth with respect to its input size. Codes having lower order of growth can be considered well implemented. This study tries to implement a model, which can predict the time complexity of the code, having code itself as an input. Most of the IDE’s(Interactive development environment) can use such models as a plugin to automatically inform programmers about the complexity. This can be a static analyser tool for quick feedback. Almost the continuous integration tools like Circle CI, Jenkins etc can also leverage this tool to run builds on their servers and post results for the same to the author. Many online contests can also take advantage of it to automatically generate results, since proposing the best solution is almost always one the criterion for the test cases attached to a problem.

**1.6 Structure of the Study**

This thesis is structured as follows. Chapter 1 begins with the background of the study in section 1.1 and discusses the problem statement in section 1.2, followed by aims and objectives of the study in section 1.2 and 1.3 respectively. Last section talks about the significance of this study.

Chapter 2 begins with the introduction of the subject with deeper dive into the concepts of time complexity analysis in section 2.2. Section 2.3 looks for different code datasets available and how they can be utilized for future studies on the same subject this thesis tries to study. Section 2.4 dives in deep on ideas and philosophy of embeddings and how they can be utilized for software code. Section 2.5 talks about the challenges in embedding space followed by the summary of the chapter 2 in section 2.6.

Chapter 3 also begins with an introduction of research methodology this thesis adopts followed by the approach which the study will take upon. That’s discussed in section 3.2. Section 3.2 also discusses about the data, why is it good for this study and what kind of inherent challenges it brings with it. Idea and whole process flowchart is also submitted in this section, which gives a brief idea on the implementation of the model. Later sections 3.3 and 3.4 mentions the potential tools and summary of the section in brief.

Chapter 4 analyzes features present in the dataset and explores its distribution. Model’s configuration and it’s architecture are also discussed in detail. New data processing pipeline is introduced next along with the shapes and sizes of data flow at each training iteration. Interactive prediction results are added for better visualization of results.

Chapter 5 discusses the results and contains explanations for model’s performance.

Chapter 6 ends with the final conclusion and recommendations on possible future works.

**CHAPTER 2**

**LITERATURE REVIEW**

**2.1 Introduction**

Time complexity of a given source code is a complex task. It needs experienced coders and developers to go through the complete program structure and then estimate its upper bound(also denoted by Big Oh) execution time. There are no algorithms which can efficiently predict such functions and hence manual intervention becomes almost always necessary. Since most of these analysis are manual, this becomes a classic use case for machine learning. In the last few years there has been a large number of studies falling into the category of using embeddings to be able to use source code for training purposes. Different kind of code embedding techniques are utilized in various example use cases and have been found useful for learning prediction tasks. This study focused on how these techniques have been applied to different domains of open source community and tries to use the learnings accumulated from those for predicting time complexities of given source code.

**2.2 Asymptotic analysis**

Asymptotic analysis generally refers to study which is input bound. Given an input of finite size, execution runtime is analyzed in terms of its growth with respect to the size of input data.

They are denoted in terms of a function of the given input size. When such study is done for software code, it may always be thought as time complexity of the given piece of program. This program can take inputs and its time complexity measures the runtime via a function of the input given to the code or any algorithm.

**2.2.1 What is asymptotic code analysis**

In this study asymptotic code analysis refers to time complexity. It should not be confused with the time taken to finish execution of a program. It would rather mean to calculate runtime growth of the function taking some input of size n. Space complexity of a program can also be measured with similar approach but that is out of scope for this study. Time complexity depends on several other factors i.e. processor speed, ram, Operating system etc. This study assumes that all kind of input/output operations are done in constant time, hence we only measure the complexity with respect to the input size.

This study also assumes that the analysis of any computational problems are of category, where it can be said that they are going to be finished in finite amount of time. In other words, the problems encountered here are Turing complete and has a feasible solution. (Stanford encyclopedia, 2020) digs in deep about such classes of problem and gives a complete insight over computational theory.

**2.2.2 Why time complexity**

It’s true that time taken by a program depends on several other factors of computer systems. Given piece of program can run a lot faster on super computers than on a normal desktop systems. (Kumarsharma et al., 2018) shows a comprehensive study of how these factors affect the execution time. Since there are a lot of variables involved here, measuring time complexity based on input size becomes the standardized way of measurement.

There could be other domains of studies where other factors can be included to learn better strategies. In search domain where application of faster algorithms is very important, (Lehre and Özcan, 2013) suggests that mixing hyper heuristics with the operators score can improve the overall performance of such hard computational problems.

**2.2.3 Types of time complexity**

Time complexity can be categorized into 3 different categories. They are

* Best case: For a given input of size n, given algorithm runs fastest.
* Average case: Algorithm is such cases are studied with random input size.
* Worst case or Big-O Notation: For a given input size n, algorithm has the slowest runtime. It is generally denoted by Big-O notation. It is also called as the upper bound for a function of input n.

**2.2.4 General functions with Big-O notation**

Figure 2.1 shows the graph of normally considered functions. It is clear from the graph, that as the input size increases, the number of operations increases and vice versa. It also shows that exponential functions have higher number of operations compared to the functions which are constant or logarithmic. It is almost always desired to have worst case complexity growing in logarithmic or linear way. Such parameters also helps programmers decide when to start looking for optimizations or places where refactoring of code is must. Higher order function like exponential functions demands more space and time to execute and hence has greater cost to maintain. Algorithms having higher order time complexity generally causes issues like stack overflow etc. which in longer run becomes very tough to debug.

This study(Understanding Time Complexity and its Importance in Technology | by Abdur Rafey Masood | Medium, 2020) shows functions which fall under green area are most desirable, having manageable order of growth in orange and worst in red. Worst case scenarios become even more challenging if the hardware limitations are imposed. Although these days most of the systems have multi core processors with GPU but those systems having limited processor speed and limited execution space can even fail to finish the program eventually. Some examples of worst case complexity contain multiple for loops, recursive loops without break condition etc.

Factorial time O(n!) is considered to be the worst and is slowest of all the functions. In the figure below, it can be seen as having the steepest slope proving that the processor has to run highest number of operations for such algorithms with a given input of size n. Such algorithms can easily cause out of memory issues, if there are multiple concurrent processes running at the same time.

Chart

Description automatically generated

**Figure 2.1 (Big-Oh complexity chart)**

**2.2.5 Application of code analysis**

Application of time complexity analysis is almost everywhere present inside the programming world. Competitive coding programs can use these predictive models for automatic evaluation of code. Several integrated development environments can also leverage it to give continuous feedback to programmers, which can help them to choose their options wisely. There are also cases of having different choices of algorithms based on the hardware limitations and the input size. For example there are a few search algorithms which might have higher order of growth for an input size but can work faster for smaller input size.

There are several research domains where studies continue to research for better algorithms over the known solutions. There are cases where the solution is known for a given problem but either the known solution is very costly or it can’t be scaled. Such studies measure their performance in terms of Big-O notation.

Another real life scenario from web development paradigm leverages for the fact that it points to scalability. General web development happens on a very small scale, where the developer works with very small dataset and develops a feature on top of that. Any algorithm designed on such low data can run very fast but the real issue comes when the same algorithm, if not considered by its time complexity, can behave a lot weird in production environments. Normally production environments deals with loads of data with heavy concurrent usage and if any algorithm with exponential complexity runs with these parameters can bring even a very god hardware to it’s knees. For example let’s say an algorithm with worst case time complexity and another with . Number of resources would scale proportionally for algorithm with exponential complexity as compared to one with linear time complexity.

Just to get an estimate of how common worst case time complexity functions grow, let’s take a look at figure 2.2 below.

Table

Description automatically generated

**Figure 2.2 (Function and time comparison)**

First column in Figure 2.2 shows the input size n and the rest of the columns shows the transformed values as a function of the input size. It’s clear from the figure that almost always the logarithmic/linear time complexities are desirable.

Figure 2.3 shows the execution times of an instruction, assuming one execution takes I milli-second. There are some drastic changes between the execution time of linear/logarithmic function, as compared to the functions having exponential/factorial growth. When we combine these illustrations with our idea of scalability, it becomes very clear that using the worst case complexity measures can save a lot of unwanted troubles in later cycles of any software applications.

Table

Description automatically generated

**Figure 2.3 (Execution time of functions)**

For NP-hard problems where there is always an approximate solution, time complexity analysis are also used to find solutions which are polynomial time. Study(Oliveto et al., 2007) on evolutionary algorithms shows promising results for such hard computational problems.

When we study algorithm’s time complexity, we can’t stay away from Moore’s law. This is because any code execution has to happen on a hardware which generally evolves around Moore’s law. (Padua et al., 2011) states that every 2 years, the number of transistors on chips will double and the chip’s size will almost go half of it’s present size. Now the number of more transistors means more information flow and also adds for information processing capabilities. In fact most of the machine learning theory progressed into well defined facts is also heavily dependent upon this evolution into processing power. With more processing power, it gave an increased possibility of even be able to develop and run complex algorithms. An incomprehensible idea that complexity means more power is greatly false, as this leads to collection of fat software which is designed to present all possible features to everyone, even though most of them might not be required in daily usage of their customers. Moreover this study (Wirth, 1995) discussed loads of idea behind having lean software and how more power should be used responsibly. It also discusses about factors like time pressure etc. causing writing of these big monolith softwires and how it doesn’t allow time for careful planning but they are out of scope for this study.

Loads of improvement in the domain of NP problems i.e. Search problems got measured their complexities using Big O notations and advancements on these are also measured using the same idea. (Papadimitriou, 2014) discussed these ideas in detail. It also discusses the potential effect of quantum computing on theory of computation.

Another study (Gomes et al., 2019) takes a deep dive into the relation between system’s parameters and it’s complexities. It tries to reach to a fact that identifying parameters based on their sensitivity analysis can help changing system’s complexity. Since the study of time complexity can also be considered as a mathematical investigation, exploiting certain theorems can also help to formulate an algorithm to have an improved logarithmic complexity, as proposed in this study (Aamir et al., 2011).

One classic example of having a measurement system of complexities in any field is recorded in history where scientists had difficulties in associating electricity and magnetism. This study (Lloyd, 2001) mentions this example and gives a non-exhaustive list of questions that needs to be asked for measuring complexities of different fields. Few questions like ‘How difficult is it to describe or create or difficulty in describing structure’ can also help in designing a simple algorithm.

Although it’s very difficult to design an architecture for diverse fields but this paper (Klir and Simon, 1991) discusses this idea in detail. Another study (Jayram et al., 2003) discusses about revealing minimum number of inputs to design a correct algorithm. This can also be termed as less amount of information complexity and reduce overall burden of returning complex outputs.

Similar to the idea of measuring complexities of algorithm, other techniques like measuring entropy of a system are also used for tasks like time series prediction and compression techniques. Even though these kind of data are expected to have some level of noise, ideas like ETC(Effort to compress) (Nagaraj et al., 2013) are in progress to improve these drawbacks. These lossless compression techniques are in use for many similar studies but has limitations over the fact that they don’t add much when compared to other entropy related ideas. To overcome such limitations another idea like using finite approximations (Soler-Toscano and Zenil, 2017) is discussed in this paper.

Another study (Gauvrit et al., 2016) discusses about an idea of finding mean complexity of an algorithm, in case finding the complexity is completely random. For example this study discusses how the idea of measuring algorithm’s complexity gave rise to an idea of randomness. Another example given in this paper is that a string is complex, is also a measure of its randomness. It also proposed a R package(ACSS) which might help psychologists understanding subjective randomness differs from algorithm’s randomness.

**2.3 Code datasets**

There are many code repositories available which has good amount of code datasets and are open source as well. (SlattonK. Clint, 2005) contains a list of software code archives from multiple disciplines of software programming. Current study focusses on CoRCoD dataset(Sikka, 2019), which is the base dataset for the research done in this paper (Sikka et al., 2020). This data contains java code for 932 programs, which are equally balanced for 5 different time complexity functions. This research focused on this formatted dataset.

There are few more similar datasets (Srikant and Aggarwal, 2014) used for research in similar domain but they are not publicly available dataset. Some publicly available datasets, which could potentially be cleaned and adapted for modelling machine learning models and predict it’s time complexity class. Usefulness of few of them are discussed below.

**2.3.1 GitHub Archive**

GitHub alone has about 10 million repositories(Soler-Toscano and Zenil, 2017), which hosts almost all the famous open source code shared by the community. It provides a rich source of data to analyze and prepare for any kind of machine learning study. Importing data for preprocessing can be directly taken from here with appropriate consideration over the quality of code or data. This study(Cosentino et al., 2016) also discussed about the advantages/disadvantage of blindly picking up data for such resources. Overall it can provide a great source of data for studies like building up a model for predicting different aspects of code i.e. time complexity etc.

**2.3.2 Scraping coding competition websites**

Many online coding competitions provide API’s to exchange data based on request. Many of them also can be scraped through their online portals. Examples of some of these online competitions are CODECHEF, TOPCODER etc.

**2.3.3 Challenges**

Main challenge in accumulating such data where code is given to you can be a very daunting task. There is every possibility of the code being scraped has some error, which would need manual check to make sure they can be used. Another issue, also seen in in (Sikka et al., 2020) is that there is no available data which maps these codes to its respective time complexity. Hence any research in this domain is very limited, which can be improved further with good quality datasets.

**2.4 Embeddings**

An embedding maps objects to its respective vectors varying in real numbers (Chen and Monperrus, 2019). This vector can be multi-dimensional having 100’s or 1000’s of dimensions which then used different ideas of dimensionality reduction to build up a desired model. Transformation of these sequence of texts to vector space makes it possible to use neural network architecture’s along with state of the art machine learning models and provide meaningful inference of the data. Models like Word2vec gave a big breakthrough for NLP research. Building on similar idea, software code can also be transformed into a multi-dimensional vector space. In a broader sense, code can also be considered as a sequence of words, grouped together with valid semantics/syntax(Hindle et al., 2012).

