



Dissertation on

“Predicting Code Runtime Complexity using Static Code Analysis and Machine Learning Techniques”

Submitted in partial fulfilment of the requirements for the award of degree of

**Bachelor of Technology
in
Computer Science & Engineering**

UE19CS390A – Capstone Project Phase - 1

Submitted by:

Deepa Shree C V	PES2UG19CS105
Jaaswin D Kotian	PES2UG19CS156
Nidhi Gupta	PES2UG19CS256
Nikhil M Adyapak	PES2UG19CS257

Under the guidance of

Prof. Ananthanagu
Assistant Professor, CSE Department
PES University

January - May 2022

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY**

(Established under Karnataka Act No. 16 of 2013)
Electronic City, Hosur Road, Bengaluru – 560 100, Karnataka, India



PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013)
Electronic City, Hosur Road, Bengaluru – 560 100, Karnataka, India

FACULTY OF ENGINEERING

CERTIFICATE

This is to certify that the dissertation entitled

‘Predicting Code Runtime Complexity using Static Code Analysis and Machine Learning Techniques’

is a bonafide work carried out by

Deepa Shree C V
Jaaswin D Kotian
Nidhi Gupta
Nikhil M Adyapak

PES2UG19CS105
PES2UG19CS156
PES2UG19CS256
PES2UG19CS257

In partial fulfilment for the completion of sixth semester Capstone Project Phase - 1 (UE19CS390A) in the Program of Study -Bachelor of Technology in Computer Science and Engineering under rules and regulations of PES University, Bengaluru during the period Jan. 2022 – May. 2022. It is certified that all corrections / suggestions indicated for internal assessment have been incorporated in the report. The dissertation has been approved as it satisfies the 6th semester academic requirements in respect of project work.

Signature
Prof. Ananthanagu
Assistant Professor

Signature
Dr. Sandesh B J
Chairperson
External Viva

Signature
Dr. B K Keshavan
Dean of Faculty

Name of the Examiners

Signature with Date

1. _____

2. _____

DECLARATION

We hereby declare that the Capstone Project Phase - 1 entitled “**Predicting Code Runtime Complexity using Static Code Analysis and Machine Learning Techniques**” has been carried out by us under the guidance of **Prof. Ananthanagu**, Assistant Professor, CSE Department PES University and submitted in partial fulfilment of the course requirements for the award of degree of **Bachelor of Technology in Computer Science and Engineering of PES University, Bengaluru** during the academic semester January – May 2022. The matter embodied in this report has not been submitted to any other university or institution for the award of any degree.

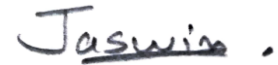
PES2UG19CS105

Deepa Shree C V



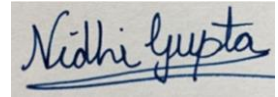
PES2UG19CS156

Jaaswin D Kotian



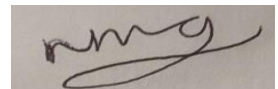
PES2UG19CS256

Nidhi Gupta



PES2UG19CS257

Nikhil M Adyapak



ACKNOWLEDGEMENT

I would like to express my gratitude to Prof. Ananthanagu, Assistant Professor, Department of Computer Science and Engineering, PES University, for his continuous guidance, assistance, and encouragement throughout the development of this UE19CS390A - Capstone Project Phase – 1.

I am grateful to the Capstone Project Coordinators, Dr. Sarasvathi V, Associate Professor and Dr. Sudeepa Roy Dey, Assistant Professor, for organizing, managing, and helping with the entire process.

I take this opportunity to thank Dr. Sandesh B J, Professor & Chairperson, Department of Computer Science and Engineering, PES University, for all the knowledge and support I have received from the department. I would like to thank Dr. B.K. Keshavan, Dean of Faculty, PES University for his help.

I am deeply grateful to Dr. M. R. Doreswamy, Chancellor, PES University, Prof. Jawahar Doreswamy, Pro Chancellor – PES University, Dr. Suryaprasad J, Vice-Chancellor, PES University for providing to me various opportunities and enlightenment every step of the way. Finally, this Capstone Project could not have been completed without the continual support and encouragement I have received from my family and friends.

ABSTRACT

In computer science, for any coding problem there are many ways of solving it. These solutions may apply different algorithms, logic to achieve the same solution. The problem caused by this is that certain algorithms tend to perform poorer with the increase in the amount of input. Hence the challenge then becomes to find the most efficient way to solve a problem.

There are multiple metrics using which the quality of any code can be analysed. One of these metrics is the runtime complexity. To calculate this runtime complexity, extensive analysis and comprehensive knowledge of algorithms is required which is a difficult task to do so manually. Hence, we would like to help automate the process of calculation of the runtime complexity by using ML techniques and static analysis of code for languages such as C, Java, and Python as of now. This can be extended to other coding languages later.

TABLE OF CONTENTS

Chapter No.	Title	Page No.
1.	INTRODUCTION	01
2.	PROBLEM DEFINITION	03
3.	LITERATURE SURVEY	04
	3.1 Learning Based Methods for Code Runtime Complexity Prediction	04
	3.2 Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection	08
	3.3 Code Characterization with Graph Convolutions and Capsule Networks	14
	3.4 Runtime Performance Prediction for Deep Learning Models with Graph Neural Network	16
	3.5 Graph Neural Network-based Vulnerability Predication	18
	3.6 A Novel Neural Source Code Representation Based on Abstract Syntax Tree	21
4.	DATA	25
	4.1 Overview	25
	4.2 Dataset	25
5.	SYSTEM REQUIREMENTS SPECIFICATION	28
	5.1 Introduction	28
	5.1.1 Project Scope	28
	5.2 Product Perspective	29
	5.2.1 Product Features	30
	5.2.2 User Classes and Characteristics	31
	5.2.3 Operating Environment	31
	5.2.4 General Constraints, Assumptions and Dependencies	32
	5.2.5 Risks	32
	5.3 Functional Requirements	33
	5.3.1 Code Language Recognition	33
	5.3.2 AST Representation of the code	33

5.3.3	Creation of Code Embeddings	34
5.3.4	The Model	34
5.4	External Interface Requirements	35
5.4.1	User Interfaces	35
5.4.2	Hardware Requirements	35
5.4.3	Software Requirements	35
5.4.4	Communication Interface	35
5.5	Non-Functional Requirements	36
5.5.1	Performance Requirement	36
5.5.2	Safety Requirements	36
5.5.3	Security Requirements	36
5.6	Other Requirements	36
6.	SYSTEM DESIGN	37
6.1	Introduction	37
6.2	Current System	37
6.3	Design Considerations	37
6.3.1	Design Goals	37
6.3.2	Architecture Choices	38
6.3.3	Constraints, Assumptions and Dependencies	39
6.4	High Level System Design	40
6.5	Design Description	41
6.5.1	Master Class Diagram	41
6.6	User Interface Diagrams	42
6.7	Sequence Diagram	43
6.8	Report Layouts	44
6.9	Design Details	45
6.10	Help	47
7.	IMPLEMENTATION AND PSEUDOCODE	48
8.	CONCLUSION OF CAPSTONE PROJECT PHASE - 1	53
9.	PLAN OF WORK FOR CAPSTONE PROJECT PHASE - 2	54
	REFERENCES	55
	APPENDIX A DEFINITIONS, ACRONYMS AND ABBREVIATIONS	58

LIST OF TABLES

Table No.	Title	Page No.
Table 1	Execution time of Binary Search vs Simple Search for different no. of inputs	1
Table 2	Data distribution and Extracted Features	4
Table 3	Measurement Metrics and per Feature Accuracy score	6
Table 4	Measurement Metrics for graph2vec and Data Ablation Test	7
Table 5	Overview of the dataset	12
Table 6	Overview of the experiment results	13
Table 7	Metadata at Dataset Level	26
Table 8	Metadata at Problem Level	27
Table 9	Submission metadata	27
Table 10	User Class and Characteristics	31
Table 11	Report Layouts	44

LIST OF FIGURES

Figure No.	Title	Page No.
Fig 1.	Siamese Networks	9
Fig 2.	A code snippet and it's AST representation	11
Fig 3.	T-SNE projection of the data	12
Fig 4.	SiCaGCN Architectural modules	14
Fig 5.	DNN Perf Module	16
Fig 6.	Architecture of AST-based Neural Network	23
Fig 7.	Statement encoder	23
Fig 8.	An example of AST Statement nodes	24
Fig. 9	Data Flow Diagram	40
Fig 10.	Master Class Diagram	41
Fig 11.	User Interface diagram	42
Fig 12.	Sequence Diagram	43

CHAPTER 1

INTRODUCTION

In computer science, for any coding problem there are many ways of solving it. These solutions may apply different algorithms, logic to achieve the same solution. The problem caused by this is that certain algorithms tend to perform poorer with the increase in the amount of input. Hence the challenge then becomes to find the most efficient way to solve a problem.

There are multiple metrics using which the quality of any code can be analysed. One of these metrics is the runtime complexity. To calculate this runtime complexity, extensive analysis and comprehensive knowledge of algorithms is required which is a difficult task to do so manually. Hence, we would like to help automate the process of calculation of the runtime complexity by using ML techniques and static analysis of code for languages such as C, Java, and Python as of now. This can be extended to other coding languages later.

	SIMPLE SEARCH	BINARY SEARCH
100 ELEMENTS	100 ms	7ms
10,000 ELEMENTS	10 seconds	14ms
1,000,000,000 ELEMENTS	11 days	32 ms

Table 1: Execution time of Binary Search vs Simple Search for different no. of inputs

This makes it very important to know how the execution time of a code increases with the increase in input size. This is where the Big O notation helps in the analysis of performance of algorithms. It tells us how fast an algorithm is. Big O notation doesn't tell us about the speed in seconds but it allows us to compare the number of operations. It tells us how fast an algorithm grows. Some of the most used representations are $O(n)$, $O(\log(n))$, etc.

Calculation of this runtime complexity manually is a pretty difficult and time-consuming task. Hence by using ML techniques, we will try to eliminate this manual work by predicting the runtime complexity of any given coding problem to the model. This will be helpful in eliminating the manual work and in analysing the performance of the codes which will help the coders to make better coding decisions as per their requirements. For helping our model to learn and perform efficiently, we will be making use of a subset of the dataset called the CodeNet dataset by IBM.

CHAPTER 2

PROBLEM DEFINITION

Given a code snippet we aim to predict the runtime complexity of it in the form of Big-O Notation, without having to execute it, using Machine Learning Techniques. We aim to build a model that supports coding languages such as C, Java, and Python.

