Faculty of Natural and Mathematical Sciences
Department of Informatics

King's College London,
Strand Campus, London,
United Kingdom

7CCSMPRJ
Individual Project Submission 2021/22

Name: Olubusayo Akeredolu
Student Number: 20107125
Degree Programme: MSc Artificial Intelligence
Project Title: Text vs Trees vs Graphs: Deep Learning Techniques for Program Understanding
Supervisor: Dr Maria Polukarov

RELEASE OF PROJECT

Following the submission of your project, the Department would like to make it publicly available via the library electronic resources. You will retain copyright of the project.

☒ I **agree** to the release of my project

☐ I **do not** agree to the release of my project

Signature:

Date:

Department of Informatics
King's College London
United Kingdom

7CCSMPRJ MSc Project

TEXT VS TREES VS GRAPHS: DEEP LEARNING TECHNIQUES FOR PROGRAM UNDERSTANDING

Name: Olubusayo Akeredolu
Student Number: 20107125
Degree Programme: MSc Artificial Intelligence

Supervisor's Name: Dr Maria Polukarov

This dissertation is submitted for the degree of MSc in Artificial Intelligence

ACKNOWLEDGEMENT

I thank my parents and siblings for their support throughout this project. I also want to thank Dr Maria Polukarov for her guidance on this project. Finally, I want to thank the Stack Overflow community for existing.

ABSTRACT

Programming Language Understanding (PLU) is a field of Artificial Intelligence and Machine Learning that seeks to find the most suitable Deep Learning (DL) models for carrying out learning tasks on source code. The emphasis here is on DL models because other non-DL machine learning models can be applied to understand programs. This project takes this core objective of PLU and aims to compare the performance of mainstream DL models to the performance of non-DL machine learning models using three different data structures. These data structures are text, trees and graphs. This research project aims to determine which DL models are the most suitable for understanding source code and which of these three data structures is the most ideal for the task. This project also goes beyond PLU using deep learning models. It looks at the performances of select non-DL models to see which model and data structure is best suited for learning tasks on source code overall.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

1.1 Project Overview

This project is concerned with Programming Language Understanding (PLU) using deep learning (DL) and machine learning (ML) techniques. PLU is a branch of deep learning that aims to use various deep learning techniques to train neural networks (NNs) to understand programs called source code.

PLU, while relatively new, is a significant branch of Artificial Intelligence (AI) and Machine Learning (ML). This is because it deals with one specific way in which NNs are yet to catch up with humans (or programmers and software developers, in this case); they are unable to tell the difference between normal human language text (HLT) and source code, regardless of the programming language. For example, take a scenario where a text processing NN is built using source code as its training data. This text-based model will treat the input the same way an extract from a novel or a piece of text would. Unfortunately, this treatment of source code in the same way as text will lead to the loss of critical information due to the many differences between HLT and source code.

One of these differences, and the most important of them all, is that every human language has a finite grammar – a list of all the words available for use in the language. On the other hand, the source code is unlimited in its grammar because the names of the elements in the program are entirely up to the programmer.

Another critical distinction between source code and HLT is that single sentences in HLT do not usually need additional context to be understood. For example, the English language sentence "The woman walked into the room" follows the grammatical rules of the English language and can be understood entirely on its own. Source code is not like this. An example of a Python program line is "c = a + b". This is meaningless if the rest of the program is not considered. This line of code requires the programmer – and the compiler or interpreter – to consider where the variables 'a', 'b' and 'c' were initially declared. Additionally, in a strongly typed language [6], the data types of each variable will have to be considered to ensure that no errors or exceptions are raised during execution and that they are all compatible with the '=' and '+' operations. The programmer will also have to examine how the variables change during the program's execution.

A third way in which programs and HLT differ is that programs contain many conceptual and structural information that HLT does not possess. These concepts include conditions and control flow, inheritance and objects in Object Oriented Programming (OOP) languages, abstract classes, abstraction, etc. Examples of structural information in programs but not in HLT include the programming paradigm (imperative, event-driven or declarative languages), typing structure (weakly typed or strongly typed languages), and the use of features from imported classes and external libraries, etc.

It is imperative to note that many different types of programming languages exist. These include non-scripting languages [1] (procedural languages, object-oriented languages, functional languages, etc.) and scripting languages [2] (server-side scripting languages, client-side scripting languages and query languages). It is also necessary to note that specific programming languages, e.g., Python, are considered both scripting and non-scripting languages.

The main difference between these two programming languages is that scripting languages are generally interpreted (processed line by line at execution), and non-scripting languages are usually compiled (processed as one at execution). The important difference between these two types of languages that needs to be considered for this project is that scripting languages tend to be closer in syntax to HLT than non-scripting languages. For example, the SQL (query language) statement “SELECT name FROM myTable” follows the rules of an English language sentence – it contains an object, a verb, and a subject – and can be understood without further context.

For this reason, this project focuses solely on non-scripting languages, specifically, the Python programming language in a Procedural programming [3] context.

1.2 Aims and Objectives

This project’s main objective is to compare ML models based on three different information storage structures to determine which data structures are the most suitable for designing NNs to carry out classification tasks on source code. Furthermore, to get the most accurate picture of how generic NN models work when trained and tested on source code, this project will compare the performances of 9 different Natural Language Processing (NLP) [7] classification models. 5 of these models are deep learning models, while the other 4 are simple classification models. These have been included in this project to observe and compare the behaviour of non-deep-learning models when trained and tested with source code.

I have chosen these models because they are conventionally used when carrying out NLP tasks. The deep learning models are the Long Short-Term Memory (LSTM) network, the Simple Recurrent Neural Network (SRNN), and the Gated Recurrent Unit (GRU) model. All Recurrent Neural Network (RNN) models and the Multi-Layer Perceptron (MLP) and the Dense model. The non-deep-learning (NDL) models are the Gaussian Naïve Bayes classifier, the Stochastic Gradient Descent (SGD) classifier, the Random Forest (RF) classifier and the Support Vector Machine (SVM) classifier.

1.2.1 Information Storage Structures

The first information storage structure I will be considering is text. As indicated above, multiple existing models for carrying out Natural Language Processing (NLP) on text exist. This reference shows how text-based NNs work when trained and tested on source code.

The second information storage structure I have chosen is the Tree data structure [4]. I have chosen this abstract data type because every program has an Abstract Syntax Tree (AST) representation, which shows the program's structure and connections. This project involves the development of the NLP models named above in such a way that they take data extracted from a program's AST as their input.

The final information storage structure is the graph data structure. This is also an abstract data type. I have chosen graphs because every program, no matter how large or small, can be represented as a flow chart or flow graph, which are forms of directed graphs. Like the tree-based models, this project aims to develop several NNs based on those described above, and each graph-based model will take data developed from a NetworkX graph as its input.

I have chosen to develop models based on trees and graphs because of the differences between text and HLT as outlined in section 1.1 and because of the shortcomings of text-based models when processing source code. In addition, the advantages of trees and graphs over text for capturing structural information are another reason for this decision.