**2.4.1 Word embeddings**

It can be described as the set of modelling and learning techniques adopted in the field of NLP(Natural Language Processing). A corpus of data is chosen for learning, in which each word or phrase is mapped to its respective vector representation. We use these vector representations to use for learning a machine learning model. It also makes it easy to adapt learning based on evolving meaning of words(Yao et al., 2018). Figure 2.4 captures one such evolution in meaning. For example apple with time(around 2016) gets closer to other software companies like google etc.

A picture containing radar chart

Description automatically generated

**Figure 2.4 (Word embeddings)**

These vector representations brought a revolutionary change in our capability of improving in NLP domain.

**2.4.2 Word2vec**

This revolutionary paper (Mikolov et al., 2013) paved the way towards great progress in learning word vectors from the huge corpora of datasets. These datasets could have billions/trillions of words and used simple skip gram models in combination with simple machine learning architectures. This gave a great advantage over the performance over low computing cost. This model architecture is so famous that even today almost all researchers use them for their respective natural language processing studies. Very same idea has also inspired other domains i.e. code sequences etc. to use code embeddings to better understand code semantics, which lead to extraordinary use cases in software engineering paradigm. We shall look into few similar embedding approaches and its usefulness in code structures.

**2.4.3 Code embedding**

Ideas taken from word2vec can also be applied to generate embeddings for source code. Experiments done already in this field shows encouraging results. Having huge amount of opens source code data presents a great potential for machine learning models. Several deep learning research on entities like images, videos etc. computes a numerical vector, which is also replicated in source code embeddings.

Since these vectors also preserve the semantic space, the similarity between different vectors are generated by using cosine similarity or using Euclidean distance(Beyer et al., 1998). This paper dives in deep to study finding nearest neighbors in high dimensional space.

**2.4.3.1 Using word2vec with source codes**

As brilliant as it sounds, word2vec is not really limited to natural languages but can also be used with programming languages. Few studies in different programming languages have explored these and they are discussed here in brief.

(Harer et al., 2018) uses word2vec for finding security risks and vulnerabilities in the software by generating word embeddings of C/C++ code. E.g. Let’s consider the code snippet given in figure 2.4, its corresponding word2vec tokens can be visualized in figure 2.6.

Diagram

Description automatically generated

**Figure 2.5 (Code and control flow graphs)**

**Map, scatter chart

Description automatically generated**

**Figure 2.6 (Word2vec embeddings example)**

(White et al., 2019)used word2vec to generate embeddings for java code and further presents an idea to use it for auto correction within code. It used AST(abstract syntax tree) to check the order of appearance of java tokens and based on its learning suggests auto correction, wherever it finds an anomaly.

For tracking student’s performance on their programming tasks, (Azcona et al., 2019) proposes an idea of tokenizing each student’s submission as flat list of tokens and save them into a matric like data structure. In this matrix. Each row corresponds to each student’s submission. This too uses word2vec for tokenization. Another attempt on auto correction in java language is proposed(Chen and Monperrus, 2018), which uses cosine similarity index to find the nearest correct possibility.

**2.4.3.2 Code embeddings using other models**

Unlike an assumption while using word2vec models that programs can be considered as sequence of characters, another approach of using abstract syntax tree (AST) can be used to get better results. Models like CODE2SEQ (Alon et al., 2018a) started ways of exploring ast of a program for auto-completion, documentation etc. These features are really useful when integrated with an integrated development environment. This study shows an example of how it can be used for predicting method names in different programming languages i.e. java, C# etc. Figure 2.7 shows an example of that , where the first code snippet is in java and another is in C#. Snippet (a) predicts method name as ‘contains’ and (b) predicts name as ‘replace’

Graphical user interface, text, application

Description automatically generated

**Figure 2.7 (Code summary and caption)**

This study uses Abstract Syntax Trees(AST), which can be thought of as a data structure which preserves the grammar of the code and represents them into several leaf and nodes of a tree like data structure. E.g. Even if there is a minute difference in implementation but achieves the same task, then also their respective tree structure will detect the difference in syntax structure. Figure 2.8 shows an example of that.

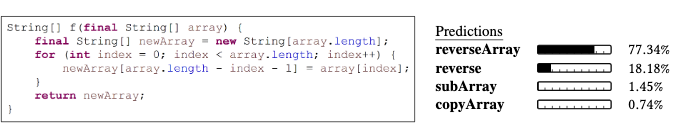
Diagram

Description automatically generated

**Figure 2.8 (Code and AST example)**

Above example shows the same implementation of counting an occurrence of a specific character in a string but the (a) uses a do-while loop whereas (b) uses a for statement. Difference in their respective syntax trees can be seen in (c) and (d).

In their subsequent paper (Alon et al., 2019) also proposed a neural model called code2vec, which generates a single fixed length vector for snippets of code. This model is actually is the base of this thesis, which tries to use the same for the CoRCoD dataset (Sikka, 2019) instead of using graph2vec(Narayanan et al., 2017). Code2vec is more apt for such problems because it tried to find the semantic relationship between different parts of the code, which is a very difficult task altogether. Further this paper uses the same task of predicting the method name and displays different probabilities of few method names it can think of. Figure 2.9 demonstrates this result.



**Figure 2.9 (Method name example)**

Another advantage of using code2vec is that it learns multiple syntactic paths within the code and forms a single vector for them. Applications of such embeddings can be utilized in many aspects of software development. For example Doing code review is almost a recurrent task for all software developers, where new methods can have weird names which might not have any relation to the method body. As stated by (Martin, 2018) in his book ‘Refactoring’, ‘If you have a good method name, you don’t need to look at its body’. In summary these code embeddings can be used for auto code review. Further it can also be used for an easy documentation search. For example let’s assume a developer searches for a documentation on method ‘m’ but it’s originally written as method ‘n’ for the simple reason that document writer found that more appropriate in any given context. Now instead of not able to find method ‘m’, a similar vector representation can easily find method ‘n’ which matches with what user was searching for.

Another advantage of code2vec is that it is uses concepts of path contexts and attention mechanism. Even if the same example is seen in the training data, embeddings recognizes the subtle differences in the code structure and makes suitable differentiation in name prediction, as shown in figure 2.10. Figure has very little syntactic difference in (a), (b) and (c) but the difference is noted by the model.



**Figure 2.10 (Similar method distinction)**

Below figure 2.11 shows the attention mechanism and it’s working. The most relevant part of the tree/path-context is given more attention and all of these contexts are within the same vector.

In this figure the path context in red has maximum attention score, followed by blue and orange.

A picture containing diagram

Description automatically generated

**Figure 2.11 (AST with attention mechanism)**

This paper (Allamanis et al., 2015) also uses embeddings to suggest descriptive class/method names. It identifies this task as a difficult inference problem, since a proper class name can depend on a very wide context of the application. For example code in figure 2.12 is given a random name ‘CameraDefaultShader’, which has a correctly proposed name(createShaders) by the model described in the paper.

Graphical user interface, text, application

Description automatically generated

**Figure 2.12 (Bad method name example)**

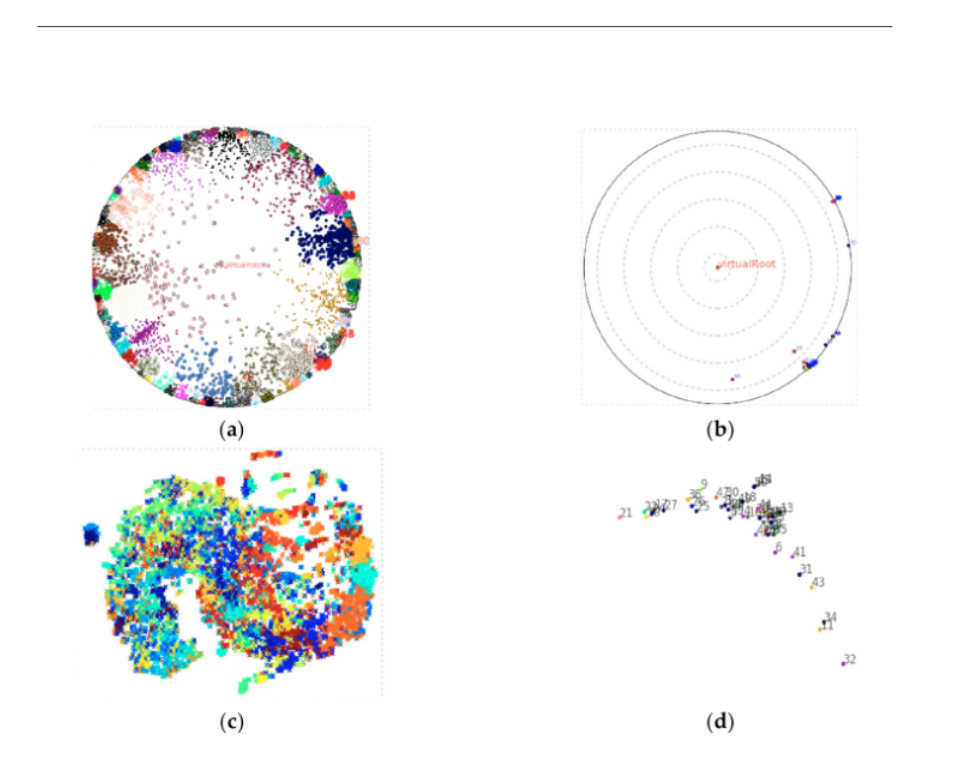
Using the fact that similar functions have similarity in their vector mappings, this paper (DeFreez et al., 2018) discusses the concept of synonym functions and how can we utilize them to find similar function is low level code sources. This is a very common problem any developer faces when he/she tries to find similar function in an open source code base. Because of its size, its generally a very tough task to dwell through the source code searching for any similar implementation. A new technique proposed called FUNC2VEC, which forms a cluster of similar methods/functions. Figure 2.13 visualizes that.

Chart, scatter chart

Description automatically generated

**Figure 2.13 (Func2vec)**

Generally in high dimensional vector space, which usually is an inevitable result of code embeddings, dimensionality reduction techniques like PCA etc. are used. (Lu et al., 2019) proposes a new embedding method called hyperbolic function embedding(HFE). It uses the function call graphs for modeling the function call relations. Main advantage of this novel technique is that it is more compact and has lower dimensions. It generates a function embedding which can be visualized in figure 2.14. (a) demonstrates the function call graphs, (b) shows the weights replaced by Ricci curvature. (c) shows the function embeddings learned using Poincare model and finally (d) shows the actual function embedding. Once the embedding is available it can be used for either state of the art Euclidean space implementations or can be used to train a neural network.



**Figure 2.14 (Hyperbolic function embeddings)**

For python enthusiasts, (Devlin et al., 2017) proposed a new architecture for predicting bugs in the code and suggestions to auto-correct it. It is evident, as mentioned in the paper that such data for training is not available and needs some synthetic preparation of data where bugs are manually introduced and then training is done. As a final outcome it proposes a novel neural architecture which is based on SSC model, which stands for share, specialize and compete. One of their working examples can be visualized in Figure 2.15.

**Graphical user interface, text, application

Description automatically generated**

**Figure 2.15 (Bug Prediction Example)**

(Buch and Andrzejak, 2019) presents an approach on AST-based RNN for clone detection in code. Since the proposed architecture doesn’t involve any embedding technique, we can think of it as out of scope for this thesis study.

Another interesting study on finding the best abstraction of code (Henkel et al., 2018), whether it may be call traces of function calls during a program execution or parsing AST of the program is covered in this study. It also does the benchmarking of different abstractions of code and suggests suitable encoding for learning models.

(Nguyen et al., 2016) discusses an important use case of code embeddings, in which they try API documentations matching for different languages. Such studies can be a very useful tool for developers doing any code migration from one to another language. They have discussed one such example where they migrate the code from java to C#. Their basic idea is to mine the API documentation of both java and C# and use their distributed vector representations to cluster similar API’s to one basket. This is particularly useful in those cases where it’s known that the same feature implementation can only have syntax changes and differently named library usage. One such example of distributed vector representation is shown below in figure 2.15. Simple example of reading a file i.e. ‘BufferReader.readline’ in java and ‘StreamReader.ReadLine’ in C# can be seen as having similar mapping.

Chart, line chart

Description automatically generated

**Figure 2.16 (DeepAPI)**

.

Many times while developing a feature, developer needs to use multiple library methods and API inbuilt into the framework he/she is working with. Most of the times its very difficult to search for API needed for a particular use. This is because of variety of reasons i.e. poor documentation, very less description given of its usage etc. (Gu et al., 2016) proposes a deep learning approach which uses the natural language query and returns the API sequences. These returned sequences can easily be used by the developers for easy understanding and figure out in advance whether its suitable for a particular use. They have named it as DeepAPI, architecture of which can be visualize din figure 2.16. It starts with a code corpus like GitHub and parses it to get annotations an API sequences. Parsed data is used for learning and building a model. An example of extracting data for training can be seen in Figure 2.17.

Diagram

Description automatically generated

**Figure 2.17 (DeepAPI architecture)**

Graphical user interface, text, application, email

Description automatically generated

**Figure 2.18 (DeepBugs)**

(Pradel et al., n.d.) proposes a new framework named DeepBugs for finding common bugs based on incorrect naming of variables or function names. They have trained their model based on huge corpus of JavaScript files and have artificially seeded bugs into the code. They have also used code embeddings for training their binary classifier and have got very promising results. Another approach on using context flow graphs(Figure 2.18) and proposed inst2vec (Ben-Nun et al., 2018) used intermediate representing(IR) of code to train a RNN model. Its implementation naturally makes it language agnostic, where inst2vec is used to generate embedding space of the code.