By doing this, we aim to achieve an efficient method to calculate the runtime complexity, so that no manual effort is needed and save the user's time, as most of the solutions available require a set of inputs from the user's end to obtain the runtime.

CHAPTER 3

LITERATURE SURVEY

[1] Sikka, Jagriti & Satya, Kushal & Singla, Yaman & Uppal, Shagun & Shah, Rajiv Ratn & Zimmermann, Roger. (2020). Learning Based Methods for Code Runtime Complexity Prediction. 10.1007/978-3-030-45439-5_21

This paper aims to predict the runtime complexity of the given code snippet. The authors have mainly used supervised learning methods like SVM. The dataset they have used is called CORCOD dataset, which is one of its kind and it is made by the authors themselves. It stands for code runtime complexity dataset and is made up of around 1000 java code snippets.

The code snippets are web scraped from a competitive coding platform called Codeforces. This website has a wide range of code snippets based on data structures and algorithms and also their runtime complexity.

They have used ASTs, abstract syntax trees, to label the dataset. Since, the code snippets are in java, they have extracted the features using java packages and libraries.

The code snippets have been classified into five main classes which is given in the table below.

Complexity class		Features from Code Samples	
	Number of samples	Number of methods	Number of breaks
$O(n)$	385	Number of switches	Number of loops
$O(n^2)$	200	Priority queue present	Sort present
$O(n \log n)$	150	Hash map present	Hash set present
$O(1)$	143	Nested loop depth	Recursion present
$O(\log n)$	55	Number of variables	Number of ifs
		Number of statements	Number of jumps

Table 2: Data distribution and Extracted Features

One main thing to note here is classes which involve multiple variables in the Big-O Notation are present, that is notations like $O(m*n)$ are converted to $O(n)$. This makes the predictions a bit easier.

The classification models are trained based on two approaches:

- Getting the features of the code using static analysis
- By using code embeddings which is a general representation of the code by using ASTs

Preprocessing

Since all the code snippets are Java and not in any other language, the authors have used Eclipse JDT for extracting the features. This tool has components like ASTParser which is an object that creates an AST and the ASTVisitor object which traverses through the nodes of the tree using the approach of DFS.

A few unused sections of the code have to be removed but it is hard to remove them manually. Hence, the authors used JDT plugins to identify the methods that are invoked from the main function and have used only these methods to extract the useful features.

To generate embeddings from ASTs, a lot of various techniques are available. The authors have used graph2vec to generate embeddings since this technique can work even if less data is available unlike the others available. This technique is useful to generate task independent embeddings from a syntax tree.

This technique is similar to doc2vec for documents. The objective of graph2vec is to compute embeddings for each graph given a set of graphs. It identifies the non-linear structures and extracts them. The non-linear graph substructures are also called rooted subgraphs.

All the ASTs have two properties: Node Type and the Node Value. But the algorithm being used requires the graphs to have a single label. So, to achieve this goal, the graph can follow two different formats:

- Merging the Node Type and Node
- Value
- Choosing each of the nodes for each of the given node type and making a decision whether to include the node or not

For both the above formats, the authors use graph2vec to generate 1024-dimensional embeddings. These are then passed as the input to train the SVM model.

Training the model

Moving on to the part where the classification model is built and trained, out of all the experiments performed SVM gave out the most promising results.

Since, less amount of data is available, the authors chose to work on Machine Learning algorithms. Also, these are computationally cheaper than other methods available, since they take less time to train and learn from the data.

The tables below show the performance metrics of the various machine learning algorithms used by the authors.

Algorithm	Accuracy	Precision	Recall	Feature	Mean Accuracy
K-means	54.76	54.34	53.95	No. of ifs	44.35
Random forest	74.26	70.85	73.19	No. of switches	44.38
Naive Bayes	57.75	60.35	58.06	No. of loops	51.33
k-Nearest	59.89	59.35	58.57	No. of breaks	43.85
Logistic Regression	73.19	72.89	73.19	Priority Queue present	45.45
Decision Tree	73.79	71.86	71.12	No. of sorts	52.40
MLP Classifier	63.10	59.13	58.28	Hash Set present	44.38
SVM	72.96	69.43	70.58	Hash Map present	43.85
				Recursion present	42.38
				Nested loop depth	66.31
				No. of Variables	42.78
				No. of methods	42.19
				No. of jumps	43.65
				No. of statements	44.18

Table 3: Measurement Metrics and per Feature Accuracy score

The authors have used SVM models on the code embeddings extracted. This gave much higher accuracy and better results than on the manually extracted features.

A few data shuffling experiments were also carried out by the authors, to see that the models performed properly. A few methods are:

- Label shuffling: This method indicates whether the model is learning all the required features and leaving out the unnecessary ones.

- Variable name modification: This method ensures that the model doesn't depend on the variable names since this could lead to miscalculation of the results.
- Changing the input variables with constant values: As the input variables influence the runtime complexity of the code snippet, replacing these with constant variables will change the complexity with $O(1)$.
- Deleting a few graph substructures: Removing these means removing a few parts of the code, that is, loops, conditional blocks which means that the runtime will change for the snippet. Performing this experiment would help in determining the validity and accuracy of the predictions made by the model.

The above tables show the results of the models with different graph representations and data shuffling experiments.

AST Representation	Accuracy	Precision	Recall
Node Labels with concatenation	73.86	74	73
Node Labels without concatenation	70.45	71	70

Ablation Technique	Accuracy		
	Feature Engineering	Graph2vec: With Concatenation	Graph2vec: Without Concatenation
Label Shuffling	48.29	36.78	31.03
Method/Variable Name Alteration	NA	84.21	89.18
Replacing Input Variables with Constant Literals	NA	16.66	13.33
Removing Graph Substructures	66.92	87.56	88.96

Table 4: Measurement Metrics for graph2vec and Data Ablation Test

Main inferences from the paper

- Abstract Syntax Trees are more useful in static code analysis and hence code runtime complexity too.
- Embeddings of graphs give better results than models trained on features extracted manually.

[2] Buch, Lutz & Andrzejak, Artur. (2019). Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection. 95-104. 10.1109/SANER.2019.8668039.

The authors of this paper aim to perform code clone detection using Abstract Syntax trees and Recurrent Neural Networks. This task is very important in the field of software engineering as the maintenance of code authenticity is important, to ensure the standard of code. To automate this process supervised and unsupervised learning methods of Machine Learning have been employed. Abstract Syntax Trees help in achieving better results and improve the performance of the existing models.

The authors aim to leverage the power of ASTs and the information obtained by their aggregation to perform the task at hand. They also aim to segregate the train and the test data by clustering based on the ASTs.

Here, a Siamese Network is implemented as two code snippets have to pass at the same time to perform the task. They also stress that the most important factor for achieving better results is a pre-trained embedding. The more accurate the representation of the code is, the better the results.

The authors stress on the various ways of extracting embeddings for a given code. A few algorithms worth mentioning are word2vec, which works directly on the high-level source code. A few others are dependent on the graph representation of code like ASTs, Control Flow Graphs (CFGs), Directed Acyclic Graphs (DAGs).

Pre-processing

Word2vec holds a prominent position in extracting embeddings from a code snippet. It is of two types: CBOW and Skip-gram. The continuous bag of words is less complicated than the skip-gram version of this algorithm. This version of the algorithm is trained using shallow neural networks and it is used to identify whether a given token representation of the code fits in the context of the code and if it makes sense. This is called shallow since it is a three-layer network.

The authors use Recurrent Neural Networks to solve the problem. Since RNNs train iteratively, they are not static, and they perform better than normal ANNs. The input to these networks is a token sequence. The network goes through them one at a time and updates the weights accordingly. This goes on until the leaf node of the tree is reached. Hence, the whole tree is traversed through.

Siamese networks are sister networks where the structure of both the networks is the same and the output of both the networks are compared for the similarity. Various measures can be used to compute the similarity. These measures can be cosine similarity, Euclidean distance, or any other distance measure.

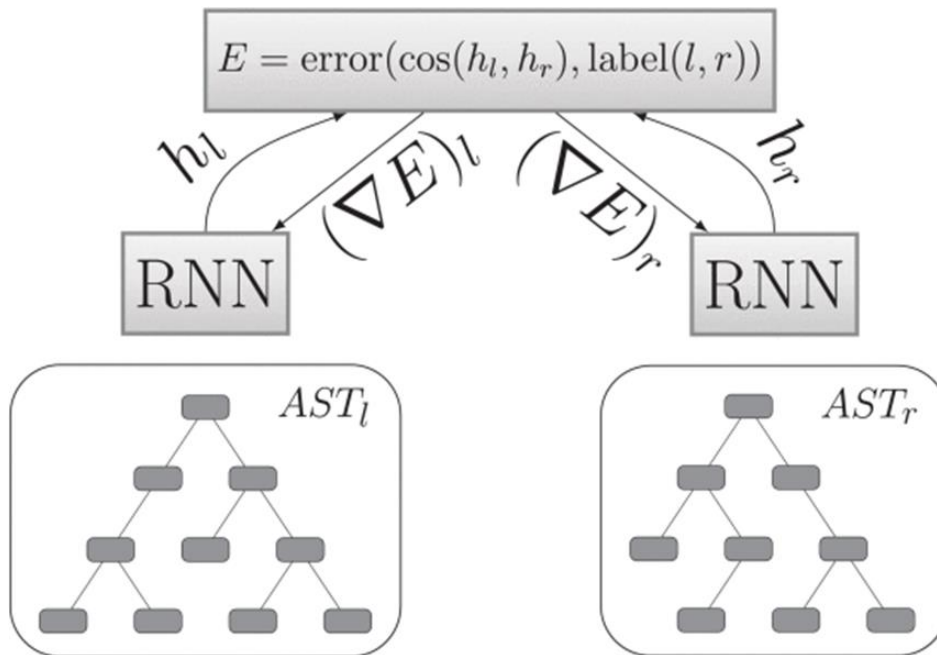


Fig 1: Siamese Networks

The diagram above shows the representation of the Siamese Neural Networks used to solve the current problem.

Since RNNs have the problem of vanishing and exploding gradients, another variation called LSTMs and GRUs are used. These overcome the problem with RNNs and give better results. The current paper implements LSTMs as they can trap the error and they can prevent unnecessary variance.

The most famous approach for code clone detection is getting vector representations of code using Deckard Method. This is used as a baseline model for the current solution.

Training the Model

The goal of the Siamese Networks is to maximize the cosine similarity for identical code pairs and output a low value of similarity for different pairs.

The ASTs have two node labels for every node, Node Type and Node Content. Node Type refers to the type of the node, that is, whether it is a keyword or a statement etc. The latter talks about the information present in the node, that is the actual values of the node.