The major shortcoming of text-based models is that they convert each word or a series of words (known as n-grams, where n is the number of words) into a vector or 'token' when carrying out learning tasks. This is an issue when training a neural network on programs because program grammars are unlimited, as outlined in section 1.1. Therefore, different programs that solve the same problem in the same way might use completely different names to label variables, functions, classes, etc. For example, in *fig 1.1a* and *fig 1.1b*, both functions pictured here implement the bubble sort algorithm similarly. The only difference is that they have their variables and functions named differently. This difference means that to a neural network trained on text vectors, they are two completely different pieces of text. Alternatively, both programs have the exact flowgraph representations when represented by a tree or flowgraph, as in *fig 1.2a* and *1.2b*.

fig 1.1a: An implementation of the Bubble Sort algorithm

```
def bubble_sort(vector):
    for i in range(len(vector)):
        for j in range(len(vector) - 1, i, -1):
            if vector [j - 1] > vector[j]:
                vector [j - 1], vector[j] = vector[j], vector [j - 1]
```

fig 1.1b: Another implementation of the Bubble Sort algorithm

```
def sort(array):
    for a in range (0, len(array)):
        for b in range(len(array)-1, a, -1):
            if array[b-1] > array[b]:
                array[b-1] = array[b]
                array[b] = array[b-1]
```

fig 1.2a: Flowgraph representation of code snippet from fig 1.1a

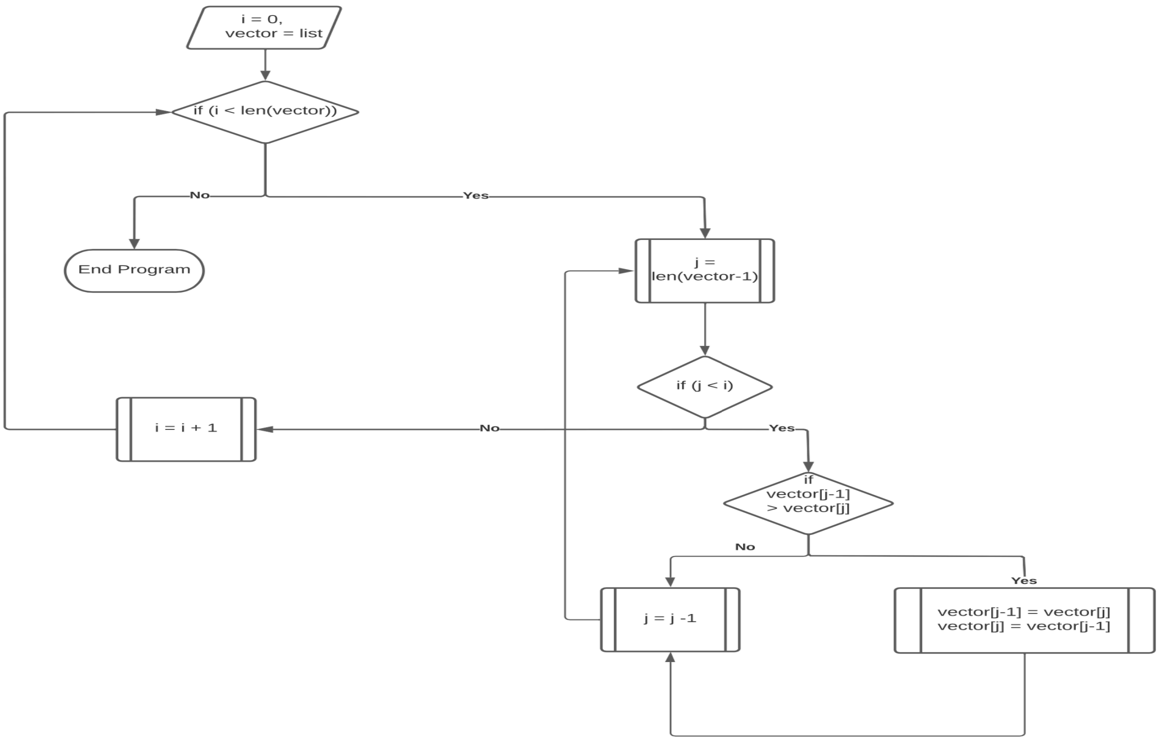
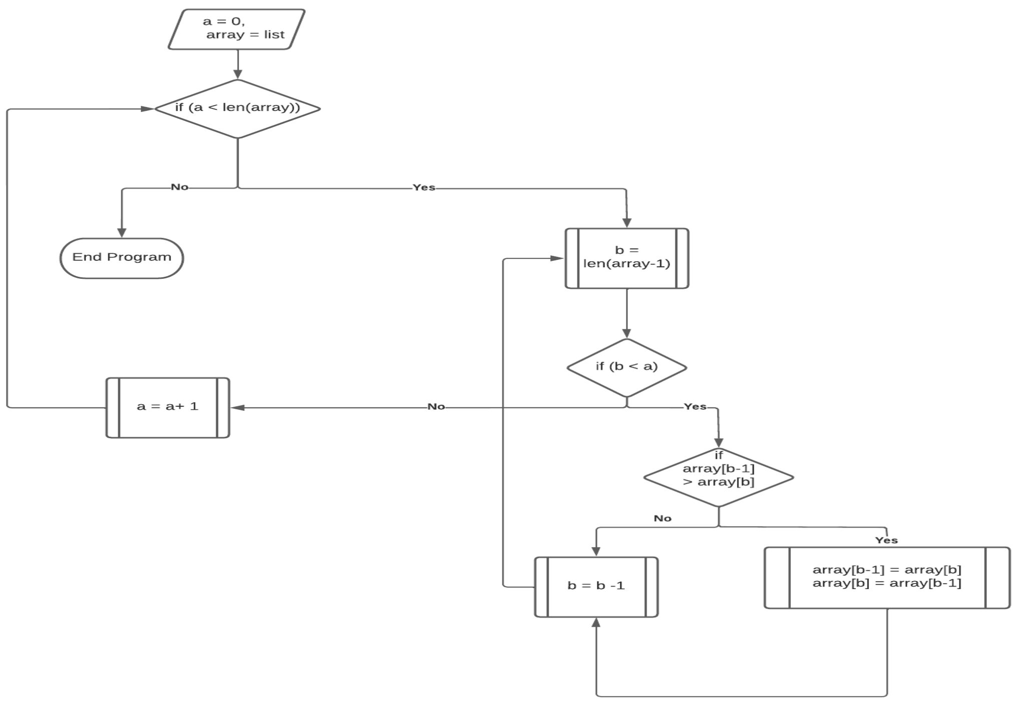


fig 1.2b: Flowgraph representation of code snippet from fig 1.1b



1.2.2 Models For Classification

The models this project aims to develop for both the graph and tree-based models are listed below:

i. Deep Learning (DL) Models:
   a. LSTM RNN classification model.
   b. GRU RNN classification model.
   c. Simple RNN classification model.
   d. MLP classification model.
   e. Dense classification model

ii. Non-Deep-Learning (NDL) Models:
   a. Gaussian Naïve Bayes (GNB) classifier.
   b. Stochastic Gradient Descent (SGD) classifier.
   c. Random Forest (RF) classifier.
   d. Support Vector Machine (SVM) classifier.

1.2.3 Classification Task

The classification task in this project is a binary classification problem to classify programs based on the sorting algorithm they implement. The applicable sorting algorithms here are the Merge Sort and the Quick Sort. I have chosen these two algorithms because they are typically longer in implementation code than most other sorting algorithms. This means that there is usually more information to be derived from the AST of a program that implements one of these algorithms.

2 BACKGROUND AND LITERATURE REVIEW

2.1 Programs As Graphs and Trees

It is well known that every program has an abstract syntax tree representation. In theory, this could lead to the assumption that every program has a graph representation or an abstract graph representation, but there has not been enough work done to demonstrate this. The leading driving theory of this project is using a program's abstract graph or tree representation, a neural network can be trained to learn the structure of the code and classify it based on that structure.

2.2 Underlying Theory and Background

The underlying theory behind this project is this; using the right design and model specifications, a neural network can be trained to carry out classification tasks on source code, resulting in a suitable level of accuracy.

This project has its background rooted in the fundamentals of the Graph Neural Network (GNN). This network architecture, as described in [14], presents a method for processing information stored in the form of a graph using a specially designed NN. There have been several developments on this model. These include the GGNN, the Gated Graph Sequence Neural Network (GGSNN) [15], the Gated Graph Convolutional Neural Network (GGCNN) [16], the Gated Graph Recurrent Neural Network (GGRNN) [17], etc.