Diagram

Description automatically generated

**Figure 2.19 (Inst2vec)**

**2.5 Challenges**

Generally with code embeddings, it’s very difficult to measure the quality of the generated vector space. It does look like a black box, which by using in different scenarios can give better/worse results. Embeddings are generated keeping in mind that they will capture the best possible semantics of the code, yet there is no generic way to evaluate them. General embedding techniques in NLP uses one to one mapping of words, whereas contextual mappings using HMM(Hidden Markov Model) can also yield better results. Having very limited amount of data also results in very limited scope of evaluating them. For example in many studies discusses in this literature review have used artificially seeded data, which may help getting better performance but may not capture the actual relationship between among data elements. An extensive study done by (Bakarov, 2018) discusses these problems(and many others) in detail.

**2.6 Summary**

This section discusses through a lot of code embedding techniques and their applications. Its evident that all these inspirational works got started after the infamous release of word2vec paper and similar ideology is applied in open source codes for various use cases, Amount of work which has already been done with code embeddings shows that near future will have a lot of progressive work done in the domain of software engineering. There is a huge amount of code corpus available online, which could pave the way of getting loads of training data. Although we are at the beginning of this new kind of revolution, research in terms of curating meaningful archives of such data could be path breaking. All aspects of software development can benefit from such endeavors and slowly the manual tasks like code submission evaluation, code quality evaluation, static code analysis etc. can be given out to models having understanding of these tasks.

**CHAPTER 3**

**RESEARCH METHODOLOGY**

**3.1 Introduction**

This section discusses the overall implementation approach of predicting class complexity of code. Data selection process, its quality and limitations etc. are discussed in detail in following sub sections. Using tools like astminer, code2vec etc. this study approaches the problem in systematic way, where the expected outcome can belong to any of the 5 different target classes. Several java source codes are transformed into their corresponding vector spaces. These vectors are then used by code2vec model and predicts a complexity class for the given code.

**3.2 Research Approach**

This thesis tries to use code2vec embedding neural framework to generate code embeddings for the selected set of java source code. These code embeddings then will be used to train a code2vec model.

**3.2.1 Data Selection**

This study will use an open source dataset provided by (Sikka, 2019), also called CoRCoD dataset. This is a complete dataset with 932 examples of java code, which belongs to 5 different classes of complexities i.e. and . Since the dataset is already balanced, there won’t be any need for data balancing techniques. Figure 3.1(a) shows the tabulated number of examples available. Since the data is properly labeled with several features of the code samples captured(Figure 3.1(b)), makes it ideal for our study.

Table

Description automatically generatedTable

Description automatically generated

Figure 3.1(a) Figure 3.1(b)

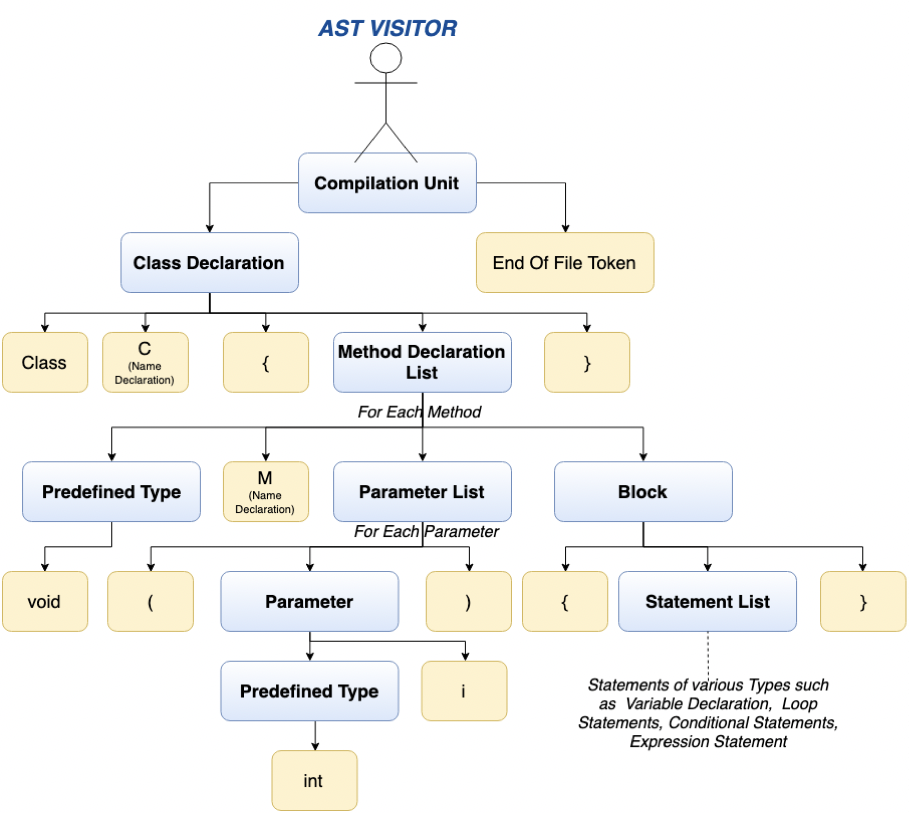
**Figure 3.1 (Data asset)**

It’s clear from the above figures, that although count of data might be less but their overall spread is even. Also the features extracted have many important metrics, which can help determining what features actually contributes more or less for a given time complexity class. Data is also checked to not have any compilation issues and made sure that they all run and finish execution in finite amount of time. There were few more criterion used to make sure all the examples in the dataset executes without any error and has no out of memory issues.

**3.2.2 Data Collection**

This section will describe how the data was collected and processed. For data collection original contributors have used [CodeForces](http://codeforces.com/) API. Codeforces is a platform for online coding competitions. They regularly host a lot of programming contests for coding enthusiasts and presents a variety of problems. Programmers with background of any programming language can take part in these contests and can submit their best solutions. Authors have used their API to collect contest information and used web scraping to download solutions for those problems. These solutions are those which were submitted in java language. They have also made sure that only those solutions are downloads which were either ‘Accepted’ or had ‘Time Limit Exceeded’. Accepted solutions makes sure that they don’t have any compilation issues, whereas ‘TLE’ solutions gave variety into the solutions space and hence to the overall dataset. These downloaded solutions were manually annotated by experienced software developers, just to make sure there are no incorrect labelling of any source code.

Extra features like ‘no of if loops’ etc. were derived using a JDT(Java Development Toolkit) AST plugin, which gives an access to the class ‘ASTVisitor’. Eclipse JDT also provides ‘ASTParser’, using which an AST can be generated for a given java code. Typical AST can look like Figure 3.2



**Figure 3.2 (Using ASTParser)**

Since an AST is just another tree with nodes, ASTVisitor class was used to write a function for node visiting and counting the numbers of different kind of code constructs. E.g. number of nested loops, number of if statements etc. One such example for counting nested loops is shown in Figure 3.3

Graphical user interface, text, application, email

Description automatically generated

**Figure 3.3 (Count nested loops using ASTVisitor)**

**3.2.3 Data Preprocessing**

A picture containing graphical user interface

Description automatically generated

**Figure 3.4 (Flowchart of data processing)**

Above flowchart visualizes the whole data processing steps. Java source code collected from the data collection section is converted into their corresponding AST. An AST parser is then used to extract important features.

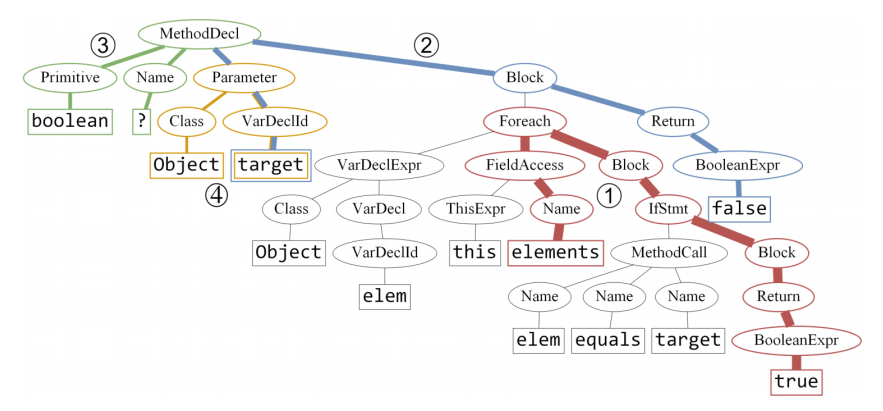
**3.2.4 Data Transformation**

This section discusses about how the source code is transformed into a vector form, so that it can be used for machine learning algorithms. Code2vec (Alon et al., 2019) model is proposed for this transformation of data. Below sub-sections discuss the reasoning behind choosing this model.

**3.2.4.1 Why code2vec**

There are a few code embeddings options i.e. code2seq (Alon et al., 2018a), word2vec (Nguyen et al., 2016) etc. Code2vec is chosen because it is relatively easy to visualize vector representations in terms of path contexts. This neural model parses the AST nodes into path contexts, which is nothing but the triplet of start-node, end-node and the path between them. Given an AST path p, its path context can be defined as , where is the starting node, p is the path (denoted by up or down arrow key) and is the terminal node. For example let’s consider a big program having a statement ‘y == 1’, it’s respective path context can be shown as: (y, (Name | EqualityOperator | Integer), 1)

These path contexts keeps the context of a code statement and the model tries to learn that context with respect to other path context. It results in preserving better semantics of the program overall and model can also choose the ones which are better mapped with the correct expectation. Another advantage of using this model is that it has an intrinsic attention mechanism, which only considers top 3 path contexts based on their respective width in the tree. It is probably safe to assume that useful implementation of the code will have wider node group in its corresponding AST. Code2vec model learns the important implementations parts of AST and gives them a score, also called attention score. In normal machine learning terms it can be understood as weights. Top 3 weights assigned to parts of the AST are picked and only considered for embeddings. This kind of attention mechanism reduces a lot of unnecessary data from the code and helps a lot in visualizing the expected results. Figure 3.5 shows an example of how it works. Path visits denoted in red is the most wider portion of the AST, hence the model gives it the highest weight followed by blue and green.



**Figure 3.5 (Red- 0.23, Blue-0.14, Green- 0.09) (AST with attention scores)**

With bags of path contexts and attention base mechanism, this thesis will try to improve on previous results got in (Sikka et al., 2020).

**3.2.5 Proposed Approach**

This section discusses the approach using code2vec, architecture diagram of which is visualized in Figure 2.24(b). This flowchart depicts all the necessary details of the flow and presents a high level view of the implementation. At first all the java source code files are split into train, test and validation folders. They would need to processed to be able to feed them to code2vec model. Then the model is trained for learning new labels mapped to its respective code embedding. Using Attention layer, most significant context vector is identified and using another dense softmax layer, output is predicted. If needed applications like Embedding Projector, studies done in this paper (Smilkov et al., 2016) etc. can also be utilized.

Architecture diagram and the flowchart of the whole process is presented in Figure 3.7. Figure 3.6 shows the architecture of code2vec.

Diagram

Description automatically generated

**Figure 3.6 (Code2vec architecture)**

**Architecture Diagram For Predicting Time Complexity Using Code2vec**

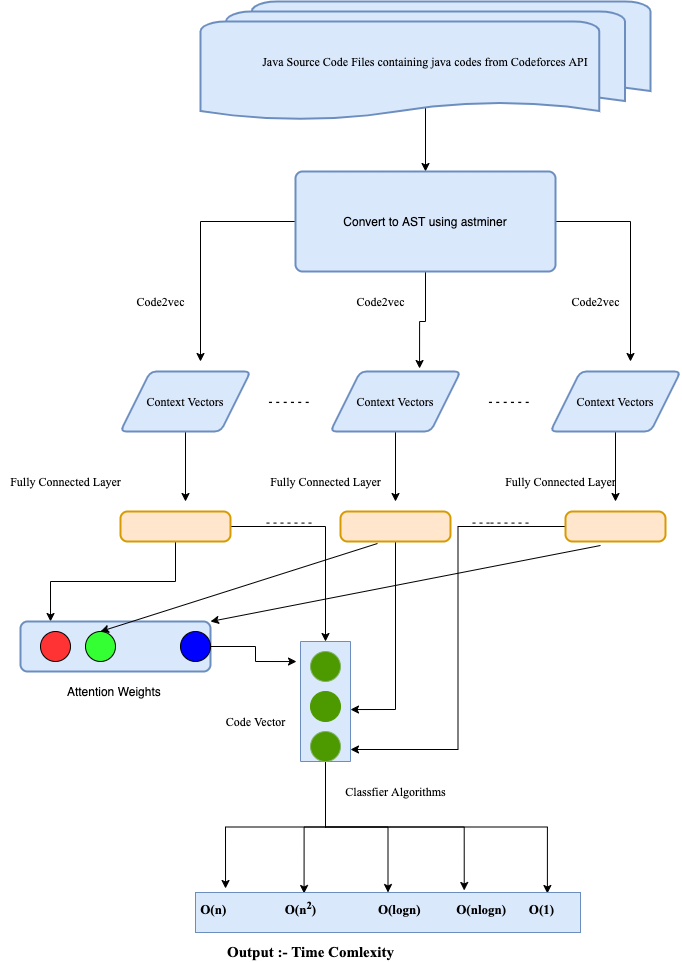


Figure 3.7

**3.2.5.1 Selection Of Classifier Algorithms**

Simple algorithms like SGD(Stochastic Gradient Descent) etc. can be used to minimize loss functions. Training is done based cross entropy loss internally(Rubinstein, 1999). Although the data selected for this thesis is well curated without having any errors but there are only 932 examples in total. Since this is a manual effort, may be automating data creation of such quality could be a nice addition to future work. Another reason for choosing these algorithms is that have very less computation cost compared to any deep learning architecture.