The LSTM goes through two phases while training. One is the forward pass, and the other is Backward pass and Weight Update. During the forward pass, one unit of LSTM goes through the whole tree to get the information about all the nodes. The later phase will look into the error and the weights of the neural network. The weights of the network are updated accordingly, so that the error is reduced. The error is high when the similarity is high non clone pairs and the similarity is low for code clones. Backpropagation looks through the error of the network and then reduces it by updating the weights accordingly.

The dataset used is the BigCloneBench, which has code snippets in Java. One important point to be noted here is, trees with more than a thousand nodes are excluded which is equivalent to a depth of twenty-eight. This leads to the elimination of a few data points from the dataset.

To handle the problem of imbalanced classes, the errors are reduced. This is done so that the number of epochs required to train the model is reduced, as the main objective of a neural network is to minimize the error. Doing this would not only address the problem of class imbalance, but also reduce the computational expensiveness of the training process.

As the code snippets are all in java, JDT is used to generate the AST representation.

A sample AST representation of a code snippet is shown below:

```
public static void main(String args[]){
    System.out.println("Hello, World!");
}
```

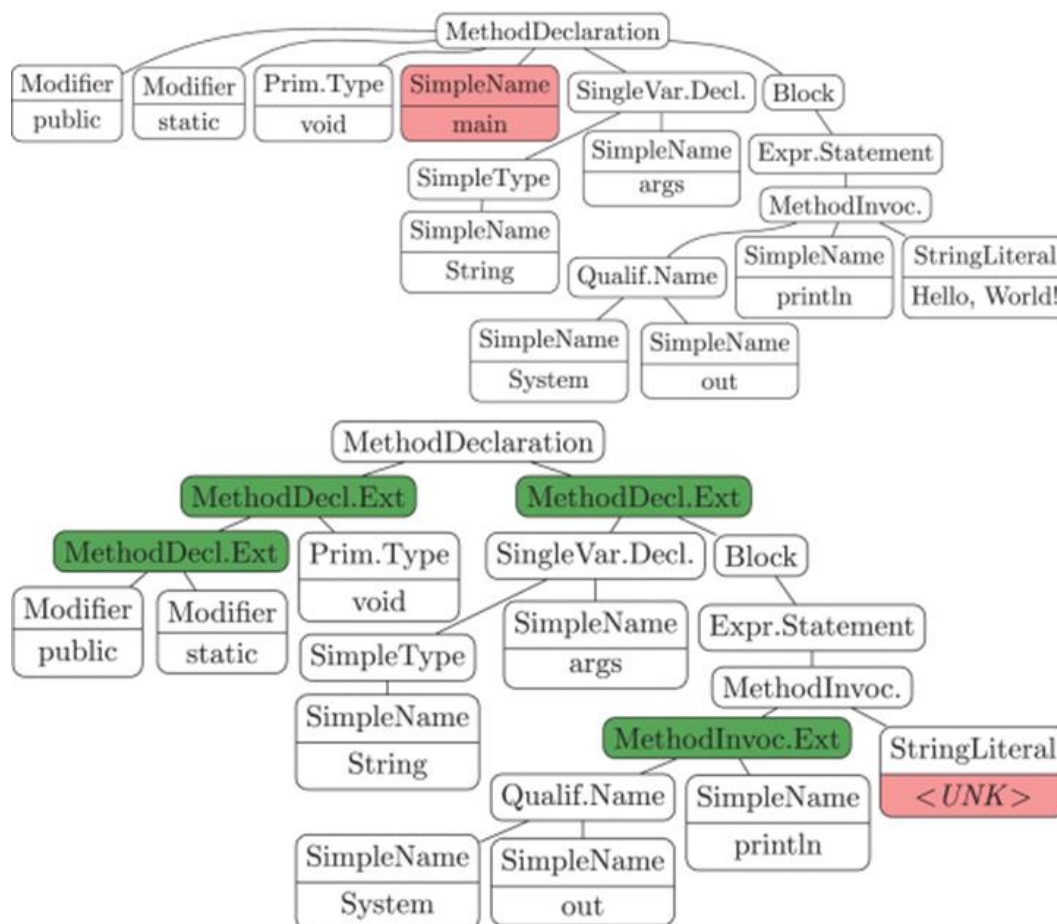


Fig 2: A code snippet and its AST representation

The performance metric used to assess the performance of the model is AUC. This metric is a function of the FP rate and the TP rate of the predictions, where FP stands for False Positives and TP stands for True Positives. The higher the value of AUC, the better the model.

Predicting Code runtime complexity using static code analysis and machine learning techniques

For the pretrained embeddings to make the input compatible to the neural networks, the algorithms, word2vec and SkipGram are used but with a few changes to make it compatible to the current solution approach. After the conversion to embeddings, the train and the test data is clustered and passed to the model. The picture below gives a brief overview of the dataset:

Fold	Training clusters	Validation cl.	Testing cl.	# Training pairs	# Validation pairs	# Testing pairs
0	4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 29	39, 43, 45, 42, 41, 38	33, 32, 30, 31, 34, 36	3,277 pos. 64,619 neg.	1,140 pos. 6,000 neg.	1,140 pos. 6,000 neg.
1	4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 31, 32, 33, 34, 36, 38, 39, 41, 42, 43, 45	23, 30, 29, 26, 25, 24	22, 21, 20, 19, 17, 18	3,482 pos. 69,289 neg.	1,070 pos. 5,600 neg.	1,005 pos. 5,100 neg.
2	18, 19, 20, 21, 22, 23, 24, 25, 26, 29, 30, 31, 32, 33, 34, 36, 38, 39, 41, 42, 43	11, 12, 13, 14, 15, 17	4, 5, 6, 7, 10, 45	3,920 pos. 82,400 neg.	841 pos. 4,109 neg.	796 pos. 3,482 neg.

Table 5: Overview of the dataset

The baseline model used, as already discussed, is the Deckard algorithm for code clone detection. To enhance the performance and hence achieve better results, hyperparameter tuning was carried out. Dimensionality reduction was also carried out, and the results were noted. This was achieved by using T-SNE. The image below shows the data in T-SNE projection.

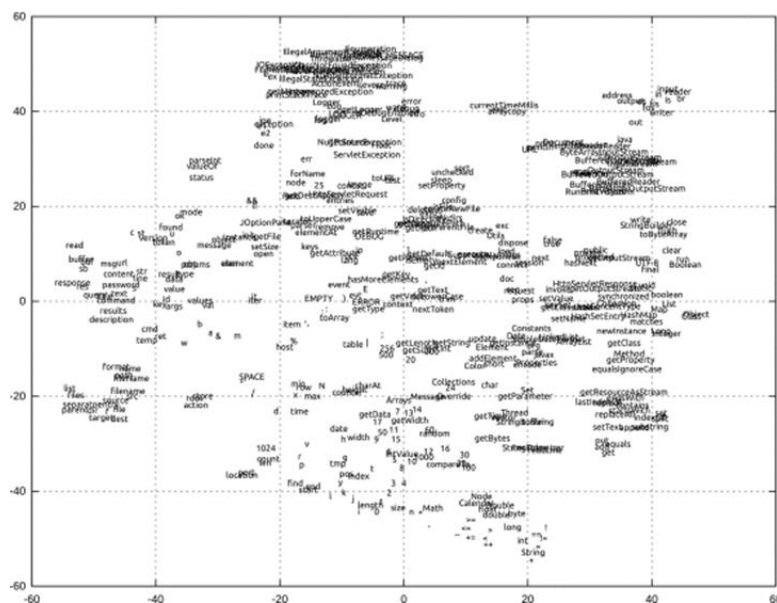


Fig 3: T-SNE projection of the data

The influence of embeddings and the variation of the network architecture were also taken into consideration to achieve better results. An overview of the results of the above experiments are shown below:

	dim.	#params	AUC₀	AUC₁	AUC₂	∅AUC
	300	752,294	0.799	0.771	0.903	0.824
	200	345,094	0.766	0.861	0.876	0.834
	150	201,494	0.764	0.841	0.929	0.845
	100	97,894	0.690	0.840	0.857	0.795
	50	34,294	0.613	0.817	0.793	0.741
	30	20,054	0.588	0.768	0.783	0.713
<i>Emb. ∅</i>	14	10,694	0.784	0.752	0.885	0.807
<i>D. cos.</i>	300	–	0.565	0.746	0.798	0.703
<i>D. Eucl.</i>	300	–	0.543	0.653	0.602	0.599

Table 6: Overview of the experiment results

Main inferences from the paper

- AST based embeddings tend to give better results than simple source-code based vector representations.
- High dimensional data can be prone to overfitting; hence care should be taken to regulate the dimensions.

[3] P. Haridas, G. Chennupati, N. Santhi, P. Romero and S. Eidenbenz, "Code Characterization With Graph Convolutions and Capsule Networks," in *IEEE Access*, vol. 8, pp. 136307-136315, 2020, doi: 10.1109/ACCESS.2020.3011909.

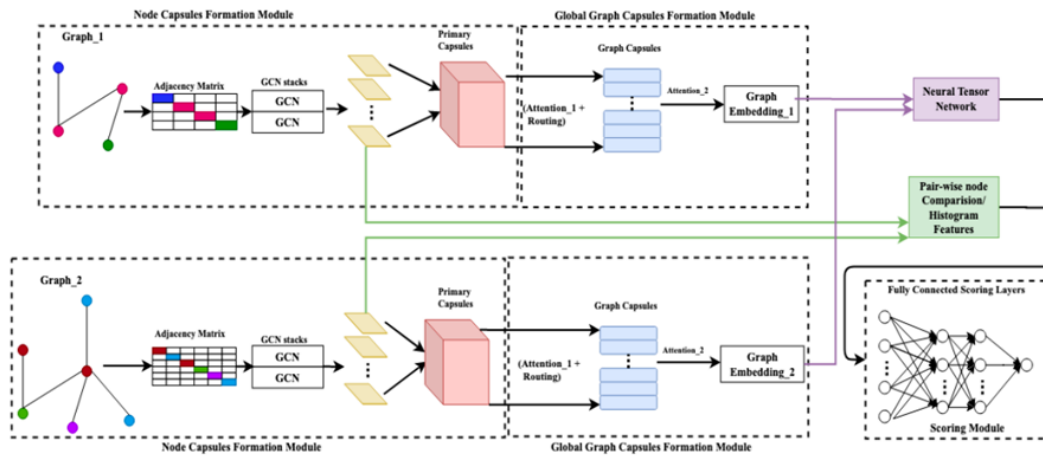


Fig 4: SiCaGCN Architectural modules

Objectives

This paper worked with representation of the programs as graphs in which they employed two common known graph similarity/distance metrics: Graph Edit Distance (GED) and maximum common subgraph metric to gain additional insights about the graphs. They formulated a unique method which utilized a neural tensor network to combine the results from the GNN and Time capsules. This combination helped them in improving the accuracy of their model which was able to predict the similarity of a given software code to a set of codes (represented as CFG's) that are permitted to run on a computational resource. This code characterization helped them to predict abusive codes which can help prevent security-based attacks.