Another field that serves as the background of this project is Natural Language Processing (NLP) [7]. NLP is a field of AI and ML with a lot of pre-existing work done to understand how NLP models process HLT. The shortcoming of NLP models, as described in section 1.2, is their use of tokens based on individual words or a series of words. This vector representation is unsuitable for source code tasks because of the lack of limits on the grammar of source code. This means that if different people wrote all the files in a training dataset to solve the same problem, the dataset could end up having very few words, or tokens, in common, leading to underfitting in the model. Underfitting is where the model does not accurately learn the training data and cannot be applied to generalize unseen data. It results in poor performance of the model. NLP tokens are also unsuitable for PLU due to the existence of programming concepts such as loops. In HLT, a loop is the equivalent of saying the same thing repeatedly, leading to an increase in the number of tokens corresponding to that word. In source code, however, a conditional loop is used, eliminating the need to repeat the statement. This results in the structural information the loop communicates being lost.

Finally, this project has its background in the study of the tree data structure. I believe that there has not been enough work to examine how these data structures can be used to represent the information contained in programs. This project will demonstrate the suitability of graph and tree-based models for understanding and processing the complex information contained in source code.

2.3 Literature Review

Although PLU as a branch of AI and ML is relatively new, there is some interesting pre-existing literature in this field. Some exciting work has been done in building NNs specifically for processing and classifying programs. One of such is [5], where the researchers define a method for converting source code to graph representations based on the program's AST. To the best of my knowledge, this paper is the first of its kind, which makes it very relevant to the work done in this project. This paper uses the Gated Graph Neural Network (GGNN) [8] to carry out two learning tasks on source code. The first is the 'VARNAMING' task which is designed to predict what a variable should be called based on how it is used in a program. The second learning task is the 'VARMISUSE' task, in which the model predicts whether a variable has been used correctly or not.

On the VARNAMING task, they receive an accuracy score of 53.6%, while on the VARMISUSE task, they receive a score of 85.5%. The results of the VARNAMING task indicate that the model used in this study might need modifications to its design and parameters to produce more robust results. On the other hand, the results of the VARMISUSE task indicate that the GGNN model built in this study can accurately tell when a variable has been misused.

These results are a motivation for this project. They indicate that NNs can be used to process and understand source code. Still, the model must be specifically designed to handle data types, such as graphs, that accurately represent the complex information contained in the source code.

Another interesting study is [9]. This paper presents a model for processing source code by introducing the Tree-Based Convolutional Neural Network (TBCNN). The method outlined in this paper uses a process the researchers refer to as one-way pooling and convolution to process a vector representation of a program's AST. To the best of my knowledge, it is also the first of its kind, making it a motivation for this project.

The most interesting aspect of this paper is its use of convolution. Convolution is commonly used when processing images [10, 11] and the researchers admit that Convolutional Neural Networks (CNNs) do not accurately represent tree-based information. The results from this study are a classification accuracy score of 94% on a task to classify programs based on functionality. This reinforces the notion that with the right techniques, conventional NN models that are usually unsuitable for working with source code can be utilised and structured in such a way that they produce good results when trained and tested on source code.

Additionally, some interesting work has been done on training NNs to detect syntax errors in programs [11] and on converting code to embeddings suitable for processing by a neural network model [12].

The main point that sets this project apart from the studies described above is that this project defines a new way of accurately creating embeddings for processing source code. To the best of my knowledge, this project is also the first to compare the performances of graph-based and tree-based models on the same tasks to determine which method is most suitable for programming language processing.

The primary motivation for this project is the shortage of pre-existing literature in the field of PLU. In addition, I am interested in exploring how graphs and trees work when processed by standard NLP models compared to how they work when the data is stored as text. Finally, I am also interested in observing the differences in the behaviour of DL and NDL models when they are trained and tested on source code.

3 Objectives, Specifications and Design

3.1 Specific Project Objectives

The main objective of this project has been outlined in section 1.2 (comparing the accuracy scores of tree-based models, graph-based models, and text-based models to assess which is most suitable for carrying out classification tasks on programs). This objective has been broken down into eight specific parts, each outlining a core objective of the project.

3.1.1 Text-Based Objectives

Objective 1: To implement a series of text-based DL and NDL models (section 1.2.1) to compare their results when trained and tested on source code.

3.1.2 Graph-Based Objectives

Objective 2: To implement a method for converting source code into a directed graph containing nodes and edges.

Objective 3: To implement a method for converting the results of Objective 2 into vector embeddings that a neural network can process.

Objective 4: To train and test the models built in Objective 1 (section 1.2.1) for carrying out classification and learning tasks on the results of Objective 3.

3.1.3 Tree-Based Objectives

Objective 5: This is to design a method for converting a program into an AST and extracting the important information from the AST to create a tree data structure to represent the program.

Objective 6: Following on from Objective 5, this is to design a method for converting the results of Objective 5 into vector embeddings that a neural network can process.

Objective 7: This objective seeks to train and test the NDL classifiers and DL models from Objective 1 (section 1.2.1) to carry out classification tasks on the results derived from Objective 6.

3.1.4 Final Objective

Objective 8: The final objective is to compare the results received from Objectives 1, 4 and 7 to assess which data structure is the most suitable for the classification task (section 1.2.2).

3.2 Technical Specifications

The models for this project have been built using the Python programming language and a range of Python-specific deep learning and machine learning libraries; Keras, TensorFlow, NumPy, Scikit Learn, Matplotlib and NetworkX. In addition to Python 3.10, these libraries must be installed on any device this project will be executed on. Keras, TensorFlow and Python 3.10 have been used for the majority of the implementation of this project.

The Dense (section 3.3.2) and RNN (LSTM, GRU & SRNN) models (section 3.3.3) have been built using a combination of Keras and TensorFlow. The SGD, RF and SVM models (section 3.3.5) have all been built using built-in functions from the Scikit Learn library. The Gaussian Naïve Bayes Classifier has been built solely using Python 3.10, while the MLP model has been built using a combination of TensorFlow and Python 3.10. Matplotlib displays graphs where necessary, NumPy is used for minor conversions and classifications, and the NetworkX library stores graph data structures.

3.2.1 Dataset

All the datasets used in this project have been collected from GitHub [18, 19]. The dataset consists entirely of Python source code files. The dataset is divided into two groups based on the sorting algorithm they implement. These groups are Merge Sort and Quick Sort.

The Merge Sort dataset comprises 84 different implementations of the Merge Sort algorithm. The Quick Sort group contains 77 implementations of the Quick Sort algorithm. These data groups will be used to train and test each model built as part of this project.

3.3 Design

In total, I have built nine different models to be as thorough as possible in my research. This project seeks to find the most suitable data structure for classifying source code based on the type of sorting algorithm implemented in the program. The models described below are all designed to work on binary classification problems, but they can be easily modified to become multiclass classification problems.