**3.2.6 Evaluation**

Below are the few evaluation metrics this thesis will use. They are:

* **Accuracy**: Since the target variable classes in the dataset are balanced, measuring accuracy on model’s performance can be a good choice. Accuracy can be defined as ratio of sum of true positives, true negatives and sum of true positives, true negatives, false positives, false negatives.

*where* TP is True Positives (precited and actual target value is true)

TN is True Negatives (precited and actual target value is false)

FP is False Positives (precited value is true and actual target value is false)

FN is False Negatives (precited value is false and actual target value is true)

* **Precision**: Since the study would naturally like to measure how precisely is it predicting the correct labels of the given code, precision would be used to measure that. Formula for calculating precision is as below:
* **Recall**: This study would also like to check how many labels of a particular class were actually captured overall. It can be calculated using below formula:
* **F1 Score**: Combining results from both precision and recall, one can get a single score for model’s performance. It can be calculated using formula given below:

**3.3 Tools**

This study would use many open source tools, they are listed down below.

**3.3.1 Python 3.7**

Although python 3.9 is released but since the system’s setup already has packages installed with 3.7 along with all the required dependencies, it makes sense to continue using it. Moreover latest version might have new bugs which are not yet tested with the libraries like code2vec and astminer etc. They are already reported to be working with 3.7. Code2vec has a requirement of using python > 3.6

**3.3.2 TensorFlow 2.0.0**

TensorFlow framework 2.0.0 is also mentioned as the requirement for code2vec and has a specific implementation tested with this version.

**3.3.3 GPU CUDA 10.0**

GPU with CUDA engine 10.0 is also one of the dependency of code2vec. This particular configuration is proven to be very fast and reliable computation of heavy duty mathematical calculations.

**3.3.4 Jupyter Notebook 6.0.1**

Jupyter notebook for analyzing data and development of prediction algorithm.

**3.3.5 Astminer**

Astminer library will be used for generating AST nodes from the sample java source codes.

**3.3.6 Scikit-learn (0.21.2)**

Scikit-learn provides python modules for machine learning and other data mining tasks. It provides many machine learning algorithms like classification, regression etc.

**3.4 Summary**

This study tries to build upon many recent studies done in the domain of software code quality and transformation. Even though memory and space became cheap in last 2 decades, an optimized software is always desirable. Time complexity of a program is just one aspect of measuring code quality and makes sure that along with the desired feature being shipped to clients, it doesn’t over burden the utilities it uses on a daily basis. According to an estimate 111 billions of lines of code is generated every day. Imagine how much extra resources it may take if even a small portion if it grows with an exponential order of growth. It not only increases the cost of maintaining it but also halts the progress of other services depending on it.

After the successful adoption of embedding techniques in NLP, source codes have come far with using the same ideology. There are many libraries and models being developed at present, which tries to create semantically best adaptation of the code to its vector space. This study takes its inspiration from work done in (Sikka et al., 2020), which used graph2vec for creating code embeddings and tries to improve over results obtained by using code2vec. There are a few new ideas implemented on code2vec over graph2vec, which can result in creating better embeddings for machine learning algorithms. Since there are limitations in terms of data, this thesis explores state of the art classifier algorithms but probably would yield different results with more data and deep learning.

**CHAPTER 4**

**ANALYSIS AND DESIGN**

**4.1 Introduction**

This section looks seep into the dataset, it’s distribution and processing steps to make it suitable for model based learning. Following that model’s architecture and hyper-parameters are discussed in detail.

**4.2 Dataset Description**

Dataset(Sikka, 2019) used in this paper is collected from GitHub and is open sourced. It has 933 rows in total corresponding to the same number of java files. Each of this java file is mapped to its corresponding time complexity class. Along with the complexity class, other metadata i.e. number of if statements, nested loop depth, recursion present etc. are also given for each java program. There are 16 such columns capturing different constructs of the given java code. Complete list of features is shown in Figure 4.1

Table

Description automatically generated

**Figure 4.1 (Features From Code Samples)**

**4.3 Dataset Distribution**

This sub-section would try to analyze how the data is distributed and what kind of programming constructs affects its overall time complexity. For example nested loop depths can lead to exponential time complexity, whereas normal if loops can return a linear complexity. Let’s take a look at overall data distribution first.

Figure 4.2 shows the overall data distribution of different kind of class complexities present in the dataset.

Chart, pie chart

Description automatically generated

**Figure 4.2 (Data Distribution Pie Chart)**

Dataset has ~40% of data with time complexity of n, along with approximately 21% java examples with n\_square complexity. Similar percentages of around 16% examples are present for both nlogn and 1 complexity. Logn has the least amount of examples present in the dataset.

**4.4 Exploratory Data Analysis**

This sub-section will try to find correlation between different features and complexity, where complexity is the target class. There could be any number of factors which may affect code’s performance but considering those factors being constant one can design algorithms with careful usage of fast data structures. Figure 4.3 shows linear time complexity gets to its peak at number of if statements are lower. High order time complexities also show similar properties.

**Chart, histogram

Description automatically generated**

**Figure 4.3 (No of If’s versus Complexity)**

**Chart

Description automatically generated**

**Figure 4.4 (hash map versus complexity)**

Figure 4.4 shows that presence of hash map in the code can result into constant time complexity, as expected. Since hash map tend to have constant time look up for values, they are preferred to be used for efficiency.

Figure 4.5 shows the similarity between linear and exponential complexities at lower number of jumps.

Chart, histogram

Description automatically generated

**Figure 4.5 (No of jumps vs Comlexity)**

Similar to Figure 4.5, Figure 4.6 shows similar trend of linear and exponential time complexities and doesn’t have much effect with increasing number of methods.

**Chart, histogram

Description automatically generated**

**Figure 4.6 (No of methods versus Complexity)**

**Chart, bar chart

Description automatically generated**

**Figure 4.7 (Recursion versus Complexity)**

Figure 4.7 shows an interesting trend, where time complexities of code being lower overall, if the code uses recursion. This is a common and extensively used concept in software programming.

Also Figure 4.8 shows expected trend, where increase in number of nested loop results in exponential complexity.

Chart, histogram

Description automatically generated

**Figure 4.8 (Nested loop depth versus Complexity)**

**4.5 Data Preparation**

Since almost all machine learning models including code2vec needs data in some numerical form, this study converts the code into their respective code embedding vectors. Before that the dataset needs to be split into train, test and validation folders. Next sub-section explains that.

**4.5.1 Data Split**

This sub-section explains how the dataset is split into train, test and validation folders. For this purpose a python package named **split-folders** is used. Figure 4.9 shows the flowchart.

A picture containing text, electronics

Description automatically generated

**Figure 4.9 ( data split architecture)**

Figure 4.9 explains how the dataset is split in train, test and validation directory with 70%, 15%, 15% of data captured for each. This package expects data to be separated in target class directories. Separated data is then given as an input to the split-folder command and results in train, test and validation directories with properly balanced data. This would take care of any unbalanced data percentage too.

**4.5.2 Using astminer library**

Another approach of feeding code2vec model with code vectors could be using astminer library by Jetbrains. This open source library provides a way to directly convert code into their corresponding vector representation. Astminer explicitly provides support for code2vec models and created code embeddings which can be directly fed to the model for learning.

There are a few dis-advantages of using this approach. At first it doesn’t give much control over tweaking the dataset for any customized use case. For example this study would need to label the individual java files to its corresponding complexity class. Since the default code2vec model is available to predicting method names, given the java source code, it’s implementation needs to be changed for the particular use case of labelling the dataset. Moreover the internals of astminer would need to change to satisfy the different use case. Hence for the sake of being able to interact more with the dataset and options of tweaking the way code2vec works, manual processing steps are done with code2vec code changed for specific needs of this study.

**4.6 Data Preprocessing**

This section explains the pre-process pipeline and the manual changes done in the original implementation of code2vec model’s processing pipeline. Figure 4.10 shows the pipeline processing.

Graphical user interface

Description automatically generated

**Figure 4.10 (Processing pipeline architecture)**

Figure 4.10 shows the performed steps to process data before it can be fed to the code2vec model. This study tries to modify the extractor utility written in java to return file path along with the path contexts of the given source code. Default implementation of code2vec is optimized for returning method names along with path contexts but this couldn’t be used for this study, since it needs labels which are mapped to respective complexity class labels. This problem can be solved in multiple ways. One approach could be passing label string, while calling the jar. This study takes another approach of labelling each path contexts with its file path, since that’s already is passe as an argument to the jar file. Reusing these file path and using metadata file, these file paths can be easily mapped to their respective target class. Newly modified jar extractor can be found on GitHub (Dhyan, 2021) , which is done as part of this study.

Since the raw files have contents aligned with the goal of identifying correct target class mappings, another shell script is run to get a dictionary output. This dictionary will be used for training the model.

**4.7 Model Architecture**

Figure 4.11 shows the code2vec model’s path attention network, which is implemented in both tenserflow and keras.

Diagram

Description automatically generated

**Figure 4.11 (Attention framework architecture)**

Raw text files i.e. \*.train.raw.txt etc. shown in Figure 4.10 contains the bag of path contexts, which represents the given code snippet. Any two elements of the code are represented as AST(Abstract Syntax Tree) nodes. These two elements along with their path representation forms the path context. For example given a code snippet C, its path context representation can be shows as:

where is the start node, is the terminal node and the path between them is p. These triplets of path contexts can represent a code snippet. Since there are many path contexts in a bag of contexts corresponding to a code snippet, they are converted into triplets of vectors also called context vectors.

In bag of path contexts , a context vector can be represented as:

**4.8 Model Configuration**

In this section, a detailed look into input and output layers of neural network and it’s parameters are discussed. Figure 4.12 shows the data flow within the network.

A screen shot of a computer

Description automatically generated with medium confidence

**Figure 4.12 (Training layer architecture)**

As shown in Figure 4.12, java code snippets goes through preprocessing and its bag of path contexts are stored in raw txt files for train, test and validation respectively. These path contexts are the input to the network to learn from. Concatenation of path and token embeddings form context vectors. They are sent to the dense layer and also mapped with the attention layer to pick which code vector has highest amount of weight, Output is another dense layer with softmax activation. Shaped of tensors or inputs are mentioned in the Figure for better visualization. Weights of token and path embedding’s, dense and attention layers are learnt during the training.

**4.9 Model Training**

For training the model, codee2vec uses cross entropy loss function, which is performed between predicted and true distributions. Let’s assume q as the predicted distribution and p as the true prediction distribution, loss function can be written as:

Where is the actual label. Minimizing this loss function determines the performance of the model. For actual training gradient descent algorithm is used along with the back propagation of error at each iteration. At each step the error is derived against learned parameters and updates the parameter’s value towards the direction which minimizes the loss function.

**4.10 Model Hyperparameters**

This section explains all the important hyperparameters and their values used while training the model.

**4.10.1 MAX\_CONTEXTS**

Maximum number of contexts to be considered for the given code snippet. It’s set to 200 and that’s why the input tensor shape in Figure 4.12 is (None, 200).

**4.10.2 MAX\_TOKEN\_VOCAB\_SIZE**

Maximum size to be kept for source token. Path contexts have source and target token embedded, source tokens collected from them is set to a limit. It’s set to 1301136.

**4.10.3 MAX\_TARGET\_VOCAB\_SIZE**

Same as source token size, target vocabulary size is set. Its set to 261245.

**4.10.4 DEFAULT\_EMBEDDING\_SIZE**

Code embedding size is set to 128. This determines the size of the context vector.

**4.10.5 DROPOUT\_KEEP\_RATE**

Its set to 0.75 but may need better tuning, since the model tends to overfit with this value.

**4.10.6 NUM\_TRAIN\_EPOCHS**

Epoch of 20 is tried to get the best result.

**4.10.7 TRAIN\_BATCH\_SIZE/TEST\_BATCH\_SIZE**

Batch size is set to 1024 for each loop iteration.

**4.11 Summary**

This chapter explains how the dataset is distributed along with the sizes of different target class present. Features present in the dataset shows good amount of correlation between them and the target class complexity. Data preprocessing steps are adapted for this study to be able to learn different label instead of the default behavior of predicting the method names. Architecture of processing pipeline is discussed in detail and manual transformation to the dataset are explained, as well.. Detailed explanation of model’s dataflow and hyperparameters are done at the end of this chapter.

**CHAPTER 5**

**RESULTS AND DISCUSSIONS**

**5.1 Introduction**

This chapter discusses the results and evaluation metrics used for analyzing the model’s performance. Since the dataset is comparatively small, having only 933 examples, current model architecture should improve with more data. Initially the dataset is split into train, validation and test directories with no data overlap between them. 70% of the data is kept for training and then rest 15% each are left for validation and testing predictions. Evaluation metrics i.e. precision, recall, accuracy and F1 score are calculated to validate and verify the results.

**5.2 Output**

Model’s output contains 5 different classifications of target class labels. These target classes are respectively, in which each of them represents the predicted worst case time complexity of the given code. Returned output contains percentage of confidence interval for each target class and based on highest percentage value, the predicted class is selected.

**5.3 Evaluation Metrics And Results**

Evaluation metrics used for measuring model’s performance are accuracy, precision, recall and F1 score. Since this is a classification task, where only one result out of 5 possibilities are chosen for each example, precision and recall are the natural choice. Dataset also seems to have decent balance for all classes, accuracy is measured henceforth. To get a single score, instead of precision and recall, F1 score is also calculated.

Since the model is implemented in both TensorFlow library and TensorFlow keras api, next two sub-sections will discuss their results separately. Since the dataset size is limited, neural network performance was expected to be low. Although this implementation is part of this study but TensorFlow default implementation was done for better comparison.