Advantages

This model consists of the GCN architecture that captures different latent properties of programs using capsules. This produced a similarity metric of two graphs which can be extended to other graphs. It also helps in differentiating the codes easily through way of representation of the graph embeddings of code. With the added advantage of capsules, better graph embeddings were produced which optimized the model to be faster.

Limitations

- Although the accuracy of similarity value is high, the trade-off is the runtime.
- The dataset is limited to only C/C++ programs.

[4] Gao, Yanjie and Gu, Xianyu and Zhang, Hongyu and Lin, Haoxiang and Yang, Mao (2021) “Runtime Performance Prediction for Deep Learning Models with Graph Neural Network” /MSR-TR-2021-3/Microsoft

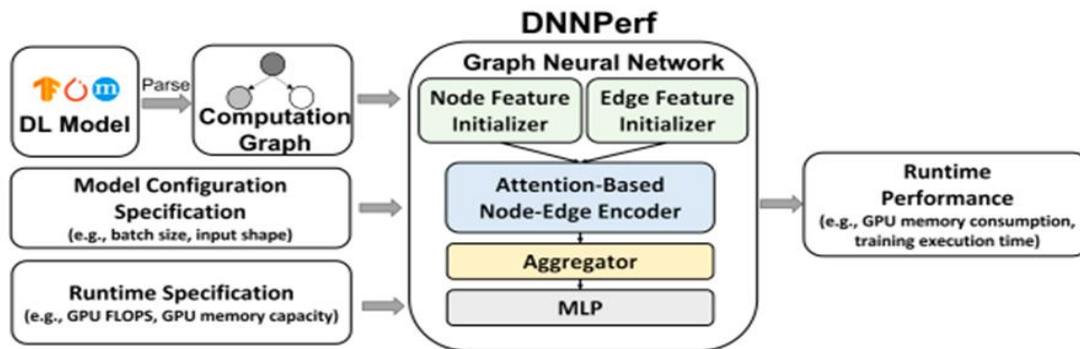


Fig 5: DNN Perf Module

Objectives

Their aim was to predict the runtime performance of DL models by utilizing Graph Neural Networks and a novel approach DNNPerf (a tool) on real world DL models and neural architecture search algorithms. Their model accepts a DL model file, a model configuration specification, and a runtime specification as input and reports the runtime performance values.

Their aim is to boost the overall productiveness of the neural architectures and to reduce the resource wastage due to the unsuitable configurations of hyperparameters mentioned. This can help prevent failed training jobs/inappropriate models.

Advantages

This approach effectively handles the irregularities caused by unequal node neighbours, which is the reason for choosing GNN. Their model achieved satisfactory precision and outperformed the baseline approaches, and all compared methods. DNNPerf has a good stability on diverse neural architectures, making it usable on different kinds of neural networks easily.

Limitations

- Since the DL models are usually implemented with proprietary NVIDIA, CUDA, etc. The internal implementation details are hidden which makes it difficult to arrive at the exact runtime performance metric.
- The configuration space/types of DL models vary due to which predicting runtime performance is not that highly accurate.

[5] Q. Feng, C. Feng, and W. Hong, "Graph Neural Network-based Vulnerability Predication," 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2020, pp. 800-801, doi: 10.1109/ICSME46990.2020.00096.

Objective

Since software vulnerability is a major threat in the security of software systems. There are two types of vulnerability detection technologies now available: static and dynamic. The static approaches look for weaknesses in the source or binary code. Static methods abstract the code that is being detected, which might lead to false positives. Dynamic approaches, on the other hand, run the code and check for vulnerabilities during execution. Dynamic approaches are accurate, but they have a problem with false negatives. Our main concept is to develop a static vulnerability detector using the source code with vulnerabilities as the specification and the deep neural network (DNN) approach. We offer a vulnerability detection system based on a generic graph neural network (GNN). To train the GNN model, which will be utilised later for vulnerability prediction, we extract distinct structures of the programmes with vulnerabilities as graphs.

This study proposes a graph neural network-based methodology for predicting code vulnerability. The framework's many manifestations in representative programme graph representations, starting node encodings, and GNN learning algorithms are investigated. The preliminary experimental findings on a relevant benchmark show that the GNN-based strategy may enhance vulnerability prediction accuracy and recall rates.

Framework

The GNN based framework was basically divided into three components i.e The code graph representation, graph structure encoding, model training and prediction.

Code graph representation

In this section, the information or the code is being represented in the form of graphs. Three graph representations have been considered here i.e

AST: A tree representation of the code program's structure

CFG: Deals with the data and control flow of the program

CPG: Basically, a combination of AST and CFGs

Graph structure encoding

Inorder to feed the data to our GNN model, the data needs to be converted into vector representations of code. So here we come up with three encoding strategies mainly ,

Random: Gives each node type a random vector

BoW: Determines the frequency of a node type in code structure

Word2Vec : Creates a vocabulary of the corpus and matches a word which is semantically closer to the target word.

Model training and prediction

Once the Embeddings have been created, it's been passed on to the neural network. Our framework predicates whether the function has any vulnerabilities or not. Here we have mentioned three GNN based learning methods for classification.

- Diffpool
- Set2Set
- DGCNN

Conclusion

We input the encoding strategies into each of the prediction models. Basically, it's a trial and error approach where we try various combinations to find which combination of graph representation, initial node encoding, and learning method is more effective for vulnerability prediction.

Our preliminary experimental results indicate that the combination, i.e., AST+Word2Vec+DiffPool, can achieve the best result on the benchmark.

Advantages

- It provides a function-level vulnerability prediction.
- Static vulnerability detection on the source code of the program under analysis.
- Solves scalability problems faced by traditional methods.

Limitations

- Restricted to C/C++ programs only

[6] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang and Xudong Liu ,“A Novel Neural Source Code Representation Based on Abstract Syntax Tree”, 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE) 1558-1225/19/\$31.00 ©2019 IEEE DOI 10.1109/ICSE.2019.00086

Objectives

Abstract Syntax Trees are sometimes very large and hence, storing and processing becomes very hard. In this paper, the researchers developed a novel way of representing the AST's called the abstract syntax tree based Neural Network. This method in their research was used for source code presentation. The AST was split into smaller blocks. This helps in storing and processing of individual blocks.

The research was conducted on two problem statements, and they were source code classification and code clone detection.

Advantages

- When traditional Information Retrieval Systems, trying to convert code to vectors assume code to be natural language texts. Hence, it is possible that we miss some important semantics from the code. Hence AST Neural Network can be used to help perform better than the traditional IR System.
- The AST Neural Network helps in capturing the lexical and syntactical knowledge of the code. Instead of loading and processing the whole AST, the AST Neural Network splits the large AST into smaller blocks encoding the statement trees to code embeddings or vectors.

- The vector equivalent of a code statement was obtained by passing the set of statement vectors into a bidirectional RNN model. In the research, these code embeddings or vector fragments were used to perform the tasks of source code processing and code summarization.
- The research also made use of tree based convolutional neural networks. This system took ASTs as the input and would calculate the node embeddings. The learning took place recursively in a bottom-up fashion in order to obtain the vectors.

Limitations

The limitations are subjected to their datasets such as

- The dataset used is the OJ dataset. Some of the code is complex and although the code is not industrial level code, the dataset does not have data collected from any production environment.
- Formation of clones of their dataset or repository since they were working on code clones' detection.
- It is limited to only Java and Python Programs.

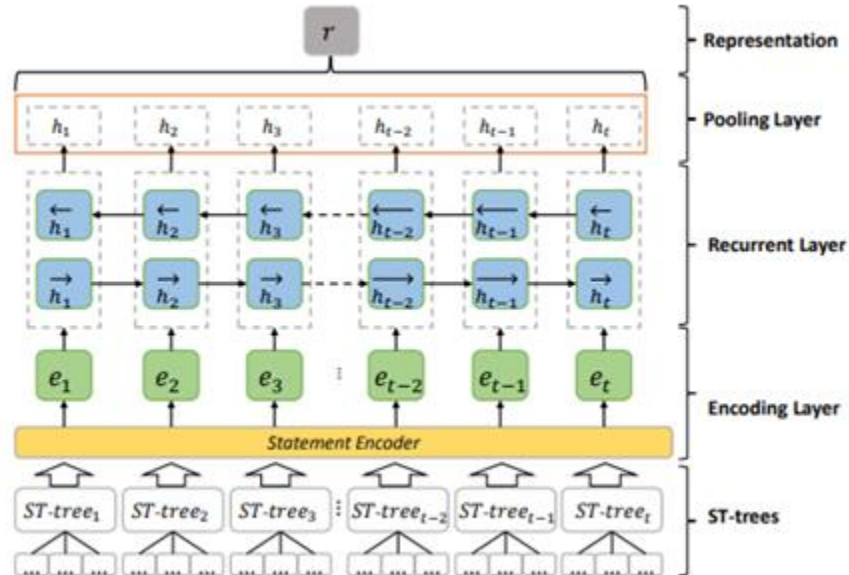


Fig 6: Architecture of AST-based Neural Network

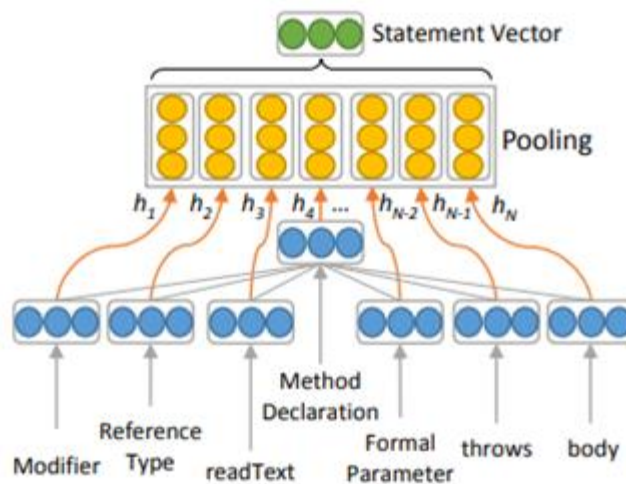


Fig 7: Statement encoder.