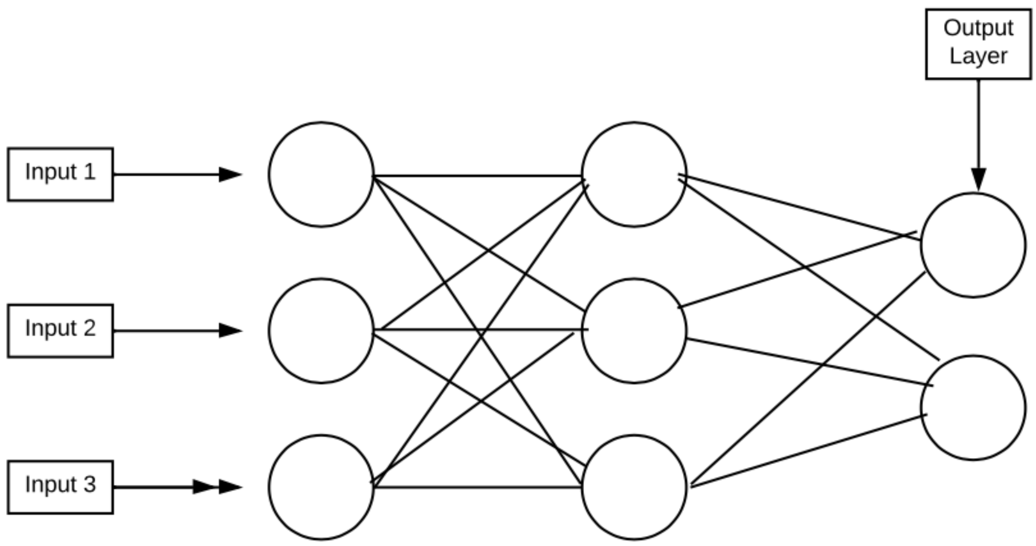
3.3.1 Deep Learning Models

3.3.1.1 Multi-Layer Perceptron (MLP) Network Model

A multilayer perceptron [20] is a type of feed forward [21] artificial neural network (ANN) [23] that consists of multiple layers of perceptrons [22], hence the name. A perceptron is a type of classifier used for binary classification problems. An example of a cross-section of an MLP is shown in *fig 3.1.* I have opted to use the MLP network model for its popularity and use on binary classification tasks.

This model has been constructed using a combination of Python 3.10 and TensorFlow. All the variables in this model, including the model weights and biases, are declared as tensors (TensorFlow variables), and the calculations are all carried out using TensorFlow functions. In addition, this model uses back-propagation [24] (section 4.2.4.1.2), a method used in ANNs to fine-tune the outputs of a neural network model and make them as accurate as possible.

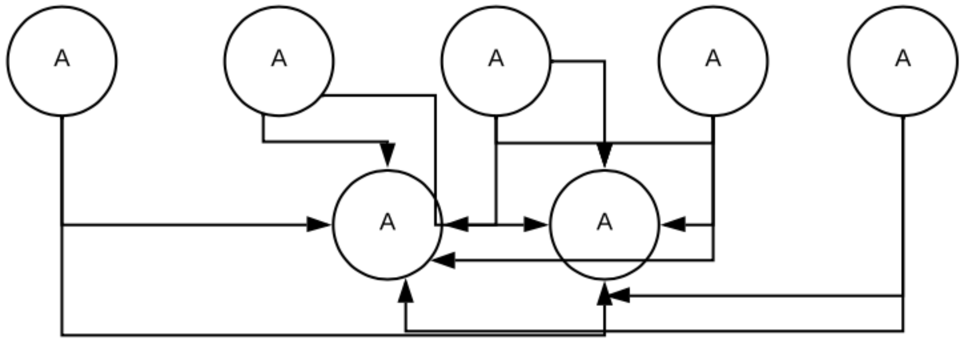
fig 3.1: Simple MLP architecture



3.3.1.2 Simple Dense Network Model

This model has been added to verify the results of the MLP model. It comprises one input layer and three densely connected layers [25]. Densely connected in this context means that all the nodes in the hidden layers connect to every single node in the following layer. An example of this dense structure can be seen in *fig 3.2*. I have chosen this model because densely connected layers are conventionally used in deep learning.

fig 3.2: Cross section of a densely connected neural network



3.3.1.3 Recurrent Neural Network (RNN) Models

3.3.1.3.1 Long Short-Term Memory (LSTM) Network Model

An LSTM is a recurrent neural network developed to solve the vanishing gradient problem [26], an issue that arises in back-propagation models such as the MLP. LSTMs are comprised of four components:

i. Cell
ii. Input gate
iii. Output gate
iv. Forget gate

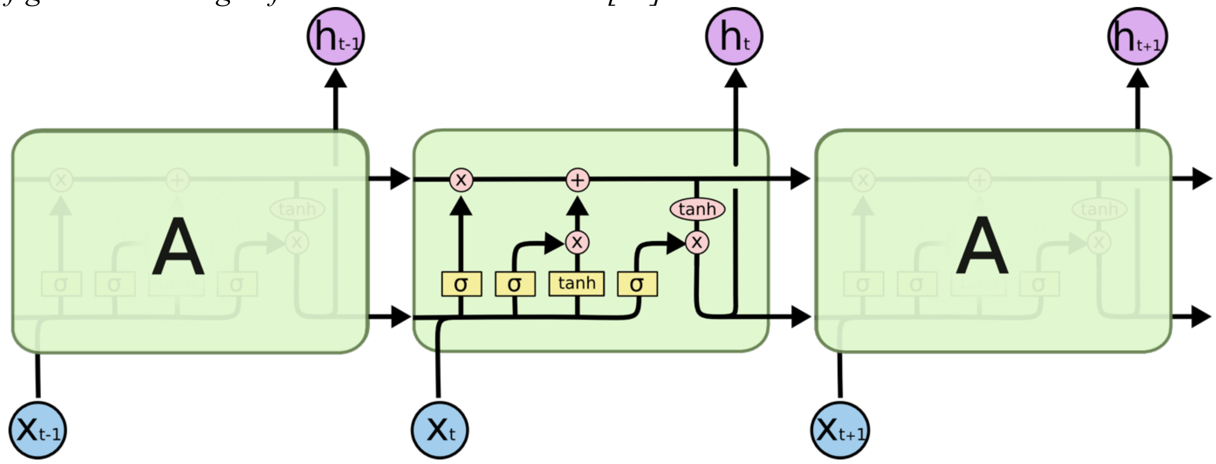
LSTMs can 'remember' data across various time steps using the cell component. This feature is why I have chosen to build this model because it means the model can remember a sequence of data, which is vital in the context of this project. I have built this model entirely in Keras and TensorFlow.

The LSTM model is comprised of an input layer (input_1), two hidden LSTM layers (lstm_1, which is bidirectional, and lstm_2), and a densely connected output layer (dense_1). I have chosen to use a bidirectional layer [27] in lstm_1 because of how Bidirectional LSTMs work. With this layer, the input moves in two directions; input-to-lstm_1 and lstm_1-to-input. This means that the information from both layers is preserved, and the results being passed to lstm_2 are likely to be more accurate.

I have opted not to make lstm_2 bidirectional to prevent overfitting – a situation where a neural network learns the training data too well and cannot perform well on unseen tasks. This layer passes information in one direction (lstm_2-to-dense_1).

Before deciding on this selection of layers, I experimented with different combinations of layers and layer types. Finally, I chose the above because it is the combination that produced the best accuracies of all my experiments. The design of a standard LSTM model can be seen in *fig 3.3*.

fig 3.3: The design of a standard LSTM model [28]

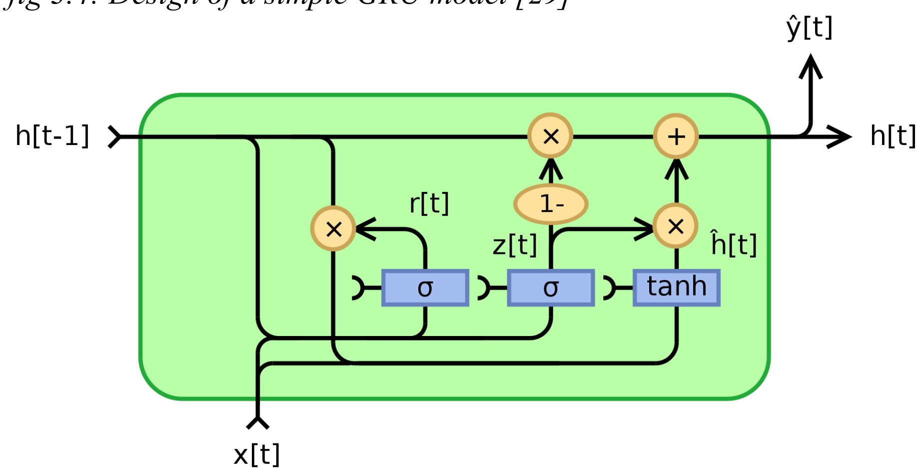


3.3.1.3.2 Gated Recurrent Unit (GRU) Model

A GRU [30] is very similar to an LSTM. The main difference between both models is that GRUs tend to have fewer parameters and no output gate. GRUs also generalise better to smaller datasets, and it is for this reason that I have opted to use this model.