**5.3.1 Using TensorFlow**

All code changes can be found on GitHub (Dhyan, 2021), results of the model’s performance are discussed in this section. Results are captured after every epoch of frequency 5, 10, 15 and 20. Best results were obtained after 5 epochs, after which there were no significant improvements. 5 epochs results are show below:

* After 1 epochs -- **top10\_acc**: [0.59280169 0.68736768 0.70853917 0.70853917 0.70853917 0.70853917 0.70853917 0.70853917 0.70853917 **0.70853917**], **precision**: 0.592801693719125, **recall**: 0.592801693719125, **F1**: 0.592801693719125
* After 2 epochs -- **top10\_acc**: [0.59844742 0.68666196 0.70853917 0.70853917 0.70853917 0.70853917 0.70853917 0.70853917 0.70853917 **0.70853917**], **precision**: 0.5984474241354976, **recall**: 0.5984474241354976, **F1**: 0.5984474241354976
* After 3 epochs -- **top10\_acc**: [0.59844742 0.68666196 0.70853917 0.70853917 0.70853917 0.70853917 0.70853917 0.70853917 0.70853917 **0.70853917**], **precision**: 0.5984474241354976, **recall**: 0.5984474241354976, **F1**: 0.5984474241354976
* After 4 epochs -- **top10\_acc**: [0.59844742 0.68666196 0.70853917 0.70853917 0.70853917 0.70853917 0.70853917 0.70853917 0.70853917 **0.70853917**], **precision**: 0.5984474241354976, **recall**: 0.5984474241354976, **F1**:0.5984474241354976

Top accuracy recorded is **0.7**, along with **0.59** score of precision, recall and F1 score respectively. Total number of trainable parameters were **16048896.**

**5.3.2 Using Keras**

Keras model also gives similar results, except that the model seems to have overfitting. Model’s performance shows a clear bias towards class complexity. Results of this trial with epochs are shown below.

* After 1 epochs – **loss:** 1.7490**, top\_acc**: [0.993], **precision**: 0.5907, **recall**: 0.5907, **F1**: 0.5907
* After 2 epochs – **loss:** 1.6993**, top\_acc**: [1.0000], **precision**: 0.5984, **recall**: 0.5992, **F1**: 0.5992
* After 3 epochs – **loss:** 1.5329**, top\_acc**: [1.0000], **precision**: 0. 5984, **recall**: 0. 5984, **F1**: 0. 5984
* After 4 epochs – **loss:** 1.5329**, top\_acc**: [1.0000], **precision**: 0. 5984, **recall**: 0. 5984, **F1**: 0. 5984
* After 5 epochs – **loss:** 1.5329**, top\_acc**: [1.0000], **precision**: 0. 5984, **recall**: 0. 5984, **F1**: 0. 5984

Top accuracy recorded is **1**, along with **0.59** score of precision, recall and F1 score respectively. Total number of trainable parameters were **16048896.**

**5.4 Interactive Predictions**

To verify model’s output, saved model can be given a code snippet as an input and output would be the individual percentage values for different classes of complexity. Complexity class having highest value of percentage assigned to it is what model thinks as the best case scenario. Lets see an example of it.

**Example 1**

* Input:- Figure 5.1 shows an example of input code snippet.

Text

Description automatically generated

**Figure 5.1 (Model Input 1)**

* Output: Figure 5.2(using keras model) shows the snapshot of output for input from Figure 5.2

**Text, letter

Description automatically generated**

**Figure 5.2 (Keras Model Output)**

* Figure 5.2 shows the output percentages and predicts with second best value(around 16%) towards . It also displays the attention given to each path context created for the given input code.
* Figure 5.3 shows(using TensorFlow) similar results for the same input as Figure 5.1

**Text, letter

Description automatically generated**

**Figure 5.3 (TensorFlow output)**

**Example 2**

* Input: Figure 5.4 shows another example of an input with complexity .

Graphical user interface, text, application

Description automatically generated

**Figure 5.4 (Input 2)**

* Figure 5.5 shows the output for input shown in Figure 5.4. Figure 5.5 shows the correct prediction of .

**A picture containing text, screenshot, newspaper

Description automatically generated**

**Figure 5.5 (Output 2)**

**5.5 Comparing Models Performance**

This section will try explaining evaluation results of TensorFlow and keras models. Both models show similar values for precision/recall and F1 score. There is a clear difference between desired accuracy and F1 score. Learned model seems to have high bias towards predicting target class. Reason behind this result is because around 40% of total number of examples belong to the complexity class of . This slight imbalance of one particular target class causes model to get biased towards learning the same and hence probably results with predicting linear time complexity on more instances. There are other target classes predicted as well, but they have comparatively lower confidence percentage than linear complexity.

**5.6 Summary**

This chapter explains model’s implementation results using two different approaches i.e. Using TensorFlow and keras. Results of both models are compared and documented in their respective sub-sections. Although dues to data size limitations TensorFlow shows better results but keras implementation shows promising results and is expected to yield better results with more data and better balancing techniques. Model’s output is explored as well, along with interactive results shown for the given input. Few reasons for model’s performance are discussed in brief with explanations of data size limitations and properties of the dataset used.

**CHAPTER 6**

**CONCLUSIONS AND RECOMMENDATIONS**

**6.1 Introduction**

This chapter sums up the new things learnt during the study and concludes on an novel approach of finding code complexity of any given code as an input, using code2vec model. Potential future improvements and recommendations are discussed as well.

**6.2 Discussion and Conclusion**

Motivation of this study is built on top of (Sikka et al., 2020) this study. Dataset used in this paper was picked up, as it was and explored more as part of this thesis. Original paper uses the concept of preparing AST’s of code and then converts them into their vector representations using graph2vec. Several state of the art machine learning algorithms are applied to measure the results.

This paper picks up the same dataset and tries applying code2vec model on top of it. Both TensorFlow and Keras layers are explored to measure the model’s performance. The main difference of current approach is that uses an added attention layer to identify which portion of the code adds up more values while learning a particular class label. Generally whole code doesn’t add up much to the overall class complexity, constructs like loops depth, recursion extra results more into the final outcome of the time complexity. These nuances are learned by the model and hence would be beneficial for nay future improvement work.

Although the original implementation is done with example of predicting method names, part of this study is also to make necessary changes in data preprocessing pipeline to change labels, so that the features can be aggregated and model can learn to identify with new set of labels. This can also provide an example for other use cases i.e. how to change the original use case of code2vec and adapt it to any specific use cases. Since the generation of code vectors is covered up as part of the code2vec implementation, this reduces an extra step of generating them and storing at some location. Code vectors can also be imported into their vectors file, if it’s needed for any SVM classifiers etc.

**6.3 Contribution to knowledge**

Most important contribution of this study is that it shows a way to customize original code2vec implementation to any specific use case. It also shows how the processing pipeline can be changed to add new labels and make changes to let model’s learn them.

Another contribution is that adds a new approach on top of the base paper and explores an opportunity to visualize code embeddings with lenses of identifying what contributes more to a code’s time complexity.

Change done on original code2vec implementation can be found on GitHub (Dhyan, 2021), which also has fixes for the issues mentioned in their original issue tracker. These changes will be published soon on their repository and can be counted as an open source contribution.

**6.4 Future Recommendations**

Few changes done on this repository (Dhyan, 2021) can be open sourced. There are few issues in the original code2vec repository GitHub (code2vec github, 2018), which are mainly related to trying to configure the model on a local setup. Few of them have been encountered in this study as well and GitHub repository of this paper contains fixes for them. They would need to be published following the contribution guidelines.

There is some potential for applying class balancing techniques. These should improve the results and bridge the gap between accuracy and precision/recall.

**REFERENCES**

Aamir, K.M., Abbas, M. and Radenović, S., (2011) A logarithmic time complexity algorithm for pattern searching using productsum property. *Computers and Mathematics with Applications*, 625, pp.2162–2168.

Allamanis, M., Barr, E.T., Bird, C. and Sutton, C., (2015) Suggesting accurate method and class names. In: *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings*. [online] New York, New York, USA: Association for Computing Machinery, Inc, pp.38–49. Available at: http://dl.acm.org/citation.cfm?doid=2786805.2786849 [Accessed 10 Dec. 2020].

Allamanis, M., Peng, H. and Sutton, C., (2016) A Convolutional Attention Network for Extreme Summarization of Source Code BT - International Conference on Machine Learning. *International Conference on Machine Learning*, [online] 48, pp.2091–2100. Available at: http://proceedings.mlr.press/v48/allamanis16.html%0Apapers3://publication/uuid/AD091DDF-C3E0-4CC8-9A24-B7CFFF69EB8F.

Alon, U., Brody, S., Levy, O. and Yahav, E., (2018a) CODE2SEQ: Generating sequences from structured representations of code. *arXiv*, 1, pp.1–22.

Alon, U., Zilberstein, M., Levy, O. and Yahav, E., (2018b) A general path-based representation for predicting program properties. *ACM SIGPLAN Notices*, 534, pp.404–419.

Alon, U., Zilberstein, M., Levy, O. and Yahav, E., (2019) Code2Vec: Learning Distributed Representations of Code. *Proceedings of the ACM on Programming Languages*, 3POPL, pp.1–29.

Anon (2020) *Understanding Time Complexity and its Importance in Technology | by Abdur Rafey Masood | Medium*. [online] Available at: https://medium.com/@abdurrafeymasood/understanding-time-complexity-and-its-importance-in-technology-8279f72d1c6a [Accessed 3 Dec. 2020].

Azcona, D., Hsiao, I.H., Arora, P. and Smeaton, A., (2019) User2Code2vec: Embeddings for profiling students based on distributional representations of source code. In: *ACM International Conference Proceeding Series*. [online] New York, NY, USA: Association for Computing Machinery, pp.86–95. Available at: https://dl.acm.org/doi/10.1145/3303772.3303813 [Accessed 10 Dec. 2020].

Bakarov, A., (2018) A Survey of Word Embeddings Evaluation Methods. *arXiv*. [online] Available at: http://arxiv.org/abs/1801.09536 [Accessed 11 Dec. 2020].

Ben-Nun, T., Jakobovits, A.S. and Hoefler, T., (2018) Neural Code Comprehension: A Learnable Representation of Code Semantics. *Advances in Neural Information Processing Systems*, [online] 2018-December, pp.3585–3597. Available at: http://arxiv.org/abs/1806.07336 [Accessed 11 Dec. 2020].

Bentley, J.L., Haken, D. and Saxe, J.B., (1980) A general method for solving divide-and-conquer recurrences. *ACM SIGACT News*, [online] 123, pp.36–44. Available at: https://dl.acm.org/doi/10.1145/1008861.1008865 [Accessed 27 Sep. 2020].

Beyer, K., Goldstein, J., Ramakrishnan, R. and Shaft, U., (1998) When is “nearest neighbor” meaningful? In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. [online] Springer Verlag, pp.217–235. Available at: https://link.springer.com/chapter/10.1007/3-540-49257-7\_15 [Accessed 10 Dec. 2020].

Buch, L. and Andrzejak, A., (2019) Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection. In: *SANER 2019 - Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering*. Institute of Electrical and Electronics Engineers Inc., pp.95–104.

Chen, Z. and Monperrus, M., (2018) The Remarkable Role of Similarity in Redundancy-based Program Repair. *arXiv*. [online] Available at: http://arxiv.org/abs/1811.05703 [Accessed 10 Dec. 2020].

Chen, Z. and Monperrus, M., (2019) A Literature Study of Embeddings on Source Code. [online] pp.1–8. Available at: http://arxiv.org/abs/1904.03061.

code2vec github, (2018) *GitHub - tech-srl/code2vec: TensorFlow code for the neural network presented in the paper: ‘code2vec: Learning Distributed Representations of Code’*. [online] Available at: https://github.com/tech-srl/code2vec [Accessed 27 Sep. 2020].

Cosentino, V., Luis, J., Izquierdo, C. and Cabot, J., (2016) Findings from GitHub: Methods, datasets and limitations. *Proceedings - 13th Working Conference on Mining Software Repositories, MSR 2016*, pp.137–141.

DeFreez, D., Thakur, A. V. and Rubio-Gonzalez, C., (2018) Path-based function embedding and its application to error-handling specification mining. In: *ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. [online] New York, New York, USA: Association for Computing Machinery, Inc, pp.423–433. Available at: http://dl.acm.org/citation.cfm?doid=3236024.3236059 [Accessed 10 Dec. 2020].

Devlin, J., Uesato, J., Singh, R. and Kohli, P., (2017) Semantic Code Repair using Neuro-Symbolic Transformation Networks. *arXiv*. [online] Available at: http://arxiv.org/abs/1710.11054 [Accessed 10 Dec. 2020].

Dhyan, (2021) *GitHub - dhyan/code2vec\_code\_complexity*. [online] Available at: https://github.com/dhyan/code2vec\_code\_complexity [Accessed 9 Feb. 2021].

Dobravec, T., (2018) Estimating the time complexity of the algorithms by counting the Java bytecode instructions. In: *2017 IEEE 14th International Scientific Conference on Informatics, INFORMATICS 2017 - Proceedings*. Institute of Electrical and Electronics Engineers Inc., pp.74–79.

Gauvrit, N., Singmann, H., Soler-Toscano, F. and Zenil, H., (2016) Algorithmic complexity for psychology: a user-friendly implementation of the coding theorem method. *Behavior Research Methods*, [online] 481, pp.314–329. Available at: http://algorithmicnature.org [Accessed 6 Dec. 2020].

Gomes, V.M., Paiva, J.R.B., Reis, M.R.C., Wainer, G.A. and Calixto, W.P., (2019) Mechanism for Measuring System Complexity Applying Sensitivity Analysis. *Complexity*, 2019.

Gu, X., Zhang, H., Zhang, D. and Kim, S., (2016) Deep API Learning. *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, [online] 13-18-November-2016, pp.631–642. Available at: http://arxiv.org/abs/1605.08535 [Accessed 11 Dec. 2020].