Blue - Initial embeddings. Orange - Hidden states. Green - Statement vector

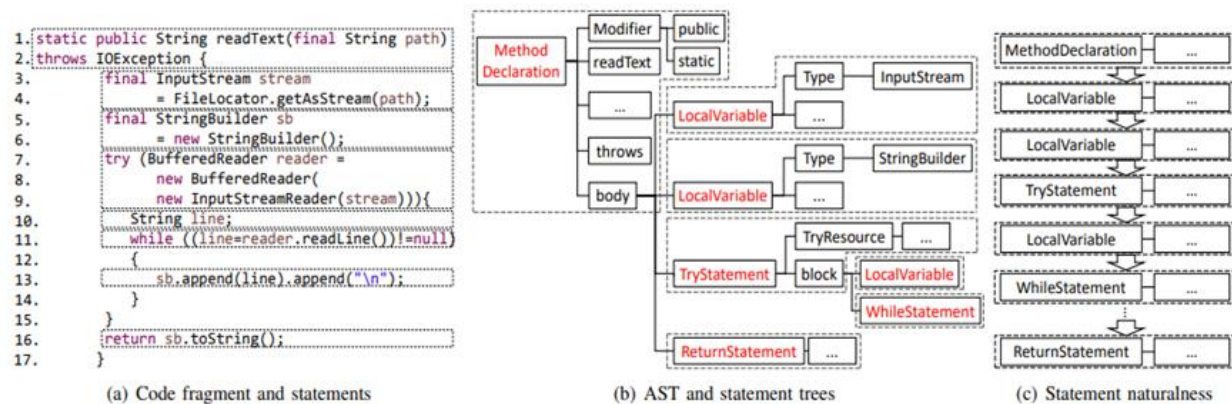


Fig 8: An example of AST Statement nodes

CHAPTER 4

DATA

4.1 Overview

The CodeNet dataset comprises a very large group of code samples in over 55 different languages with extensive metadata. These are derived from the data available on two online judge websites: AIZU and AtCoder. It consists of a very large collection of source files, extensive metadata, tooling to access the dataset and make tailored selections, and documentation. These documented tools are provided to transform code samples into various intermediate representations, to make tailored choices as per one's need and to help access the dataset easily.

Their goal is to provide the community with a large, high-quality curated dataset that can be used to advance AI techniques for source code. The dataset consists of submissions to the problems available on the two coding platforms AIZU and AtCoder. These are judged by an automated process for correctness. Submission outcomes, problem descriptions, and associated metadata are available via various REST APIs provided by these two websites.

4.2 Dataset

The dataset contains a total of 13,916,868 submissions, which are divided into 4053 problems. Among the submissions, 7,460,588 (53.6%) are accepted (compilable and pass the prescribed tests), 29.5% are marked with wrong answers, and the remaining are rejected due to their failure to meet run time or memory requirements.

To their knowledge, amongst similar kinds of datasets, this is the largest dataset. Submissions are in 55 different languages; 95% of them are coded in C++, Python, Java, C, Ruby, and C#. C++ is the most common language, with 8,008,527 submissions (57% of the total), of which 4,353,049 are accepted. With so many available code samples, the users can easily extract large benchmark datasets that are customized to their own downstream use.

The level of problems in the dataset range from elementary exercises to advanced problems that require sophisticated algorithms. The solution submitters range from experienced coders to beginners. All submissions are accordingly labelled according to the language used and wherever the solutions were found to be correct and erroneous. The submissions are in many different languages.

Every code sample is contained in a separate file which includes the inputs provided to the test cases and output of these computed results. The file names use the standard extensions which denote the programming language used, e.g., .c for C language. The majority of code snippets consist of only one function, whereas the submissions made to more complex problems have several functions.

Since the dataset is so huge, a metadata is provided to easily understand and find what is required from this dataset. The metadata helps with the choices among the large collection of problems, data queries, languages, and source files. The metadata is organized in two levels in two csv files. The first csv file is at the dataset level, which explains the composition of all the problems. The second csv file is at the solution level, which consists of all the details of the submissions made to a single problem.

Every problem consists of an HTML file which provides a detailed description of the question, its requirements and constraints along with the IO examples.

Additionally, the metadata and data in the dataset structure are separated. At the dataset level, a single CSV file lists all problems and their origins, along with the CPU time and memory limits set for them as shown in table 3. At the problem level, every problem has a CSV file.

name of column	data type	unit	description
id	string	none	unique anonymized id of the problem
name	string	none	short name of the problem
dataset	string	none	original dataset, AIZU or AtCoder
time_limit	int	millisecond	maximum time allowed for a submission
memory_limit	int	KB	maximum memory allowed for a submission
rating	int	none	rating, i.e., difficulty of the problem
tags	string	none	list of tags separated by " "; not used
complexity	string	none	degree of difficulty of the problem; not used

Table 7: Metadata at Dataset Level

name of column	data type	unit	description
submission_id	string	none	unique anonymized id of the submission
problem_id	string	none	anonymized id of the problem
user_id	string	none	anonymized user id of the submission
date	int	seconds	date and time of submission in the Unix timestamp format (seconds since the epoch)
language	string	none	mapped language of the submission (ex: C++14 ->C++)
original_language	string	none	original language specification
filename_ext	string	none	extension of the filename that indicates the programming language used
status	string	none	acceptance status, or error type
cpu_time	int	millisecond	execution time
memory	int	KB	memory used
code_size	int	bytes	size of the submission source code in bytes
accuracy	string	none	number of tests passed; *Only for AIZU

Table 8: Metadata at Problem Level

The only limitation of this dataset is that all the code snippets in the dataset are not extensively commented on for some solutions. As these comments are in various languages, AI techniques that rely on learning from preponderance of comments in the code may face challenges. The code samples are solutions to high-school and beginning college level programming problems. This dataset is not suitable for users looking for code with enterprise API's and advanced design patterns.

column	unit/example	description
submission_id	s[0-9]{9}	anonymized id of submission
problem_id	p[0-9]{5}	anonymized id of problem
user_id	u[0-9]{9}	anonymized user id
date	seconds	date and time of submission
language	C++	consolidated programming language
original_language	C++14	original language
filename_ext	.cpp	filename extension
status	Accepted	acceptance status, or error type
cpu_time	millisecond	execution time
memory	kilobytes	memory used
code_size	bytes	source file size
accuracy	4/4	passed tests (AIZU only)

Table 9: Submission metadata

CHAPTER 5

SYSTEM REQUIREMENTS SPECIFICATION

5.1 Introduction

Algorithm analysis is significant in computer science. Many algorithms can be used to solve a problem, but the most efficient one must be chosen. The greatest tool for predicting defect likelihood and code quality is to measure code runtime complexity. Maintenance becomes more costly and uncertain as the software life cycle gets more complicated. Additionally, greater code complexity is linked to a higher prevalence of code flaws, making the code more costly to maintain.

Time Complexity is one of the metrics used to measure the efficiency and quality of code written. The lesser the time complexity, the faster the execution of code.

Static code analysis is the process of predicting the output of a program without actually executing it. We would want to use this process to extract useful features from the code snippets as in the number of loops, the conditional flow and functions and libraries used.

5.1.1 Project Scope

It's no secret that writing, debugging, and maintaining code is difficult, yet it's important for excellent software quality. Furthermore, excessive code complexity leads to a larger number of faults, making the code more expensive to maintain. To accurately anticipate time complexity, it takes careful investigation and a thorough understanding of algorithms. An algorithm's runtime analysis is not only necessary for understanding its inner workings, but it also leads to a more effective implementation.

5.2 Product Perspective

Businesses have been watching out for opinions of their consumers on their products as well as their competitor's products since times immemorial. Tracking customer sentiments helps businesses to make better decisions for improving customer satisfaction as well as looking out for potential rivals to their products. The digital era has brought with it vast amounts of information about people. The most valuable sources of information from the company's perspective come from social media and online news sources. This application provides a platform for mining data from various social media sites and processing it to extract information. The client will be able to view all mentions about his(their) products and generate timely statistical reports which show aggregations of the data collected and public sentiments about their products.

Advantages of Time complexity

- It can make sure that every path has been tested at least once. Thus, help to focus more on uncovered paths.
- Code coverage can be improved.
- Risk associated with the program can be evaluated.

This project aims to come up with a Machine learning Approach to Predict Time complexity for a given problem solution and check its feasibility by using static analysis. We plan to implement this in four programming languages i.e., Java, python, C, C++ and classify them into five different time complexities i.e., $O(n)$, $O(\log n)$, $O(n \log n)$, $O(n^2)$, $O(n!)$. The code snippet would firstly be evaluated, preprocessed and represented in a Abstract Syntax Tree or Control Flow Graph format for further evaluation. The represented format would be converted into word/graph embeddings as an input to the neural network.

Graph Neural Networks (GNNs) are a type of deep learning algorithm that is used to infer data from graphs. GNNs are neural networks that can be applied directly to graphs, making node-level, edge-level, and graph-level prediction jobs simple. The aforementioned model was chosen for model training because of the high capability and performance benefits of graph neural networks. The model would be used to divide the data into five categories: $O(n)$, $O(\log n)$, $O(n \log n)$, $O(n^2)$, and $O(n!)$. We plan to develop and deploy the project as a flask/Django web application to make it easy for a user to test and analyse his code.

Limitations

Since static analysis is being followed, we assume that the code snippet written is free from compile/runtime errors and so the code written needs to meet the syntax of the language. Time complexity for languages like python needs to be dealt separately and so needs to have a different approach for analysing time complexity.

The Project limits its scope to only basic programming codes and not complex machine learning algorithms.

Size of the code snippet is limited (5mb of file size).

5.2.1 Product Features

As computer science students who do competitive programming on sites such as geeks for geeks, hacker rank, leetcode and code chef, most of us would fail certain test cases since it was failing the time constraint of the problem. Hence, we came up with the idea to automatically calculate the time complexity of the code submitted in order to help the user identify the time complexity of their code and would hence try to rectify it to pass the test cases. Our project is trying to extend it to software testing to predict time complexities of codes.

- Classification – Labels the code submitted to the programming language and classifies it as C, Java, Python
- Extract – Extract code features and converts the code to embeddings via line/code toVec.
- Tree conversion – Converts the code to an Abstract Syntax Tree for further processing
- Implementation classes using Machine Learning Models to classify the code into different time complexities is being decided.

Currently we are exploring different methods of extracting code in terms of abstract syntax tree representations, processing the code to make them into vectors for further easier processing and going through various Machine Learning Models to help classify the code into different classes of time complexities.