The setup of the GRU model is identical in layers to the LSTM. There is an input layer (input_1), two hidden GRU layers (gru_1, which is bidirectional, and gru_2) and a densely connected output layer (dense_1). An example of the design of a GRU model is shown in *fig 3.4*.

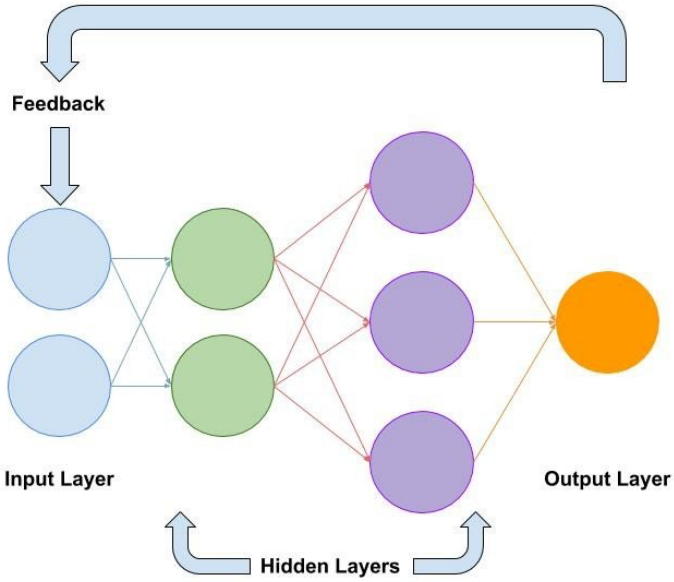
fig 3.4: Design of a simple GRU model [29]



3.3.1.3.3 Simple Recurrent Neural Network (SRNN) Model

I have built the SRNN model because it is the original iteration of the RNN and the simplest of the available RNN model architectures. This model is also identical in structure to the GRU and the LSTM. The input layer (input_1), two hidden SimpleRNN [31] layers (simplernn_1, which is bidirectional, and simplernn_2) and a densely connected output layer (dense_1). A simple RNN architecture is demonstrated in *fig 3.5*.

fig 3.5: Simple RNN architecture [32]



3.3.2 Non-Deep learning Models

3.3.2.1 Naïve Bayes Classifiers

The Naïve Bayes (NB) classifier is a type of simple classifier that carries out classification based on Bayes' theorem [33]. It is based on probabilities.

NB classifiers assume independence of variables, i.e., each value is independent of all the others. They tend to be outperformed by most other models, but I have included this classifier because NB classifiers scale well to real-world problems [33] and are commonly used in NLP for spam detection [33]. I have built a Gaussian NB Classifier (one that assumes the data is normally distributed). This classifier has been built using only Python 3.10 and is tested using cross-validation [34]. The Naïve Bayes probability calculation is shown in fig 3.6.

fig 3.6: Naïve Bayes probability model

$$P(X|Y) = \frac{P(Y|X)P(X)}{P(Y)}$$

where:
P(X) is the probability of X occurring
P(Y) is the probability of Y occurring
P(X|Y) is the probability of X given that Y has already occurred
P(Y|X) is the probability of Y given that X has already occurred

3.3.2.2 Random Forest, Stochastic Gradient Descent, and Support Vector Machine Classifiers

These three models have been built using Python 3.10 and Scikit Learn code. I have chosen to add these models because I want to explore as many possible options for carrying out source code classification tasks. The architectures of these models are pictured in figures *3.7*, *3.8* and *3.9* respectively.

Fig 3.7: Random Forest classifier architecture [35].

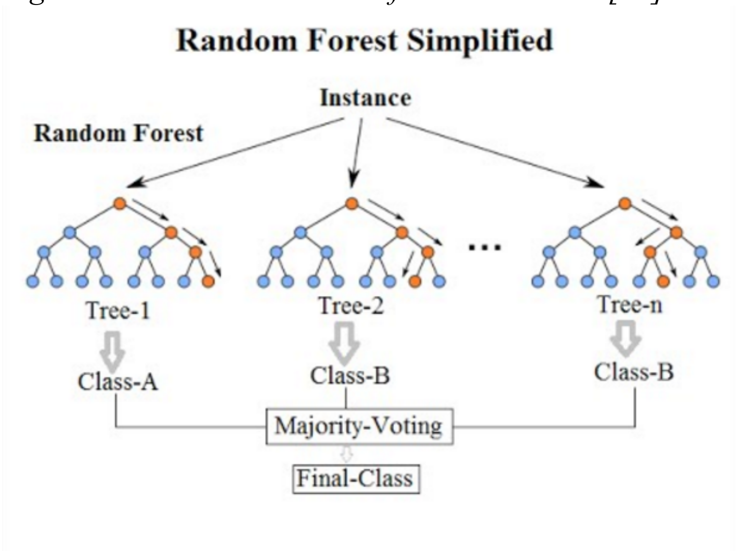


fig 3.8: Gradient Descent Algorithm pictured as a graph [36]

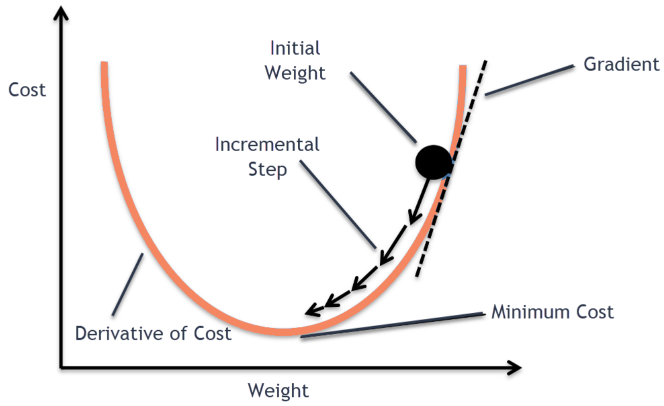
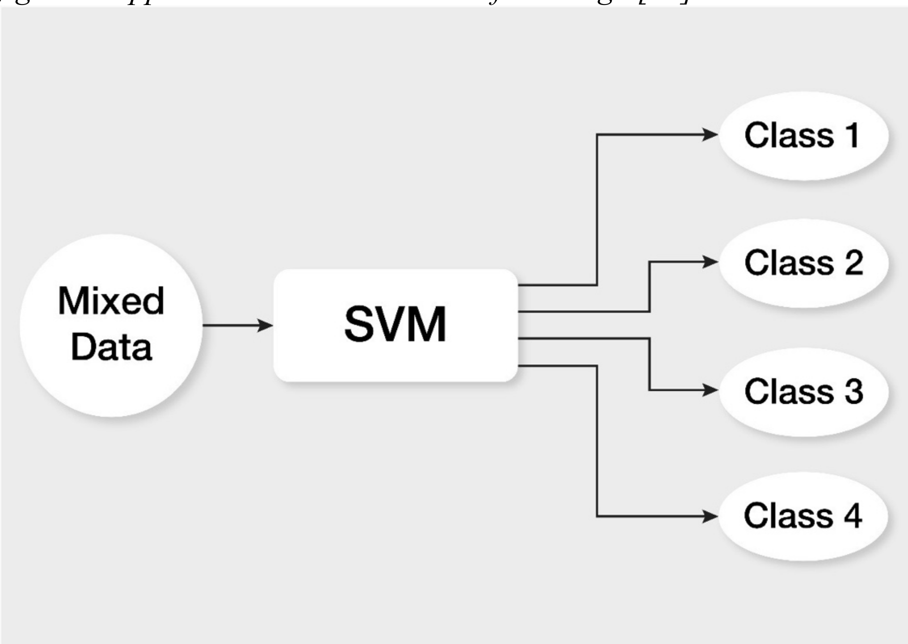


fig 3.9: Support Vector Machine classifier design [37]