Harer, J.A., Kim, L.Y., Russell, R.L., Ozdemir, O., Kosta, L.R., Rangamani, A., Hamilton, L.H., Centeno, G.I., Key, J.R., Ellingwood, P.M., Antelman, E., Mackay, A., McConley, M.W., Opper, J.M., Chin, P. and Lazovich, T., (2018) Automated software vulnerability detection with machine learning. *arXiv*. [online] Available at: http://arxiv.org/abs/1803.04497 [Accessed 10 Dec. 2020].

Henkel, J., Lahiri, S.K., Liblit, B. and Reps, T., (2018) Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces. *ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, [online] 18, pp.163–174. Available at: http://arxiv.org/abs/1803.06686 [Accessed 11 Dec. 2020].

Hindle, A., Barr, E.T., Su, Z., Gabel, M. and Devanbu, P., (2012) On the naturalness of software. In: *Proceedings - International Conference on Software Engineering*. pp.837–847.

Jan Šimbera, (2019) *pysynth · PyPI*. [online] github. Available at: https://pypi.org/project/pysynth/ [Accessed 27 Sep. 2020].

Jayram, T.S., Kumar, R. and Sivakumar, D., (2003) Two applications of information complexity. In: *Proceedings of the thirty-fifth ACM symposium on Theory of computing - STOC ’03*. [online] New York, New York, USA: Association for Computing Machinery (ACM), p.673. Available at: http://portal.acm.org/citation.cfm?doid=780542.780640 [Accessed 6 Dec. 2020].

Klir, G.J. and Simon, H.A., (1991) The Architecture of Complexity. In: *Facets of Systems Science*. [online] Springer US, pp.457–476. Available at: https://link.springer.com/chapter/10.1007/978-1-4899-0718-9\_31 [Accessed 6 Dec. 2020].

Kumarsharma, D., Vohra, S., Gupta, T. and Goyal, V., (2018) Predicting the Algorithmic Time Complexity of Single Parametric Algorithms Using Multiclass Classification with Gradient Boosted Trees. *2018 11th International Conference on Contemporary Computing, IC3 2018*, pp.2–4.

Lehre, P.K. and Özcan, E., (2013) A runtime analysis of simple hyper-heuristics: To mix or not to mix operators. *FOGA 2013 - Proceedings of the 12th ACM Workshop on Foundations of Genetic Algorithms*, pp.97–104.

Li, J., He, P., Zhu, J. and Lyu, M.R., (2017) Software defect prediction via convolutional neural network. In: *Proceedings - 2017 IEEE International Conference on Software Quality, Reliability and Security, QRS 2017*. Institute of Electrical and Electronics Engineers Inc., pp.318–328.

Lloyd, S., (2001) Measures of Complexity: A Nonexhaustive List. *IEEE Control Systems*, 214, pp.7–8.

Lu, M., Liu, Y., Li, H., Tan, D., He, X., Bi, W. and Li, W., (2019) Hyperbolic function embedding: Learning hierarchical representation for functions of source code in hyperbolic space. *Symmetry*, 112.

M, H. and M.N, S., (2015) A Review on Evaluation Metrics for Data Classification Evaluations. *International Journal of Data Mining & Knowledge Management Process*, 52, pp.01–11.

Martin, F., (2018) *Refactoring*. [online] Available at: https://martinfowler.com/books/refactoring.html [Accessed 10 Dec. 2020].

McCabe, T.J., (2020) *A complexity measure | Proceedings of the 2nd international conference on Software engineering*. [online] ICSE ’76: Proceedings of the 2nd international conference on Software engineering. Available at: https://dl.acm.org/doi/10.5555/800253.807712 [Accessed 27 Sep. 2020].

Mikolov, T., Chen, K., Corrado, G. and Dean, J., (2013) Efficient estimation of word representations in vector space. In: *1st International Conference on Learning Representations, ICLR 2013 - Workshop Track Proceedings*. [online] International Conference on Learning Representations, ICLR. Available at: http://ronan.collobert.com/senna/ [Accessed 8 Dec. 2020].

Nagaraj, N., Balasubramanian, K. and Dey, S., (2013) A new complexity measure for time series analysis and classification. *European Physical Journal: Special Topics*, 2223–4, pp.847–860.

Narayanan, A., Chandramohan, M., Venkatesan, R., Chen, L., Liu, Y. and Jaiswal, S., (2017) graph2vec: Learning Distributed Representations of Graphs. *28th Modern Artificial Intelligence and Cognitive Science Conference, MAICS 2017*, [online] pp.189–190. Available at: http://arxiv.org/abs/1707.05005.

Nguyen, T.D., Nguyen, A.T. and Nguyen, T.N., (2016) Mapping API elements for code migration with vector representations. In: *Proceedings - International Conference on Software Engineering*. [online] New York, NY, USA: IEEE Computer Society, pp.756–758. Available at: https://dl.acm.org/doi/10.1145/2889160.2892661 [Accessed 11 Dec. 2020].

Oliveto, P.S., He, J. and Yao, X., (2007) Time complexity of evolutionary algorithms for combinatorial optimization: A decade of results. *International Journal of Automation and Computing*, 43, pp.281–293.

Padua, D., Ghoting, A., Gunnels, J.A., Squillante, M.S., Meseguer, J., Cownie, J.H., Roweth, D., Adve, S. V., Boehm, H.J., McKee, S.A., Wisniewski, R.W., Karypis, G., Malony, A.D., Gottlieb, S., Riesen, R., Maccabe, A.B., Bilardi, G., Pietracaprina, A., Kejariwal, A., Nicolau, A., Lengauer, C., Gustafson, J.L., Gropp, W., Prost, J.-P., Padua, D., Lowney, G., Amestoy, P., Buttari, A., Duff, I., Guermouche, A., L’Excellent, J.-Y., Uçar, B., Halstead, R.H., Nemirovsky, M., Amestoy, P., Buttari, A., Duff, I., Guermouche, A., L’Excellent, J.-Y., Uçar, B. and Pakin, S., (2011) Moore’s Law. In: *Encyclopedia of Parallel Computing*. [online] Boston, MA: Springer US, pp.1177–1184. Available at: http://link.springer.com/10.1007/978-0-387-09766-4\_81 [Accessed 6 Dec. 2020].

Papadimitriou, C., (2014) Algorithms, complexity, and the sciences. *Proceedings of the National Academy of Sciences of the United States of America*, [online] 11145, pp.15881–15887. Available at: www.pnas.org/cgi/doi/10.1073/pnas.1416954111 [Accessed 6 Dec. 2020].

Pradel, M., Darmstadt, T. and Koushik Sen, G., (n.d.) *DeepBugs: A Learning Approach to Name-based Bug Detection*. [online] Available at: https://lgtm.com/ [Accessed 11 Dec. 2020].

Rubinstein, R., (1999) The Cross-Entropy Method for Combinatorial and Continuous Optimization. *Methodology And Computing In Applied Probability*, [online] 12, pp.127–190. Available at: https://link.springer.com/article/10.1023/A:1010091220143 [Accessed 13 Feb. 2021].

Sikka, J., Satya, K., Kumar, Y., Uppal, S., Shah, R.R. and Zimmermann, R., (2020) Learning Based Methods for Code Runtime Complexity Prediction. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 12035 LNCS, pp.313–325.

Sikka, J.K.A., (2019) *GitHub - midas-research/corcod-dataset: CoRCoD: Code Runtime Complexity Dataset*. [online] MIDAS Research Laboratory, IIIT-Delhi. Available at: https://github.com/midas-research/corcod-dataset [Accessed 27 Sep. 2020].

SlattonK. Clint, B.L.E., (2005) *Source Code Repository - an overview | ScienceDirect Topics*. [online] Available at: https://www.sciencedirect.com/topics/computer-science/source-code-repository [Accessed 6 Dec. 2020].

Smilkov, D., Thorat, N., Nicholson, C., Reif, E., Viégas, F.B. and Wattenberg, M., (2016) Embedding Projector: Interactive Visualization and Interpretation of Embeddings. [online] Available at: http://arxiv.org/abs/1611.05469 [Accessed 12 Dec. 2020].

Soler-Toscano, F. and Zenil, H., (2017) A Computable Measure of Algorithmic Probability by Finite Approximations with an Application to Integer Sequences. *Complexity*, 2017.

Srikant, S. and Aggarwal, V., (2014) A system to grade computer programming skills using machine learning. In: *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. [online] New York, New York, USA: Association for Computing Machinery, pp.1887–1896. Available at: http://dl.acm.org/citation.cfm?doid=2623330.2623377 [Accessed 6 Dec. 2020].

Stanford encyclopedia, (2020) *Computational Complexity Theory (Stanford Encyclopedia of Philosophy)*. [online] Available at: https://plato.stanford.edu/entries/computational-complexity/ [Accessed 3 Dec. 2020].

SUMIT VOHRA vipul goyal, (2017) *Predicitng\_Time\_Complexity/Output\_Data at master · saisumit/Predicitng\_Time\_Complexity · GitHub*. [online] github. Available at: https://github.com/saisumit/Predicitng\_Time\_Complexity/tree/master/Output\_Data [Accessed 27 Sep. 2020].

Sun, Z., Zhu, Q., Mou, L., Xiong, Y., Li, G. and Zhang, L., (2019) A Grammar-Based Structural CNN Decoder for Code Generation. *Proceedings of the AAAI Conference on Artificial Intelligence*, [online] 3301, pp.7055–7062. Available at: https://github.com/zysszy/GrammarCNN [Accessed 27 Sep. 2020].

White, M., Tufano, M., Martinez, M., Monperrus, M. and Poshyvanyk, D., (2019) Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities. *SANER 2019 - Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering*, pp.479–490.

Wirth, N., (1995) *Wirth (1995) IEEE Computer.pdf*. *Cybersquare*, .

Yao, Z., Sun, Y., Ding, W., Rao, N. and Xiong, H., (2018) Dynamic word embeddings for evolving semantic discovery. *WSDM 2018 - Proceedings of the 11th ACM International Conference on Web Search and Data Mining*, 2018-Febua, pp.673–681.

Yonai, H., Hayase, Y. and Kitagawa, H., (2019) Mercem: Method Name Recommendation Based on Call Graph Embedding. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, 2019-Decem, pp.134–141.

**APPENDIX A**

**RESEARCH PROPOSAL**

Predicting code time complexity using code2vec and machine learning

Narendra Kumar

Research Proposal

September 2020

**Abstract**

Estimating time complexity of any program is a difficult task. It needs an experienced programmer to be able to correctly judge the asymptotic upper bound of any code. Determining this is a very subtle process and needs comprehensible knowledge of different data structures and the way they are implemented in a given programming language. Turing Halting problem states that its practically impossible to estimate running time complexity. Hence an approximated analysis of code gives programmers an overview of how their code performs on any given machine of any configuration. Such intrinsic nature of this problem makes it suitable for machine learning implementation. This paper used CoRCod dataset, discussed in this paper(Sikka et al., 2020) is used to train a multiclass classifier model, which is expected to learn from the data and predict time complexity of any unseen program. Solutions like these can be very useful for any automatic grading of online programming contests, static analysis of code in IDE and others. This research studies only the upper bound of any program, also known as worst case time complexity and denoted by O(Big Oh) notation.

**1.** **Background**

Time complexity of an algorithm is the amount of runtime, an algorithm needs to finish execution without any error. Formally it is measured as an order of growth of a function of the given input size. This research studies worst case time complexity, which is also known as the upper bound time complexity and are represented by O(Big Oh) notation. For example shows the asymptotic upper bound of an algorithm, where n is the input size.

Measuring time complexity of an algorithm is an arduous task, since it requires significant amount of effort to figure them out. There are several manual methods to compute them, but either they are very problem specific or are error prone. For example Master’s theorem(Bentley et al., 1980) is limited to the divide and conquer algorithms. Cyclomatic complexity(McCabe, 2020) uses call graphs to determine the logic strength of an algorithm but doesn’t account for how many times a specific path in the graph is called. This leaves a question with its robustness while analysing different kind of algorithms.

Turing halting problem proves that it is mathematically impossible to estimate code complexity. Rice’s theorem and similar studies have also shown that there can’t be any general mathematical equation, which can be formulated for any code with polynomial order complexity. These limitations make this problem ideal for machine learning. For data requirements there are many online sources like Public Git Archive, stack overflow, which hosts a large amount of dataset to be explored.

Research done by(Sikka et al., 2020) does an excellent work in terms of creating a dataset, which maps a program with its respective complexity. It achieves motivating results using traditional machine learning models, along with using graph2vec(Narayanan et al., 2017) for code embeddings. It also proposes results for all baseline classifier models, along with the study of features which has high effect on the computed complexity class. Data generated for learning took a lot of manual effort and hence the total number of samples generated are limited. Although the dataset is balanced and has no bias but it’s limited number makes it not suitable for exploring deep learning models.

This paper tries to substantially explore synthetic data generation techniques to add more samples, without reducing the data quality. It will also try to explore different techniques to find distributed code representations. Distributed code representations used AST of the code to generate graphs and uses the same to generate a vector for training a machine learning model. Overall this study tries to contribute:

* Creating more data samples while maintaining data quality. It should ensure that the original distribution of the data is not affected much.
* Use code2vec(Alon et al., 2019) neural network for generating code embeddings and compare its performance with graph2vec.
* Compare baseline tradition classifier models performance with the models which were trained on original CoRCoD dataset(Sikka, 2019).

**2.** **Related Work**

Predicting time complexity of the code is relatively new research area, which is gaining traction because of it’s obvious potential use in different areas. Initially most of the research in this area was concentrated on analysis of sorting and searching algorithms, recently work on generating code embeddings or graph structures opens up new ways to explore the same. Work done on sorting algorithms like quicksort suggests an improvement over the traditional algorithm. In comparative analysis of traditional and proposed searching techniques, advantages/disadvantages are discussed in detail. It also shows considerable advantages over traditional algorithms like linear/binary search. New algorithms like Bi-linear search, nearest neighbour search, multiple solution vector approach supports deeper contrasts between them. Although being very interesting and informative, their limitations are bounded within searching and sorting techniques.