5.2.2 User Classes and Characteristics

User Class	Description
1.Local Systems	This will help the user get to know their code's time complexity on their local machine.
2.Online Users	The code time complexity shall be displayed via a web interface or might be implemented as a plugin in online code editors on competitions or coding platforms or reviewing code after tests/assignments integrated to educational institutes or websites.
3.Software Testing	To check for code exceeding the given time complexity which was estimated

Table 10: User Class and Characteristics

5.2.3 Operating Environment

The designed system is intended to be a website and will be available via any web-browser application. It will therefore be built to be compatible with all modern browsers and not be dependent on the technical capabilities or operating system of user's device. There might be a local machine implementation based on the time constraints of the project.

5.2.4 General Constraints, Assumptions and Dependencies

Constraints

- Time Complexity calculation of functions which are inbuilt and from imported libraries.
- Time complexity of the code will be calculated, however improving time complexity by suggesting better coding practices or code will not be implemented.
- Generating a common AST for all languages since libraries in different functions give different representations.
- Unavailability of sufficient data from online sources. This is due to the limited amount of research done in the field.
- Time complexity works differently for different languages
- Analysing time complexity for built in libraries needs a different approach
- Finding suitable tools to annotate the dataset with the target time complexity labels.
- As per Turing's Halting Problem, estimating code complexity was not possible.

Assumptions

- The code written needs to meet the syntax of the language and thus free from compile time and runtime errors.
- The code is simple and does not involve complex function calls or invokes complex inbuilt libraries and methods.
- The code is within 50 MB.
- The time complexity generated will be an approximation in terms of Big-O notations since we are assuming that it is going to be the worst case time complexity.

5.2.5 Risks

- Analysing time complexity for built in libraries needs a different approach and is a hassle
- Imbalanced Dataset
- Insufficient funds for deploying and scaling the service on cloud if the Machine learning models are computationally demanding

5.3 Functional Requirements

5.3.1 Code Language recognition

Description and Priority

- The user will provide a code snippet of size 50 mb maximum in either of the following languages:
 - C
 - Java
 - Python

Stimulus/Response Sequence

- User will provide the code snippet in a text file or appropriate language file in C, Java or Python.
- The language used in the code snippet will be identified.
- Further operations will be done based on the language identified.

Requirements

- A simple web interface for the user (if possible) otherwise, input through a simple terminal interface.
- Appropriate libraries for identification of the language used in the snippet.

5.3.2 AST Representation of the code

Description and Priority

The given code snippet is converted to an AST representation for further evaluation.

Stimulus/Response Sequence

- The language used in the code snippet will be provided as input along with the code file.
- This code will be converted into its respective AST representation.
- The corresponding AST representation of the code will be made available for further evaluation.

Requirements

To convert the given code snippet into its AST representation as every coding language has a different way to represent its AST.

To perform the AST representation of code in C, Python and Java.

5.3.3 Creation of Code Embeddings

Description and Priority

Using the AST representation of the code, we have to make code embeddings which will be fed into the Neural Network for evaluation of the runtime complexity.

Stimulus/Response Sequence

- The AST representation will be made available as input to the function.
- This function will help convert the AST to the code embeddings.
- This will be provided as input to the Neural Network for further evaluation

Requirements

To find an efficient way of creation of the code embeddings (Better the quality of the code embeddings, better will be the performance of the Neural Network in prediction.)

5.3.4 The Model

Description and Priority

- The vector embeddings extracted from the AST representation of the code are passed to a neural network.
- Neural Networks are Deep Learning based algorithms that can achieve high accuracy and better performance.
- We plan to implement the model, using Python and Google Colab, as we would need a GPU to train the model.
- The Deep Learning Framework we would be using is TensorFlow.

5.4 External Interface Requirements

5.4.1 User Interfaces

- If possible, we aim to have a web-based interface for user interaction.
- This would be implemented using Python's framework Flask.
- The input would be a file of a program snippet, and can be chosen from the any of the following languages:
 - C
 - Java
 - Python
- In case the file input by the user is not in the given languages, we aim to show a message saying the input language is not recognized and ask them to enter a file from the available languages.
- If time permits, we plan to extend the input to images. Here, the user can give in the input of a code snippet image. Using OCR, we aim to convert the image to text and extract the required information.
- The user is limited to a file size of 50 mb, exceeding which an error is thrown.

5.4.2 Hardware Requirements

- It can run on any device with a minimum RAM of 2 GB and running any modern web browsers.

5.4.3 Software Requirements

- The user interface is most likely to be web-based.
- Incase of availability of time, we plan to store the code snippet files entered by the user (only after getting the user's consent), to extend the dataset and improve the model performance. A NoSQL database, like MongoDB, is preferred for the above task.
- The website would run on any Operating System.
- The tools and libraries used would be Flask, HTML, Bootstrap, JavaScript.

5.4.4 Communication Interface

- HTTP protocol should be used as an interface of communication between client and server sides, as communication takes place through the internet.

5.5. Non-Functional Requirements

5.5.1 Performance Requirement

The model should be fast and should provide accurate results.

The terminal/web-based application should be easy to handle and use.

Servers should be able to handle the server load.

5.5.2 Safety Requirements

The application should be able to provide easy recovery and backup incase of server failure.

5.5.3 Security Requirements

It is an anonymous platform. The users aren't required to create any account for the usage of the application.

5.6 Other Requirements

We will be developing our model for demonstration purposes. We will try to provide a web interface for the same, if we get enough time.

CHAPTER 6

SYSTEM DESIGN

6.1 Introduction

This document provides a detailed description accompanied by diagrams which describes the design choices taken up for the product and the interaction between various components of the system. The purpose of this document is to add further detail to the requirements specification to prepare a model suitable for reference while coding.

6.2 Current System

Most of the current solutions present require the users to give in a series of inputs and based on the runtime of different sized inputs, the time complexity label is predicted. These solutions do not make use of Machine Learning or Deep Learning Techniques to do so.

Another existing solution to the current problem is a web interface called AProVE. It is also available for download. Here, static code analysis is performed, and the complexity label is predicted using various algorithms. This interface supports multiple languages like Java, C, Prolog, Haskell. But this solution does not use Machine Learning Techniques.

6.3 Design Considerations

6.3.1 Design Goals

- The main goal of the current project is to find a novel method of approach to find the runtime complexity of the given code snippet.
- Unlike the other solutions, we aim to build a model which supports three coding languages, Python, C and Java

6.3.2 Architecture Choices

Most of the currently available systems do not use machine learning approaches and are hence less reliable because the latter is considered to perform more efficiently than traditional methods in many scenarios.

We aim to implement Neural Networks like Graph Neural Networks and Recurrent Neural Networks, rather than supervised learning algorithms.

Pros:

- Neural Networks learn from events and make decisions by learning from similar events.
- Neural networks are observed to perform better and give more accurate results over other machine learning algorithms when the datasets are huge. This is one of the main advantages of using Neural Networks.
- Neural Networks have the ability to work even when there is no sufficient knowledge.

They do not need to be reprogrammed.

Cons:

- They usually require a huge number of resources and more time to train on.
- They often require processors which support parallel processing; hence their training can be expensive.
- Since there is no specific rule to determine the architecture of a neural network to solve a particular problem, a lot of work has to be put in as it is a trial-and-error process.

6.3.3 Constraints, Assumptions and Dependencies

Constraints:

- Dataset Availability: The major concern can be to find a suitable dataset to carry out this project.
- Unavailability of sufficient data from online sources. This is due to limited amount of research done in the field.
- Time complexity works differently for different languages
- Analysing time complexity for built in libraries needs a different approach

Assumptions:

- The code written needs to meet the syntax of the language and thus free from compile time and runtime errors

Dependencies:

- A few dependencies prevail in our system:
- The model class depends on the input language that is identified by the language class.
- Also, the size of the input program is proportional to the time taken to classify.
- The current system works for code snippets of three languages:
 - o C
 - o Python
 - o Java
- The file size is limited to 50mb.
- One of the main assumptions is that the user's code is syntactically correct, that is, it is error free.

6.4 High Level Design

Frontend Module:

- The user will interact with the application either through a terminal or a web based interface (if we get sufficient time to implement this)
- A basic help manual will be available to the user consisting of instructions on how to use the application.

Backend Module:

- This will consist of the ML model and the server (if website is also implemented)

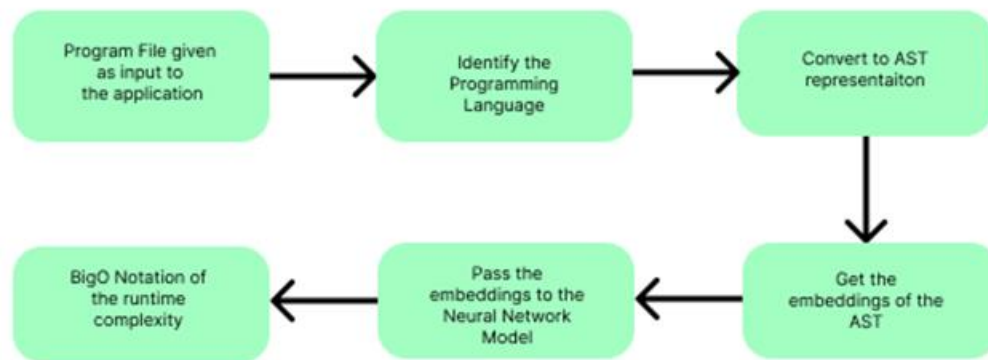


Fig 9: Data Flow Diagram

1. Conceptual or logical – A systematic mathematically formulated evaluation will be carried out based on the CFG's of the programs
2. Process - The basic flow diagram is given above.
3. Physical – The deployment will mostly be as a standalone application, and if we have sufficient time, we will try to implement a web interface.
4. Module – The code will be presented in a modular fashion. Management can be slightly tricky; hence we want to use GitHub for code management. It will also help make the process of installing builds and versions easier.
5. Security – It is of utmost importance to ensure the security of our users. Thus, in order to maintain privacy and security, the system features abide by the Data security act of India