4 METHODOLOGY AND IMPLEMENTATION

4.1 Methodology

The chosen methodology for achieving the project aims is to build multiple classifiers and NN models, train them using a section of the dataset and test them on unseen data from the dataset. I have made use of Python 3.10 for the implementation of the entirety of this project. I made this decision because a wide range of DL libraries can be used with Python. Furthermore, this decision was reinforced by Keras and TensorFlow being built for Python programming. These libraries provide potent tools for building NN models, which is essential to this project. Additionally, all the previous work referenced in the literature review (section 1.3) uses Python for implementation, reinforcing my decision.

4.2 Implementation

This project has been implemented using the Object-Oriented Programming paradigm [38]. Each file in the entire program – except the experiment files – contains a class that defines all the properties needed from that file. Files, or objects, are called by creating instances of the class declared in the respective file.

The implementation process has been broken down into five stages:

i. Pre-Processing Stage.
ii. Embedding Stage.
iii. Data Structuring Stage.
iv. Model Implementation or Neural Network Construction Stage.
v. Model Execution or Neural Network Processing Stage.

4.2.1 Pre-Processing Stage: Classes for Reading and Parsing Program Files

Parsers: The parser class is the first step in the implementation process. For each data structure, I have implemented a parser class (TextParser for text, GraphParser for graphs, and TreeParser for trees). Each of these deals with reading the program files' data and converting it into either text, a tree, or a graph. This class is also responsible for assigning the class labels (0 for Merge Sort, 1 for Quick Sort) to the appropriate file.

- TextParser Class (TextParser.py): With the text data structure, the parsing process starts and ends in the parser class. Once the files are read in this class and class labels are assigned, the data is split into training and testing data (70% for training, 30% for testing). The next step is the vectorization of the programs as text. This vectorization process is straightforward and is carried out by the Scikit Learn Count

Vectorizer [39], which converts the programs' contents into vectorized tokens, assuming they are no different from HLT. This is the end of the pre-processing process for text.

- GraphParser Class (GraphParser.py): This class is initialised with one parameter: ‘hashed: bool’. This means that ‘hashed’ is a Boolean variable that, if set to ‘True’, denotes that the parser is to run the HashVisitor class. If it is set to false, the parser is to run the Visitor class. The GraphParser class contains six functions. The first of these is ‘convertToGraph’. This function takes a single file path as its parameter and reads the file corresponding to the file path using the Python AST class ‘ast. parse’ [40] method. This method returns an AST, which is then passed to a Visitor object, traverses the AST and returns the list of edges present in the AST.

The next functions are ‘assignLabels’ and ‘assignLabelsToFiles’ which assign class labels to program files and return the graph representations and class labels of all the files in the dataset. After these come the ‘convertToMatrix’ and ‘extractNodes’ methods. ‘convertToMatrix’ converts an individual graph to its matrix representation using the NetworkX API [41], while ‘extractNodes’ extracts a list of nodes from the NetworkX digraph object.

The final method is ‘readFiles’, which combines all the methods described above and executes the stream of reading all the files, converting them to graphs and assigning labels.

- TreeParser Class (TreeParser.py): This class is also initialised with one parameter: ‘hashed: bool’. It has three functions. The first, called ‘parse’, takes a file path as its input and executes the same function as the ‘assignLabels’ in the GraphParser class. The next is ‘createTreeFromEdges’, which, given a list of edges present in a tree, creates a tree made up of TreeNode objects. The final is ‘convertToTree’ which reads a program file, applies either Visitor or HashVisitor to the file, and then runs ‘createTreeFromEdges’ on the list of edges returned by the Visitor or HashVisitor.

Visitor Class (Visitor.py): The process of conversion from code to graph or tree is carried out by the ‘Visitor’ class. This class inherits from the Python AST Nodevisitor class [40]. Its function is to traverse the AST of the program and add nodes to the tree or graph according to the relationships observed in the program’s AST. This file comprises three classes; the AbstractVisitor super class, which defines the essential functions required by its subclasses, and the Visitor and HashVisitor subclasses, which inherit from AbstractVisitor. The HashVisitor establishes the process of traversing a graph and embedding the nodes by implementing a hashing function. The Visitor class leaves the nodes to be embedded later by a vectorization function (unhashing). The difference between hashing and unhashing is explained in further detail in section 4.2.2.

- The AbstractVisitor Class defines multiple functions. The first of these is the generic_visit method that is inherited from the Nodevisitor object. This method is an abstract method that is left to be implemented by each individual subclass. AbstractVisitor also implements three methods (splitSnakeCase, splitCamelCase and splitIdentifier) for splitting the identifier of an object or AST node into separate words based on either snake case (e.g., snake_case) or camel case (e.g., camelCase). The next method implemented in this abstract class is the convertToGraph function. This method creates a NetworkX directed graph [41] from a list of edges and returns a NetworkX Digraph object. The remaining methods (visitSpecial, visitDef, etc.)

are concerned with traversing the AST, getting the type of an AST node (each object in the AST) and returning a string representation of the AST node.

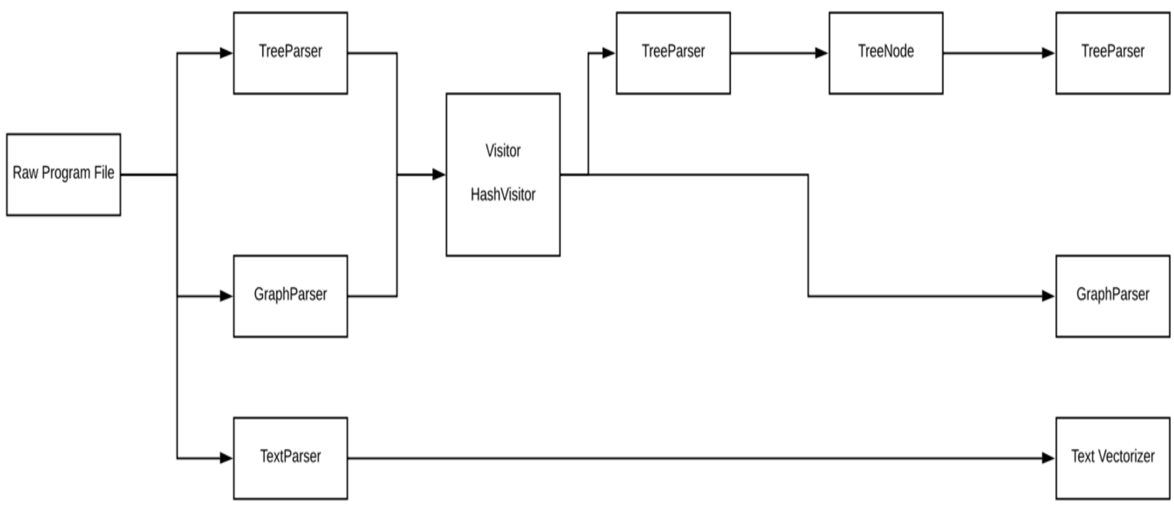
- Visitor: The Visitor subclass implements a generic_visit method that takes an AST root node and simply traverses the children of the node recursively and creates the required list of edges from the node's children. This means that the generic_visit method 'calls' itself repeatedly until all the nodes in the AST have been visited, preventing the need for implementing the method multiple times or using nested loops, which take more time to run as the number of nested loops increases.