Some work is also done towards computing the time complexity on any algorithm in general. In paper by Tomaz Dobravec(Dobravec, 2018), researcher has proposed a new framework ALGator. This framework tries to predict the complexity by counting the respective bytecodes of the java program. Although it’s a novel approach, its dependence over the algorithm constants can’t be generalised. Count formula proposed in this paper needs constants which are algorithm specific, along with the fact that the whole process is time consuming.

Another approach(SUMIT VOHRA vipul goyal, 2017) uses control flow graphs for estimating time complexity. Proposed algorithm and it’s design are tedious, along with the fact that it can only be applied to smaller segments of the program.

In paper(Kumarsharma et al., 2018), authors have proposed a novel approach of using supervised learning, using gradient boost trees for predicting time complexities. It focusses on using gradient boost tress for complexities which occur frequently in the dataset. For generating the dataset, this research runs multiple C++ programs on computer systems of different configurations. Upon running them multiple times in loops, data is generated for modelling.

In deep learning community, there has been a lot of recent studies on software codes. These researches contribute on predicting the structure of the code or predicting some attributes of the code. Most of the code editors used by the programmers use features like auto code completion, syntax suggestions etc benefits a lot from these kind of studies. Some studies have also been used to predict if the code is syntactically correct or properly structured. These semantic checks are studied in this paper(Sun et al., 2019). This research used CNN, which uses the Abstract Syntax Tree(AST) of the program and then predicts the expected grammar rules.

This research(Allamanis et al., 2016) studies on predicting the method or variable’s name for the given code. Attention technique in CNN is used for modelling the algorithm. Similar to this study, another similar approach is studied in this paper(Alon et al., 2018b). Difference between these two approaches is that the latter used the AST(Abstract Syntax Tree) for determining the probable structure of the code. That way it learns the code semantics better, since it doesn’t treat the code as just a sequence of characters, which might not hold them better. Use of AST serves as context for generating code embeddings and classification models are trained on top of them.

Another study done by (Yonai et al., 2019) uses call graphs and graph embedding techniques to build method call graph or method embeddings. This paper proposes a new approach to recommend method names based on the target method called. This can have a greater impact over writing comprehensible code, where method names are more sensical and such approaches can be applied to doing static code analysis. Bad method names are very common and it can help to have a better check on that.

Many other researches(Li et al., 2017) working on the combination of AST and CNN are used for defect prediction in the code.

Most of the studies done so far used some form of code representations and use them to build context which holds both semantic and syntactic relationship. In fields like NLP(Natural Language Processing), techniques like word2vec gave a major breakthrough in terms of improvement in model’s accuracy, similar idea can also be tried with code embeddings. Very recent research on using it for time complexity(Sikka et al., 2020) seems very promising and uses graph2vec for generating a vector. Code AST is converted into a vector, which is then used to train a classifier model. This research provides the main idea behind the current study.

Another embedding approach called code2vec(Alon et al., 2019) can be used to generate code embeddings, on top of which models can be trained.

**3.** **Aim and Objectives**

The main aim of this research is to propose a model which predicts time complexity of the software code based on it’s intrinsic features. These features can be number of if statements, number of for loops, nested loop depth among other common code constructs. The goal of this research is to come up with a better performing model using code2vec(code2vec github, 2018), which is a neural network for learning distributed representations of code.

The research objectives are formulated based on the aim of study which are as follows:

* To analyse the important features contributing to code’s time complexity.
* To suggest a suitable data generation method from existing samples of data.
* To compare between graph2vec and code2vec neural network’s performance on the data asset.
* To evaluate state of the art multi-class classification algorithms and compare their performance.

**4.** **Significance and Scope**

Even though the cost of computing resources have become relatively cheaper than few decades ago, one can never deny over the advantages of software running at its optimum level. Optimized software is easy to scale and overall maintainability of the code improves manifold. One of the metrics to evaluate a software code is to measure its time complexity or order of growth with respect to its input size. Codes having lower order of growth can be considered well implemented. This study tries to implement a model, which can predict the time complexity of the code, having code itself as an input. Most of the IDE’s(Interactive development environment) can use such models as a plugin to automatically inform programmers about the complexity. This can be a static analyser tool for quick feedback. Almost the continuous integration tools like Circle CI, Jenkins etc can also leverage this tool to run builds on their servers and post results for the same to the author. Many online contests can also take advantage of it to automatically generate results, since proposing the best solution is almost always one the criterion for the test cases attached to a problem.

Scope of this paper is limited to predicting time complexity of the code. Code running time also depends on the configuration of system it is running on. Systems having higher clock speed, better RAM are expected to execute the code instructions faster. Hence same algorithm can show different running times in different environment. Kernel processes can also change the runtime based on how prioritised the current process is. Code instructions having high priority instructions will be grabbed first than those having low priority. Basic assumptions like constant time lookup for a variable, insertion and deletion are inherently present in the current study. Other concepts like space complexity are not considered, instead main focus is on studying the order of growth based on the input size.

**7.** **Research Methodology**

Latest research(Sikka et al., 2020) done on this subject does the static analysis of the given code and using a java parser it computes the code structure into 14 different features. These features are mapped to a target feature named complexity. Complexity is our target class which was formulated manually by set of experienced programming experts. Several state of the art multiclass classifiers will be trained on these features and their performances will be compared. A novel approach of generating code embeddings are done using the AST(Abstract Syntax Tree) of the program and converted to a vector. Transformation from these graph to vector is done using graph2vec. This research tries to use code2vec, which converts code into its corresponding vector. Generated vector can be used to train a model.

This section is divided into multiple sub-sections. Sub-section 7.1 discusses about the dataset and its features in detail. 7.2 discusses about using code2vec and using the transformed data for modelling. 7.3 discusses about the probable models this study can try to train and then the last section will be about the possible evaluation metrics.

**7.1 Dataset**

Given dataset is constructed using the source code obtained by using the Codeforces. Codeforces is an online platform for hosting coding competition. They provide a wide variety of problem sets and requires coders to finish it in a suitable time slot. Solutions to these problems are expected to run with an expected time complexity with usage of proper data structures.

Present dataset is the collection of 933 java source codes for which the solutions are available on the site. Using the Codeforces API problem and the contest details are retrieved. Solutions for the given problem sets are obtained by scraping the site. To validate the solutions, only those solutions are scraped which meets the expected time criterion to finish execution and has the status of Accepted or time limit exceeded. Time limit exceeded solutions are also chosen to ensure that there are multiple solutions available for the same problem and can be used for comparison. Also solutions are filtered out based on criteria of having at least four test cases passing and is in accepted state. For annotating these code solutions, five experts having computer science or related background were used to identify the correct time complexity. Only order of time complexity is measured, which is a function of the given input size.

Computed dataset features and their respective data-types are as below:

* Number of methods (Integer)
* Number of switches (Integer)
* Priority queue present (Integer)
* Hash map present (Boolean)
* Nested loop depth (Integer)
* Number of variables (Integer)
* Number of statements (Integer)
* Number of breaks (Integer)
* Number of loops (Integer)
* Sort present (Boolean)
* Hash set present (Boolean)
* Recursion present (Boolean)
* Number of ifs (Integer)
* Number of jumps (Integer)
* Complexity (Target Class with five different classes)
* File name (source code)

Target class(Complexity) description with respective number of samples present in the dataset are as below:

* O() (385 samples)
* O() (200 samples)
* O() (150 samples)
* O() (143 samples)
* O() (55 samples)

Few assumptions were made while annotating the source code. They are:

* It has been assumed that the sorting algorithms in java collections have an order of complexity O() .
* Insert and delete operation in HashSet and HashMap have a constant time complexity.
* Tree Map and Tree Set have O() complexity for an insertion or deletion.

**7.2 Data processing**

Given dataset is balanced and may not need any data balancing techniques like sampling etc. Since the number of samples are very limited, this study is going to focus on generating synthetic samples, while preserving the data quality. Some python packages like pysynth(Jan Šimbera, 2019) can help on this regard. Also over-sampling techniques like SMOTEs(Synthetic Minority Oversampling Technique) can help in generating more samples, while maintaining the overall distribution of the given data. Original data annotation is based on a lot of manual effort and that’s one area which needs to be looked into.

**7.3 Models**

Classification model will be trained based on two approaches. First approach is to use the existing features, as is to train multiclass classifier model and second approach will use code embeddings.

**7.3.1 Using traditional ML algorithms**

Deep learning models tend to improve with the increase in data quality and quantity. With the amount of data this study has, classic machine learning algorithms can give better results than the former models. Scope of this research is to try a few traditional machine learning algorithms and compare their results.

**7.3.2 Using code embeddings**

This paper is using code2vec library for converting code to its corresponding vector. Since all the algorithms expect some kind of mathematical data structure to work with, converting the source codes to their respective vectors give us a chance to study them and verify their results. Code2vec is a neural network which learns distributed representation of the code.

Main advantage of using code2vec over graoh2vec is that it has an attention mechanism which is capable of recognising the tree(AST) path which contributes more to the semantics of the given source code.. This recognition is loosely based on the part of the generated tree, which has maximum depth. This study may leverage the use of transfer learning, since the code2vec do have some pretrained models to use for.

**7.4 Evaluation Metrics**

Based on this study(M and M.N, 2015) and available dataset, there can be different evaluation rubrics applied to classification problems. It depends on whether they are multi-class or multi-label problems. Current research focusses on solving a multi-class problem, hence calculating accuracy, precision, recall, F-score, confusion matrix etc will be used as a launching pad. Reason behind choosing these are because they are easy to compute and are human readable. To check whether the model learning is optimized, AUC will be used. Since this paper doesn’t contribute towards finding the best performing model, usage of discriminatory techniques are not included.

**8.** **Required Resources**

Resources required for this study are as below:

* Python 3.7, TensorFlow, Scikit-learn
* Code2vec
* GPU access with CUDA or Nimble box (For training code embeddings)

**9.** **Research Plan**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Oct 12** | **Oct 26** | **Nov 9** | **Nov 23** | **Nov 30** | **Dec 14** | **Dec 28** | **Jan 11** | **Jan 25** | **Feb 1** |
| **Training and evaluation of classifier models** |  |  |  |  |  |  |  |  |  |  |
| **Application of data generation techniques** |  |  |  |  |  |  |  |  |  |  |
| **Application of code embeddings using code2vec** |  |  |  |  |  |  |  |  |  |  |
| **Summarize work done and start preparing mid thesis report** |  |  |  |  |  |  |  |  |  |  |
| **Mid thesis submission** |  |  |  |  |  |  |  |  |  |  |
| **Revisit data generation step and retrain models** |  |  |  |  |  |  |  |  |  |  |
| **Retrain code embeddings and model tuning** |  |  |  |  |  |  |  |  |  |  |
| **Analyse working/promising outcomes and improve on that** |  |  |  |  |  |  |  |  |  |  |
| **Final thesis report preparation.** |  |  |  |  |  |  |  |  |  |  |

**References**

Aamir, K.M., Abbas, M. and Radenović, S., (2011) A logarithmic time complexity algorithm for pattern searching using productsum property. *Computers and Mathematics with Applications*, 625, pp.2162–2168.

Allamanis, M., Barr, E.T., Bird, C. and Sutton, C., (2015) Suggesting accurate method and class names. In: *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings*. [online] New York, New York, USA: Association for Computing Machinery, Inc, pp.38–49. Available at: http://dl.acm.org/citation.cfm?doid=2786805.2786849 [Accessed 10 Dec. 2020].

Allamanis, M., Peng, H. and Sutton, C., (2016) A Convolutional Attention Network for Extreme Summarization of Source Code BT - International Conference on Machine Learning. *International Conference on Machine Learning*, [online] 48, pp.2091–2100. Available at: http://proceedings.mlr.press/v48/allamanis16.html%0Apapers3://publication/uuid/AD091DDF-C3E0-4CC8-9A24-B7CFFF69EB8F.

Alon, U., Brody, S., Levy, O. and Yahav, E., (2018a) CODE2SEQ: Generating sequences from structured representations of code. *arXiv*, 1, pp.1–22.

Alon, U., Zilberstein, M., Levy, O. and Yahav, E., (2018b) A general path-based representation for predicting program properties. *ACM SIGPLAN Notices*, 534, pp.404–419.

Alon, U., Zilberstein, M., Levy, O. and Yahav, E., (2019) Code2Vec: Learning Distributed Representations of Code. *Proceedings of the ACM on Programming Languages*, 3POPL, pp.1–29.

Anon (2020) *Understanding Time Complexity and its Importance in Technology | by Abdur Rafey Masood | Medium*. [online] Available at: https://medium.com/@abdurrafeymasood/understanding-time-complexity-and-its-importance-in-technology-8279f72d1c6a [Accessed 3 Dec. 2020].

Azcona, D., Hsiao, I.H., Arora, P. and Smeaton, A., (2019) User2Code2vec: Embeddings for profiling students based on distributional representations of source code. In: *ACM International Conference Proceeding Series*. [online] New York, NY, USA: Association for Computing Machinery, pp.86–95. Available at: https://dl.acm.org/doi/10.1145/3303772.3303813 [Accessed 10 Dec. 2020].

Bakarov, A., (2018) A Survey of Word Embeddings Evaluation Methods. *arXiv*. [online] Available at: http://arxiv.org/abs/1801.09536 [Accessed 11 Dec. 2020].

Ben-Nun, T., Jakobovits, A.S. and Hoefler, T., (2018) Neural Code Comprehension: A Learnable Representation of Code Semantics. *Advances in Neural Information Processing Systems*, [online] 2018-December, pp.3585–3597. Available at: http://arxiv.org/abs/1806.07336 [Accessed 11 Dec. 2020].