6.5 Design Description

6.5.1 Master Class Diagram

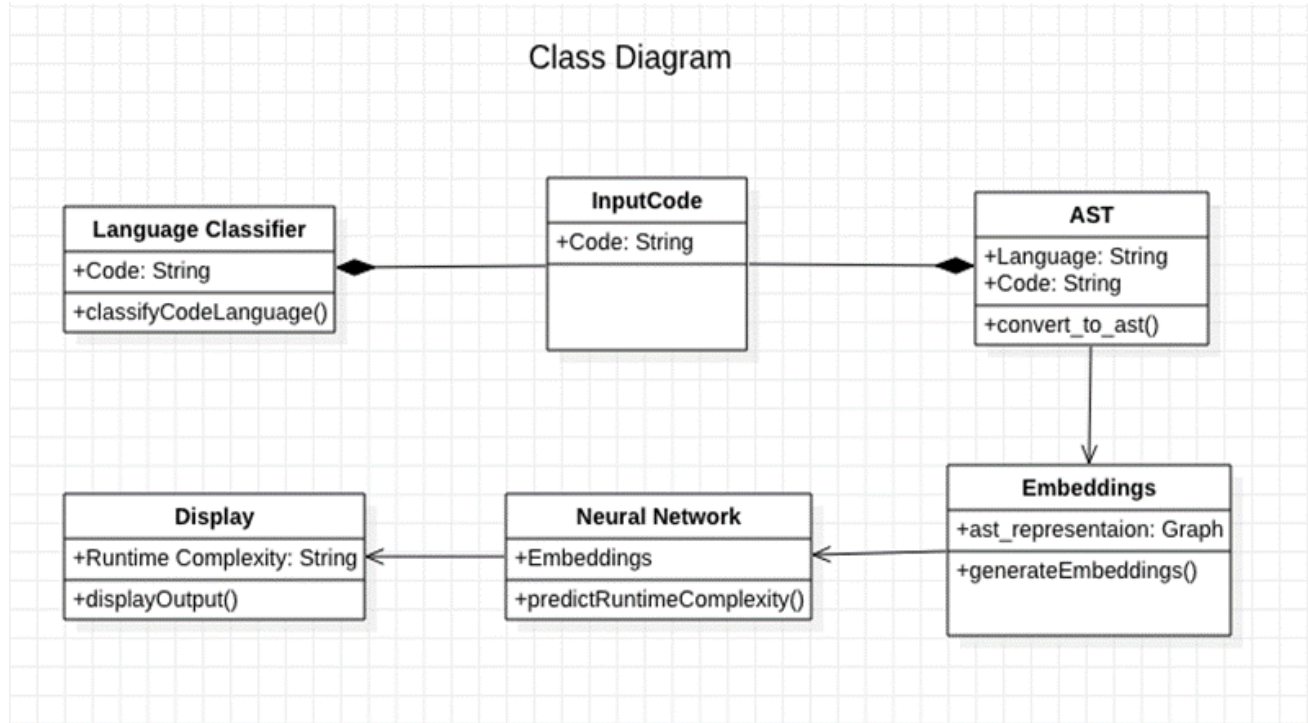


Fig 10: Master class diagram

The Input Code is a component of both the language classifier class and the abstract syntax tree class. Both these classes need the input code as one of the main components, hence we use composition to represent the relationship between the classes.

The language classifier class takes in the code as the input and gives the programming language of the code as the output. The AST class uses both the code and the input language. It converts the code to the abstract syntax tree representation depending on the programming language, since AST representations are not generic and depend on the programming language.

The Embeddings class takes care of generating the embeddings from the obtained AST representation of the code. This is done by using various kinds of algorithms.

Then a Neural Network is used to train the classification model. The embeddings are the vector representation of the code and are passed to the neural network.

The Neural Network takes in the input as the embeddings and predicts the runtime complexity as the Big-O notation.

6.6 User Interface Diagrams

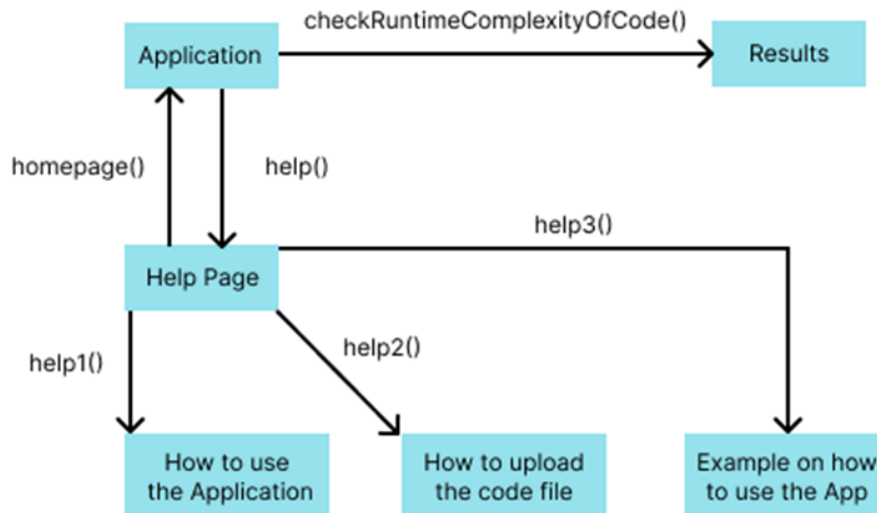


Fig 11: User interface diagram

- The different screens that will be available are:
- Welcome Page / Home Screen, Help Page, Results Screen
- Users will be able to move from one screen to every other screen at any given time.

6.7 Sequence Diagram

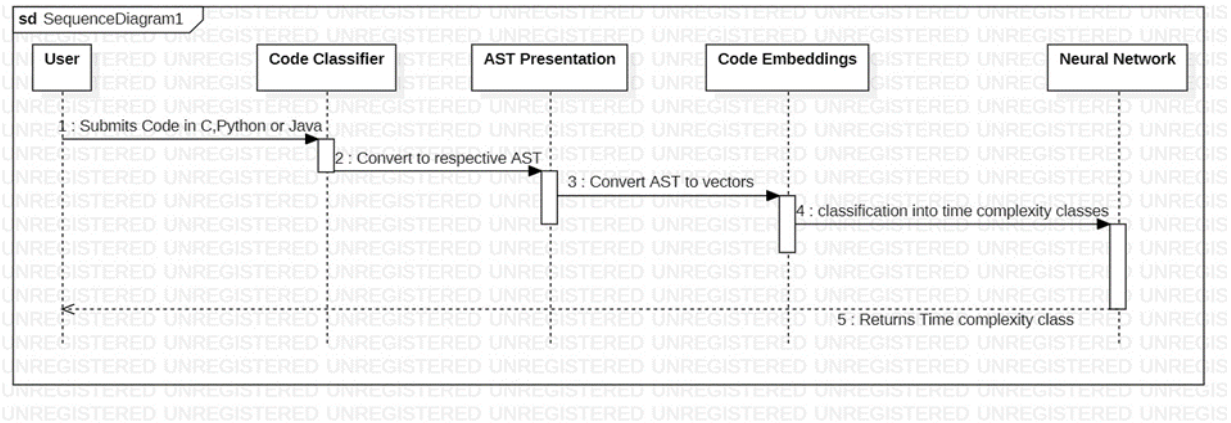


Fig 12: Sequence Diagram

- The User first submits their code in High Level Languages such as C, Python or Java to the system.
- The Code language classifier detects the coding language.
- The code is then passed on to construct the AST.
- The AST generated will be passed to the neural network as input to generate the code embeddings or vectors.
- The code embeddings or vectors will be passed to the Neural network to classify the Code to which class of Big-O the code belongs to.

6.8 Report Layouts

Report Number	Report Name	Purpose	Description (References)
1	Literature Survey	To gain a better understanding of the current research on a specific subject or field of study.	Papers: Current Document
2	Dataset	A Dataset of code files in different languages	Project CodeNet
3	System Requirements specification	The intended purpose, specifications, and design of the software to be created are all described in an SRS document.	SRS: Current Document
4	System Design	To provide enough detailed relevant data about the system and its related functions to allow for the implementation of architectural entities as specified in the system architecture's models and views.	Current Document
5	Current implementation	The code that we plan to work on.	Pseudocode: Current Document
6	Current approach	To get the expected results.	Using embeddings from ASTs and passing it to a BiLSTM network

Table 11: Report Layouts

6.9 Design Details

6.9.1 Novelty

- Predicting run time complexity of programs so far has been done in the traditional way by running the program over and over again by using varied sizes of inputs, plotting the graph of execution time vs input size and hence concluding the time complexity.
- Other methods include manual evaluation of code to predict run time complexity that can be tedious and hard and time consuming for larger programs or more complex programs that involve some inbuilt libraries.
- Our method is a novel way that uses machine learning models to predict code run time complexity of programming languages and we are looking forward to extending it to some common standard libraries, without running the program or using static code analysis that has been previously done before.

6.9.2 Innovativeness

- The implementation can be done in many ways and the final implementation will be finalized soon.
- The source code in programming languages such as C, Python and Java would be first pre-processed, converted to an AST and would be extracted block wise.
- In addition to this, we would be using CFG to find out the control flow of the program.
- The code can be extracted using libraries such as word to vec, code to vec etc to further process the source code in terms of vectors that would be passed to ML algorithms that would predict the code run time complexity.

6.9.3 Interoperability

- At the moment we are going to use Python as our coding language for the project to predict run time complexities for C, Python and Java.

- This is being planned to be implemented on Windows 10 OS and would be made to work on other OS's.

6.9.4 Performance

The project will be used to predict approximate code run time complexity based on the classes of Big-O notations we use. This will be compared with the code run time complexities obtained by other methods to compute the accuracy of our project.

6.9.5 Security

The software being developed does not require any security since it is going to be a tool to predict run time complexity.

6.9.6 Reliability

Sufficient testing will be done to ensure that the reliability of the system is not compromised.

6.9.7 Maintainability

- The project can be extended and hosted on a cloud platform and accessed by a considerable number of users. Hence maintenance of a cloud platform is managed by the service provider and the code monitored and maintained easily on the cloud platform where bug fixing, versioning can be done easily.
- The maintenance can also be done since it is going to be developed using GitHub.

6.9.8 Portability

We are planning to run our project on local systems currently and this can be extended to making a website or a plugin or even host it on a cloud platform to increase the portability of the code.

6.9.9 Legacy to modernization

It is easily portable if the project is made to be hosted on the cloud or as a website that can be accessed on any device anywhere.

6.9.10 Reusability

The code will be reusable to some extent across the 3 programming languages that is C, Python and Java with respect to AST generation, DAG and CFG code generation.

6.9.11 Resource utilization, Etc.,

- Python as the programming language to carry out the entire implementation, since all of us are familiar with python and a large variety of libraries are available.
- TensorFlow to implement the neural network. It supports a lot of libraries for the easy implementation of neural networks.
- Google colab for training and testing of the model, since the platform supports collaboration amongst teammates and provides support for GPU and various other resources.

6.10 Help

The users will be provided a downloadable help manual to use the application. The help manual creation is in progress and will be updated with the updating of the project.