- HashVisitor: The HashVisitor subclass implements a generic_visit method that takes an AST root node as its input and gets the string representation of the node by calling the visitSpecial function inherited from the AbstractVisitor class. After this string representation is collected, it is hashed, and the result of this hash is divided by 1. Additionally, the node object is hashed and has its result divided by 1. Finally, these two hash values are added together to get the vectorized embedding for that node. This generic_visit method also implements recursion.

TreeNode Class (TreeNode.py): This class defines a node object for the tree. On initialisation, it takes an embedding as its parameter, representing that node's embedding. In addition, it implements a function called 'preOrderTraversal'. This method traverses the tree using Pre-Order Traversal [42] and returns the list of all the nodes in the tree. It also uses recursion to avoid making use of nested loops.

Overall, the flow of data across these classes is demonstrated in *fig 4.1* below.

fig 4.1: Dataflow diagram for pre-processing stage



4.2.2 Embedding Stage: Vector Embeddings for Trees and Graphs

Before a neural network can be trained on any type of data, that data must first be converted into vector embeddings because this is the only data format that NNs and ML models can process. This is the second step in the implementation process. Each model I build will be tested using two types of vector embeddings: Hashed and Unhashed. I have described the process of Hashing in the HashVisitor section of section 4.1.1. The alternative to this will be referred to as 'Unhashing'.

4.2.2.1 Unhashing in Trees (TreeEmbeddingLayer.py)

This class implements an embedding feature that works like this:

i. Run pre-order traversal [42] on the root node of the tree to get all the nodes in the tree and their respective children.
ii. Get the depth of the tree (maximum number of nodes along a single branch).
iii. Create a copy of the list of all nodes, call it 'unvectorized'.
iv. Initialise random weights and biases for each node in the tree.
v. Create a random float value between 0 and 1, assign this value as the vector embedding of the root node and add it to a new list called 'vectors'.
vi. Remove the root node from the unvectorized list.
vii. Check if the length of the unvectorized list is not equal to zero. If yes:
   a. For each node in the list of nodes (that is not the root node):
      1. Check if it is in the unvectorized list.
      2. If yes:
         - Remove it from the unvectorized list.
         - Get the index of the node and the index of its parent node.
         - Run the vectorization function (outlined at the end of this section) on the node.
         - Add the result of vectorization to the vectors list.
         - For each child of the current node, repeat the process from point 'vii'.
viii. If no, the length of the unvectorized list is equal to zero, so stop the process because all the nodes have been vectorized.

4.2.2.1.1 Tree Vectorization Function (vecFunction):

The vectorization function for an individual node is described below.

i. Get the number of children the node has.
   - If this value is greater than 0:

$$pre = td * \left(\frac{pcc}{cc}\right) * (weights_{pi} + weights_{ci})$$

$$\begin{gathered} where\ pi\ is\ the\ index\ of\ the\ parent\ node, \\ ci\ is\ the\ index\ of\ the\ child\ node, \\ pcc\ is\ the\ parent's\ child\ node\ count, \\ cc\ is\ the\ child\ node's\ child\ count, \\ and\ td\ is\ the\ tree\ depth \end{gathered}$$

   - Else if it is 0:

$$pre = td * (weights_{pi} + weights_{ci})$$

ii. $a = pre + bias_{ci}$
iii. $result = \log(a) * 0.1$
iv. $if\ result < 0: result = result * -1.0,$
v. $return\ result$
vi. The value of result is the node embedding

4.2.2.2 Unhashing in Graphs (GraphEmbeddingLayer.py)

This class implements an embedding feature that works in an almost identical way to the tree embedding function. The only difference is that the graph vectorization function does not use graph depth in its calculations.

After vectorisation is carried out in both the tree and the graph, each set of embeddings is saved into a text file. This prevents the embedding process from occurring all the time and slows down the execution process unnecessarily. For all the stages that come after this, the contents of the text files are read to be used for processing.

4.2.3 Data Structuring Stage

In the text vectorizer, all the text is fitted based on a fixed size. This means that at the end of the vectorization process, the vectorized forms of all the files in the dataset are the same size. However, when working with graphs and trees, each tree or graph has its structure and size, which might be completely different from all the other trees and graphs. These cannot be passed to a neural network because NNs require all the inputs to be of the same fixed size. To circumvent this issue, I have applied two methods. The first is Segmentation [43], where each graph or tree is clustered down into segments of a fixed size, and the second is Padding.

4.2.3.1 Padding (GraphDataProcessor.py)

The length of the longest tree or largest graph (maxLen) is derived. All the other trees and graphs are padded with zeros until they are as long as maxLen to ensure all the inputs are the same size. I have excluded padding from trees in this project because all my experiments with trees and padding resulted in training accuracies of over 80% and validation accuracies of less than 30%. The reason this is insufficient is explained in section 5.1.

Another reason padding is insufficient is that the model will be unable to handle unseen trees or graphs that are longer maxLen.

4.2.3.2 Segmentation (TreeSegmentationLayer.py, TreeSegmentation.py, TreeDataProcessor.py and GraphDataProcessor.py)

Segmentation [43] is a process that is usually applied in image classification and computer vision. It is a process where parts of a vector representation of an image with the same class label are combined into the same cluster. I have decided to use segmentation to prove that methods for other classification tasks can be applied to source code problems, providing good results if the underlying data structure accurately captures the program's information.

The segmentation function has been implemented using TensorFlow. TensorFlow has eleven different options for segmentation based on two categories: sorted and unsorted segmentation. Unsorted segmentation can be done using one maximum, mean, minimum, square root of n, sum and product. Sorted segmentation is the same, except it does not include the square root of n.

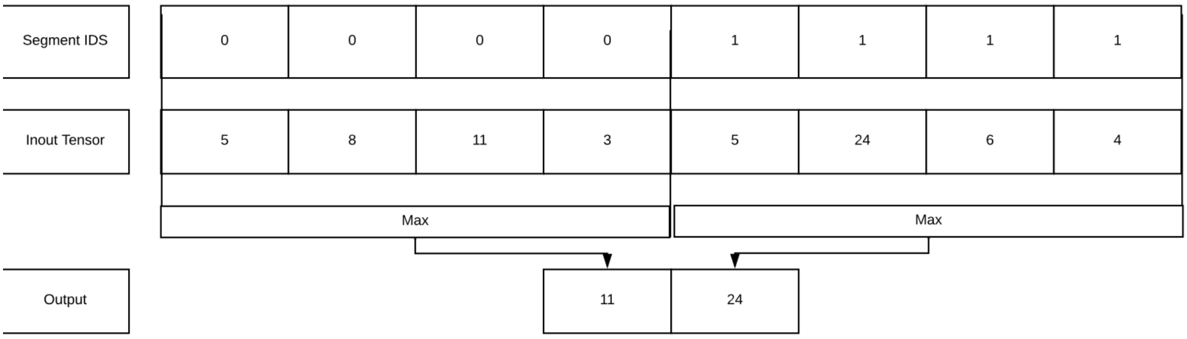
The way segmentation generally works is that the programmer defines the segment ids as in *fig 4.2* and declares several segments (numSegments as in *fig 4.2*), which must be equal to the number of values in the segment ids. The dataset is then split into segments corresponding to the values in segment ids. For example, if you want three segments, segment ids must have three values, e.g., [0, 1, 2]. The first dimension of the dataset to be segmented must also be the same size as the number of segments; else, an error will occur. If max segmentation has been used, for example, a segment of [1, 2, 3, 4] will have [4] as its output – the maximum value in that segment. An example of sorted max segmentation can be seen in *fig 4.3.*