Bentley, J.L., Haken, D. and Saxe, J.B., (1980) A general method for solving divide-and-conquer recurrences. *ACM SIGACT News*, [online] 123, pp.36–44. Available at: https://dl.acm.org/doi/10.1145/1008861.1008865 [Accessed 27 Sep. 2020].

Beyer, K., Goldstein, J., Ramakrishnan, R. and Shaft, U., (1998) When is “nearest neighbor” meaningful? In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. [online] Springer Verlag, pp.217–235. Available at: https://link.springer.com/chapter/10.1007/3-540-49257-7\_15 [Accessed 10 Dec. 2020].

Buch, L. and Andrzejak, A., (2019) Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection. In: *SANER 2019 - Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering*. Institute of Electrical and Electronics Engineers Inc., pp.95–104.

Chen, Z. and Monperrus, M., (2018) The Remarkable Role of Similarity in Redundancy-based Program Repair. *arXiv*. [online] Available at: http://arxiv.org/abs/1811.05703 [Accessed 10 Dec. 2020].

Chen, Z. and Monperrus, M., (2019) A Literature Study of Embeddings on Source Code. [online] pp.1–8. Available at: http://arxiv.org/abs/1904.03061.

code2vec github, (2018) *GitHub - tech-srl/code2vec: TensorFlow code for the neural network presented in the paper: ‘code2vec: Learning Distributed Representations of Code’*. [online] Available at: https://github.com/tech-srl/code2vec [Accessed 27 Sep. 2020].

Cosentino, V., Luis, J., Izquierdo, C. and Cabot, J., (2016) Findings from GitHub: Methods, datasets and limitations. *Proceedings - 13th Working Conference on Mining Software Repositories, MSR 2016*, pp.137–141.

DeFreez, D., Thakur, A. V. and Rubio-Gonzalez, C., (2018) Path-based function embedding and its application to error-handling specification mining. In: *ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. [online] New York, New York, USA: Association for Computing Machinery, Inc, pp.423–433. Available at: http://dl.acm.org/citation.cfm?doid=3236024.3236059 [Accessed 10 Dec. 2020].

Devlin, J., Uesato, J., Singh, R. and Kohli, P., (2017) Semantic Code Repair using Neuro-Symbolic Transformation Networks. *arXiv*. [online] Available at: http://arxiv.org/abs/1710.11054 [Accessed 10 Dec. 2020].

Dhyan, (2021) *GitHub - dhyan/code2vec\_code\_complexity*. [online] Available at: https://github.com/dhyan/code2vec\_code\_complexity [Accessed 9 Feb. 2021].

Dobravec, T., (2018) Estimating the time complexity of the algorithms by counting the Java bytecode instructions. In: *2017 IEEE 14th International Scientific Conference on Informatics, INFORMATICS 2017 - Proceedings*. Institute of Electrical and Electronics Engineers Inc., pp.74–79.

Gauvrit, N., Singmann, H., Soler-Toscano, F. and Zenil, H., (2016) Algorithmic complexity for psychology: a user-friendly implementation of the coding theorem method. *Behavior Research Methods*, [online] 481, pp.314–329. Available at: http://algorithmicnature.org [Accessed 6 Dec. 2020].

Gomes, V.M., Paiva, J.R.B., Reis, M.R.C., Wainer, G.A. and Calixto, W.P., (2019) Mechanism for Measuring System Complexity Applying Sensitivity Analysis. *Complexity*, 2019.

Gu, X., Zhang, H., Zhang, D. and Kim, S., (2016) Deep API Learning. *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, [online] 13-18-November-2016, pp.631–642. Available at: http://arxiv.org/abs/1605.08535 [Accessed 11 Dec. 2020].

Harer, J.A., Kim, L.Y., Russell, R.L., Ozdemir, O., Kosta, L.R., Rangamani, A., Hamilton, L.H., Centeno, G.I., Key, J.R., Ellingwood, P.M., Antelman, E., Mackay, A., McConley, M.W., Opper, J.M., Chin, P. and Lazovich, T., (2018) Automated software vulnerability detection with machine learning. *arXiv*. [online] Available at: http://arxiv.org/abs/1803.04497 [Accessed 10 Dec. 2020].

Henkel, J., Lahiri, S.K., Liblit, B. and Reps, T., (2018) Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces. *ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, [online] 18, pp.163–174. Available at: http://arxiv.org/abs/1803.06686 [Accessed 11 Dec. 2020].

Hindle, A., Barr, E.T., Su, Z., Gabel, M. and Devanbu, P., (2012) On the naturalness of software. In: *Proceedings - International Conference on Software Engineering*. pp.837–847.

Jan Šimbera, (2019) *pysynth · PyPI*. [online] github. Available at: https://pypi.org/project/pysynth/ [Accessed 27 Sep. 2020].

Jayram, T.S., Kumar, R. and Sivakumar, D., (2003) Two applications of information complexity. In: *Proceedings of the thirty-fifth ACM symposium on Theory of computing - STOC ’03*. [online] New York, New York, USA: Association for Computing Machinery (ACM), p.673. Available at: http://portal.acm.org/citation.cfm?doid=780542.780640 [Accessed 6 Dec. 2020].

Klir, G.J. and Simon, H.A., (1991) The Architecture of Complexity. In: *Facets of Systems Science*. [online] Springer US, pp.457–476. Available at: https://link.springer.com/chapter/10.1007/978-1-4899-0718-9\_31 [Accessed 6 Dec. 2020].

Kumarsharma, D., Vohra, S., Gupta, T. and Goyal, V., (2018) Predicting the Algorithmic Time Complexity of Single Parametric Algorithms Using Multiclass Classification with Gradient Boosted Trees. *2018 11th International Conference on Contemporary Computing, IC3 2018*, pp.2–4.

Lehre, P.K. and Özcan, E., (2013) A runtime analysis of simple hyper-heuristics: To mix or not to mix operators. *FOGA 2013 - Proceedings of the 12th ACM Workshop on Foundations of Genetic Algorithms*, pp.97–104.

Li, J., He, P., Zhu, J. and Lyu, M.R., (2017) Software defect prediction via convolutional neural network. In: *Proceedings - 2017 IEEE International Conference on Software Quality, Reliability and Security, QRS 2017*. Institute of Electrical and Electronics Engineers Inc., pp.318–328.

Lloyd, S., (2001) Measures of Complexity: A Nonexhaustive List. *IEEE Control Systems*, 214, pp.7–8.

Lu, M., Liu, Y., Li, H., Tan, D., He, X., Bi, W. and Li, W., (2019) Hyperbolic function embedding: Learning hierarchical representation for functions of source code in hyperbolic space. *Symmetry*, 112.

M, H. and M.N, S., (2015) A Review on Evaluation Metrics for Data Classification Evaluations. *International Journal of Data Mining & Knowledge Management Process*, 52, pp.01–11.

Martin, F., (2018) *Refactoring*. [online] Available at: https://martinfowler.com/books/refactoring.html [Accessed 10 Dec. 2020].

McCabe, T.J., (2020) *A complexity measure | Proceedings of the 2nd international conference on Software engineering*. [online] ICSE ’76: Proceedings of the 2nd international conference on Software engineering. Available at: https://dl.acm.org/doi/10.5555/800253.807712 [Accessed 27 Sep. 2020].

Mikolov, T., Chen, K., Corrado, G. and Dean, J., (2013) Efficient estimation of word representations in vector space. In: *1st International Conference on Learning Representations, ICLR 2013 - Workshop Track Proceedings*. [online] International Conference on Learning Representations, ICLR. Available at: http://ronan.collobert.com/senna/ [Accessed 8 Dec. 2020].

Nagaraj, N., Balasubramanian, K. and Dey, S., (2013) A new complexity measure for time series analysis and classification. *European Physical Journal: Special Topics*, 2223–4, pp.847–860.

Narayanan, A., Chandramohan, M., Venkatesan, R., Chen, L., Liu, Y. and Jaiswal, S., (2017) graph2vec: Learning Distributed Representations of Graphs. *28th Modern Artificial Intelligence and Cognitive Science Conference, MAICS 2017*, [online] pp.189–190. Available at: http://arxiv.org/abs/1707.05005.

Nguyen, T.D., Nguyen, A.T. and Nguyen, T.N., (2016) Mapping API elements for code migration with vector representations. In: *Proceedings - International Conference on Software Engineering*. [online] New York, NY, USA: IEEE Computer Society, pp.756–758. Available at: https://dl.acm.org/doi/10.1145/2889160.2892661 [Accessed 11 Dec. 2020].

Oliveto, P.S., He, J. and Yao, X., (2007) Time complexity of evolutionary algorithms for combinatorial optimization: A decade of results. *International Journal of Automation and Computing*, 43, pp.281–293.

Padua, D., Ghoting, A., Gunnels, J.A., Squillante, M.S., Meseguer, J., Cownie, J.H., Roweth, D., Adve, S. V., Boehm, H.J., McKee, S.A., Wisniewski, R.W., Karypis, G., Malony, A.D., Gottlieb, S., Riesen, R., Maccabe, A.B., Bilardi, G., Pietracaprina, A., Kejariwal, A., Nicolau, A., Lengauer, C., Gustafson, J.L., Gropp, W., Prost, J.-P., Padua, D., Lowney, G., Amestoy, P., Buttari, A., Duff, I., Guermouche, A., L’Excellent, J.-Y., Uçar, B., Halstead, R.H., Nemirovsky, M., Amestoy, P., Buttari, A., Duff, I., Guermouche, A., L’Excellent, J.-Y., Uçar, B. and Pakin, S., (2011) Moore’s Law. In: *Encyclopedia of Parallel Computing*. [online] Boston, MA: Springer US, pp.1177–1184. Available at: http://link.springer.com/10.1007/978-0-387-09766-4\_81 [Accessed 6 Dec. 2020].

Papadimitriou, C., (2014) Algorithms, complexity, and the sciences. *Proceedings of the National Academy of Sciences of the United States of America*, [online] 11145, pp.15881–15887. Available at: www.pnas.org/cgi/doi/10.1073/pnas.1416954111 [Accessed 6 Dec. 2020].

Pradel, M., Darmstadt, T. and Koushik Sen, G., (n.d.) *DeepBugs: A Learning Approach to Name-based Bug Detection*. [online] Available at: https://lgtm.com/ [Accessed 11 Dec. 2020].

Rubinstein, R., (1999) The Cross-Entropy Method for Combinatorial and Continuous Optimization. *Methodology And Computing In Applied Probability*, [online] 12, pp.127–190. Available at: https://link.springer.com/article/10.1023/A:1010091220143 [Accessed 13 Feb. 2021].

Sikka, J., Satya, K., Kumar, Y., Uppal, S., Shah, R.R. and Zimmermann, R., (2020) Learning Based Methods for Code Runtime Complexity Prediction. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 12035 LNCS, pp.313–325.

Sikka, J.K.A., (2019) *GitHub - midas-research/corcod-dataset: CoRCoD: Code Runtime Complexity Dataset*. [online] MIDAS Research Laboratory, IIIT-Delhi. Available at: https://github.com/midas-research/corcod-dataset [Accessed 27 Sep. 2020].

SlattonK. Clint, B.L.E., (2005) *Source Code Repository - an overview | ScienceDirect Topics*. [online] Available at: https://www.sciencedirect.com/topics/computer-science/source-code-repository [Accessed 6 Dec. 2020].

Smilkov, D., Thorat, N., Nicholson, C., Reif, E., Viégas, F.B. and Wattenberg, M., (2016) Embedding Projector: Interactive Visualization and Interpretation of Embeddings. [online] Available at: http://arxiv.org/abs/1611.05469 [Accessed 12 Dec. 2020].

Soler-Toscano, F. and Zenil, H., (2017) A Computable Measure of Algorithmic Probability by Finite Approximations with an Application to Integer Sequences. *Complexity*, 2017.

Srikant, S. and Aggarwal, V., (2014) A system to grade computer programming skills using machine learning. In: *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. [online] New York, New York, USA: Association for Computing Machinery, pp.1887–1896. Available at: http://dl.acm.org/citation.cfm?doid=2623330.2623377 [Accessed 6 Dec. 2020].

Stanford encyclopedia, (2020) *Computational Complexity Theory (Stanford Encyclopedia of Philosophy)*. [online] Available at: https://plato.stanford.edu/entries/computational-complexity/ [Accessed 3 Dec. 2020].

SUMIT VOHRA vipul goyal, (2017) *Predicitng\_Time\_Complexity/Output\_Data at master · saisumit/Predicitng\_Time\_Complexity · GitHub*. [online] github. Available at: https://github.com/saisumit/Predicitng\_Time\_Complexity/tree/master/Output\_Data [Accessed 27 Sep. 2020].

Sun, Z., Zhu, Q., Mou, L., Xiong, Y., Li, G. and Zhang, L., (2019) A Grammar-Based Structural CNN Decoder for Code Generation. *Proceedings of the AAAI Conference on Artificial Intelligence*, [online] 3301, pp.7055–7062. Available at: https://github.com/zysszy/GrammarCNN [Accessed 27 Sep. 2020].

White, M., Tufano, M., Martinez, M., Monperrus, M. and Poshyvanyk, D., (2019) Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities. *SANER 2019 - Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering*, pp.479–490.

Wirth, N., (1995) *Wirth (1995) IEEE Computer.pdf*. *Cybersquare*, .

Yao, Z., Sun, Y., Ding, W., Rao, N. and Xiong, H., (2018) Dynamic word embeddings for evolving semantic discovery. *WSDM 2018 - Proceedings of the 11th ACM International Conference on Web Search and Data Mining*, 2018-Febua, pp.673–681.

Yonai, H., Hayase, Y. and Kitagawa, H., (2019) Mercem: Method Name Recommendation Based on Call Graph Embedding. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, 2019-Decem, pp.134–141.