We will try to include a demo video on the usage of the application if possible.

CHAPTER 7

IMPLEMENTATION AND PSEUDO CODE

PseudoCode:

```
class Input_Code:
```

```
    function input_code():
```

```
        return input_code
```

```
class Language:
```

```
    function identify_language(input_code):
```

```
        import required package/library
```

```
        language = Programming language of the code_snippet returned by the  
        library
```

```
        return language
```

```
class AST:
```

```
    language = Language.identify_language(input_code)
```

```
    function generate_AST(input_code, language):
```

```
        load AST library of the language
```

```
code_ast = AST representation of the code given language
```

```
return code_ast
```

```
class Embedding:
```

```
    function generate_embedding(code_ast):
```

```
        load embedding library
```

```
        embedding_ast = embedding of the AST
```

```
        return embedding_ast
```

```
class Neural_Network:
```

```
    function build_neural_net():
```

```
        #labels = target label of the given code snippet
```

```
        import required neural network libraries
```

```
        build model
```

```
        return model
```

```
class TrainModel:
```

```
    #each folder code file is traversed and each code file is passed through the above functions
```

```
    language = Lang.identify_language(input_code)
```

```
    ast = AST.generate_ast(input_code,language)
```

```
    embedding = Embedding.generate_embedding(ast)
```

```
    model = Neural_Network.build_model()
```

```
file_path = file path to save the model weights
```

```
function train_model(model, embeddings, labels):
```

```
    model.train(embeddings, labels)
```

```
    model.save(file_path)
```

```
    return file_path
```

```
class Ouput_label:
```

```
    model = load_model(file_path)
```

```
    function output_label(input_code):
```

```
        language = Lang.identify_language(input_code)
```

```
        ast = AST.generate_ast(input_code, language)
```

```
        embedding = Embedding.generate_embedding(ast)
```

```
        output = model.predict(embedding)
```

```
        return output
```

A description of the pre-processing and neural network modules that can be used:

Potential Algorithms to get the embeddings are as follows:

- Word2vec:

Word2Vec creates a vector of words. This is a distributed numerical representation of the characteristics of a word. The characteristics of these words can consist of words that represent the

context of each word in the vocabulary. Finally, word embedding helps establish an association with another word that has a similar meaning to one word through the created vector.

- **Code2Vec:**

Code2vec is a neural model that learns analogies related to source code. The model was trained in a Java code database but can be applied to any code base. To do this, break the code into a collection of paths in the abstract syntax tree and learn the atomic representation of each path. Code embedding is trained on large external datasets, so it can bring more knowledge about the semantic and syntactic meaning of code tokens.

- **ASTToken2Vec:**

A new embedding method (vector representation) for AST nodes. This method is called ASTToken2Vec and, like Word2Vec, trains the neural network by using contextual information to get a vector representation of the AST node. It consists of a four-layer neural network to encode the contextual information of the AST and generate a semantics-based embedded representation vector that hides more knowledge behind the AST.

- **Deep Walk algorithm:**

This algorithm uses random walk technique to traverse through the neighbouring nodes to aggregate the features and add it to the current node. It then uses either Word2vec or SkipGram Models to convert the aggregated data to vectors. These models are trained and then the embeddings are obtained from these trained models.

- **Node2Vec**

Node2Vec works on the same principle as Deep Walk Algorithm but with one difference. This algorithm uses either Breadth First Traversal (BFS) or Depth First Traversal (DFS) to aggregate information from neighbouring nodes. After the required information is aggregated, the same algorithms as Deep Walk, wither SkipGram or Word2Vec for conversion to vector representation.

- Graph2Vec

This algorithm considers both edges and nodes rather than just nodes like the two previous algorithms. It works by considering subgraphs of the graph. The embeddings are extracted in an unsupervised manner and are task independent. Hence, these obtained embeddings can be used for a variety of tasks.

The following is a brief description of the Neural network that we would be using to build the classification model. BiLSTM is being used as the neural network. This is used when the prediction depends not only on the previous input but also on the future input. Let's say “I like to sing _”. Here, the song does not matter. Therefore, prediction depends on understanding the future along with the previous one. This is the role of BiLSTM. This solves the prediction problem between fixed sequences. The Vanilla RNNs have the limitation that both the input and the output are the same size. Bidirectional LSTMs cannot be used if the entire sequence cannot be waited for before inference begins. It's like a continuous translation.

CHAPTER 8

CONCLUSION OF CAPSTONE PROJECT PHASE - 1

Extensive literature survey has been carried out to understand how the code can be fed to the neural network for the prediction of runtime complexity. We have learnt a lot about AST's, CFG's, GNN's, Bi-LSTM's, the procedure to produce quality code embeddings, etc, and have found potential novel solutions. We have also identified the tools and libraries as well as the technologies which we will be using to help us to build our ML model. High level design document covers the overall application architecture including the data flow diagram, master class diagram, ER-diagram, and interface diagram. The dataset (CodeNet) we are using covers all kinds of programs from basic to advanced algorithms thereby helping our model to perform better.

CHAPTER 9

PLAN OF WORK FOR CAPSTONE PROJECT PHASE - 2

- Labelling the dataset
- Generating ASTs for code snippets as a part of preprocessing
- Classifying the language of the input code
- Generating Code embeddings
- Building a neural network
- Train and test the model
- Validate the performance using various online tools
- Write a research paper for the project

REFERENCES

- [1] Sikka, Jagriti & Satya, Kushal & Singla, Yaman & Uppal, Shagun & Shah, Rajiv Ratn & Zimmermann, Roger. (2020). Learning Based Methods for Code Runtime Complexity Prediction. 10.1007/978-3-030-45439-5_21.
- [2] Buch, Lutz & Andrzejak, Artur. (2019). Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection. 95-104. 10.1109/SANER.2019.8668039.
- [3] P. Haridas, G. Chennupati, N. Santhi, P. Romero and S. Eidenbenz, "Code Characterization With Graph Convolutions and Capsule Networks," in IEEE Access, vol. 8, pp. 136307-136315, 2020, doi: 10.1109/ACCESS.2020.3011909.
- [4] Gao, Yanjie and Gu, Xianyu and Zhang, Hongyu and Lin, Haoxiang and Yang, Mao (2021) "Runtime Performance Prediction for Deep Learning Models with Graph Neural Network" /MSR-TR-2021-3/Microsoft.
- [5] Q. Feng, C. Feng and W. Hong, "Graph Neural Network-based Vulnerability Predication," 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2020, pp. 800-801, doi: 10.1109/ICSME46990.2020.00096.
- [6] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang and Xudong Liu , "A Novel Neural Source Code Representation Based on Abstract Syntax Tree", 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE) 1558-1225/19/\$31.00 ©2019 IEEE DOI 10.1109/ICSE.2019.00086.

[7] Haridas, Poornima & Chennupati, Gopinath & Santhi, Nandakishore & Romero, Phillip & Eidenbenz, Stephan. (2020). Code Characterization With Graph Convolutions and Capsule Networks. IEEE Access. PP. 1-1. 10.1109/ACCESS.2020.3011909.

[8] Lin, Chen & Ouyang, Zhichao & Zhuang, Junqing & Chen, Jianqiang & Li, Hui & Wu, Rongxin. (2021). “Improving Code Summarization with Block-wise Abstract Syntax Tree Splitting”. 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC) | 978-1-6654-1403-6/20/\$31.00 ©2021 IEEE | DOI: 10.1109/ICPC52881.2021.00026.

[9] Reza, Sayed Mohsin & Rahman, Md & Parvez, Md & Badreddin, Omar & Al Mamun, Shamim. (2020). “Performance Analysis of Machine Learning Approaches in Software Complexity Prediction.” 10.1007/978-981-33-4673-4_3.

[10] Tomczak, Jakub & Lepert, Romain & Wiggers, Auke. (2019), Qualcomm AI Research, Qualcomm Technologies Netherlands B.V. “Simulating Execution Time of Tensor Programs using Graph Neural Networks.” arXiv:1904.11876v3.

[11] Wang, Yanlin & Li, Hui. (2021), Microsoft Research Asia, School of Informatics, Xiamen University. “Code completion by modeling flattened abstract syntax trees as graphs.”

[12] Puri, Ruchir and Kung, David S. and Janssen, Geert and Zhang, Wei and Domeniconi, Giacomo and Zolotov, Vladimir and Dolby, Julian and Chen, Jie and Choudhury, Mihir and Decker, Lindsey and Thost, Veronika and Buratti, Luca and Pujar, Saurabh and Ramji, Shyam and Finkler, Ulrich and Malaika, Susan and Reiss, Frederick, "CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks", in arXiv, 2021, doi: 10.48550/ARXIV.2105.12655.

[13] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. DeepWukong: “Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network.” ACM Trans. Softw. Eng. Methodol. 30, 3, Article 38 (July 2021), 33 pages. <https://doi.org/10.1145/3436877> .

[14] Long Chen, Wei Ye and Shikun Zhang (2019). “Capturing Source Code Semantics via Tree-based Convolution over API-enhanced AST”. CF ’19, April 30-May 2, 2019, Alghero, Italy © 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.ACM ISBN 978-1-4503-6685-4/19/05 <https://doi.org/10.1145/3310273.3321560> .

[15] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, Jeronimo Castrillon. (2020) “Compiler-Based Graph Representations for Deep Learning Models of Code”. Proceedings of the 29th International Conference on Compiler Construction (CC ’20), February 22-23,2020, San Diego, CA, USA. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3377555.3377894> .

[16] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, Zhi Jin. (2020). “Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree”. SANER 2020, London, ON, Canada 978-1-7281-5143-4/20/© 2020 IEEE.

Appendix A: Definitions, Acronyms and Abbreviations

Acronyms and Abbreviations:

1. AST - Abstract Syntax Trees
2. GNN - Graph Neural Network
3. RNN - Recurrent Neural Network
4. LSTM - Long Short-Term Memory
5. BiLSTM - Bidirectional LSTM
6. CFG - Control Flow Graphs
7. DAG - Directed Acyclic Graph

Definitions:

1. **TLE**: Time Limit Exceeded – This is a situation when the written code takes longer to run the specified time limit.
2. **GNN**: Graph Neural Network – A type of Neural Network associated with Graphs and Graph Embeddings.
3. **AST**: Abstract Syntax Tree – The representation of the important features of a code snippet in the form of a tree.
4. **CoRCoD**: Code Runtime Complexity Dataset – The Dataset of Java Code Snippets used to train the model in the first paper mentioned in the literature survey.
5. **CFG**: Control Flow Graphs – Another representation of a given code snippet and its semantics