In this project, the first step in segmentation is to decide the number of segments n to be used. After deciding on this number, each graph or tree's embedding list is converted to an n-dimensional version of itself. Then, segmentation is carried out on this new n-dimensional version. For example, the Python list [1, 2, 3, 4] with 3 segments becomes [[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]]. I experimented with varying segments, starting from 10 and decided on 40 after observing that a higher number of segments produces better results from the models.

fig 4.2: Declaration of a segmentation function using unsorted square root of n with segment count = 40

```
def runSegmentation(self, nodeEmbeddings: tf.Tensor, numSegments: int):
    segments = tf.constant([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
    11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
    22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39])
    segFunc = tf.math.unsorted_segment_sqrt_n(nodeEmbeddings, segments, num_segments = numSegments)
    return segFunc
```

fig 4.3: Example of sorted max segmentation



4.2.4 Model Implementation or NN Construction Stage

4.2.4.1 Deep Learning Models

The Deep Learning models developed in this project have two main things in common. First, they accept only tensors [44] as their input. The second is that they make use of the same four activation functions.

4.2.4.1.1 Activation Functions

An activation function is a function that is applied to a value in a neural network to get its output. I have tested each DL model using four different activation functions – Tanh, ReLu, SoftMax and Logarithmic Sigmoid – to determine which activation function produces the best outputs. The formula of each function is shown below.

1. Tanh: $f(x) = \frac{(e^{2x}-1)}{e^{2x}+1}$ $where\ x = (input *\ weight) + bias$

2. ReLu: $f(x) = \max(0, x)$ $where\ x = (input *\ weight) + bias$

3. SoftMax: $f(x) = \frac{e^{xi}}{\sum_{y=1}^{L} e^{\beta xy}}, where\ x = (input *\ weight) + bias$

4. Logarithmic Sigmoid: $f(x) = \frac{1}{1+e^{-x}}\ , where\ x = (input *\ weight) + bias$

4.2.4.1.2 MLP Model (MLP.py)

This model is made up of 8 functions. The most important is the 'runFFModel', where FF stands for 'Feed Forward'. This method is where all the other methods are initiated from. It starts by initialising the model weights and biases by calling 'initialiseWeights'. Next, a for loop is created based on the number of epochs, and the 'backPropagate' method is called. This method, in turn, starts by calling the 'FFLayer' method. In the FFLayer method, the inputs are multiplied by the weights and added to the biases across all the hidden layers – a forward pass. Then, the applicable activation function is gotten from the 'getActivationFunction' method and is applied to the output of each layer to give a final weight. After running the FFLayer method, the predictions are returned to the backPropagate method. Next, these predictions are used to calculate the loss value. Finally, the loss is calculated in the 'lossFunction' method.

This method calls the TensorFlow 'softmax_cross_entropy_with_logits' function on the predictions and the actual labels. Its value is returned to backPropagate where the gradient of the loss and weights is calculated. The results of this calculation are used in the 'updateWeights' function to update the weights. This process is what is known as back-propagation [24]. After the weights are updated, the updated weights are applied to the inputs to get more accurate predictions. Finally, the SoftMax activation function is applied to the outputs to derive the final prediction.

Finally, the model is applied to predict the labels of the testing data in the 'makePredictions' method. This method runs the model on the test data, applies SoftMax to the outputs and calculates the accuracy of the predictions using the NumPy or TensorFlow argmax function. The implementation of runFFModel and backPropagate are shown in *fig 4.4* and *4.5,* respectively.

fig 4.4: Implementation of runFFModel method in the MLP class

```
def runFFModel(self, x_train, y_train, x_test, y_test):
    index = 0
    loss = 0.0
    metrics = {'trainingLoss': [], 'trainingAccuracy': [], 'validationAccuracy': []}
    self.initialiseWeights()
    for i in range(self.epochs):
        print('Epoch {}'.format(i), end='........')
        predictions = []

        if index % 5 == 0:
            print(end=".")
        if index >= len(y_train):
            index = 0

        # FIRST FORWARD PASS
        loss = self.backPropagate(x_train, y_train)
        # SECOND FORWARD PASS/RECURRENT LOOP
        newOutputs = self.FFLayer(x_train)
        pred = tf.argmax(tf.nn.softmax(newOutputs), axis = 1)
        predictions.append(pred)
        index += 1

        predictions = tf.convert_to_tensor(predictions)
        # print(predictions)
        metrics['trainingLoss'].append(loss)
        unseenPredictions = self.makePrediction(x_test)
        metrics['trainingAccuracy'].append(np.mean(np.argmax(y_train, axis=1) == predictions.numpy()))
        metrics['validationAccuracy'].append(np.mean(np.argmax(y_test, axis=1) == unseenPredictions.numpy()))
        print('\tLoss:', metrics['trainingLoss'][-1], 'Accuracy:', metrics['trainingAccuracy'][-1],
              'Validation Accuracy:', metrics['validationAccuracy'][-1])
    return metrics
```

fig 4.5: Implementation of backPropagate method in the MLP class

```
def backPropagate(self, xValues, yValues):
    with tf.GradientTape(persistent=True) as tape:
        output = self.FFLayer(xValues)
        loss = self.lossFunction(output, yValues)

    for i in range(1, self.layerCount):
        tape.watch(self.weights[i])
        self.weightDeltas[i] = tape.gradient(loss, self.weights[i])
        self.biasDeltas[i] = tape.gradient(loss, self.bias[i])

    del tape
    self.updateWeights()
    return loss.numpy()
```

4.2.4.1.3 LSTM, GRU & Simple RNN (RNN.py) Models

This model is made up of 7 methods. The first is ‘RNNLayer’, which creates and returns a TensorFlow Simple RNN layer object. The second is ‘LSTMLayer’, which is the same as RNNLayer but returns a TensorFlow LSTM layer object instead. The third is ‘GRULayer’, which is the same as the two before it, but it returns a GRU layer object. The parameters used here are activation function – any of the four listed in section 4.2.4.1 –, useBias, the value of which is determined at runtime, input shape, which is the shape of the first dimension of the input and the number of neurons, which is also defined at runtime. This is followed by the ‘DenseLayer’ function, which returns a TensorFlow Dense Layer object.

‘getActivationFunction’ comes next. Depending on the activation function passed as the desired input, this method returns the corresponding TensorFlow function. The only exception is the logarithmic sigmoid function which I have implemented using a combination of standard Python 3.10 and TensorFlow.

The final function is the function that calls and executes the LSTM, SRNN and GRU. This method has been called ‘runModel’. It begins by instantiating the input layer, the input shape and the output layer (dense). It then creates a Sequential [45] Keras model and adds the input layer as the first layer. The next step is to decide if the model will be an LSTM, a GRU, or a Simple RNN. This decision will be based on the input to the method. Finally, the output layer is added after deciding which type of hidden layers the model will have.

The model is compiled and fitted on the training and testing data with a batch size [46] and the number of epochs [47] determined at runtime.

4.2.4.1.4 Dense Model (DenseModel.py)

This model is made up of only one function called ‘runDenseModel’. This method creates a sequential Keras model, adding an input layer. Three densely connected layers follow this input layer with 128, 64 and 2 neurons, respectively. The model parameters used for fitting are binary crossentropy to calculate loss, the Adam optimizer for optimization and batch size and number of epochs to be determined at runtime.

4.2.4.2 Non-Deep Learning Models

4.2.4.2.1 Gaussian Naïve Bayes Classifier (NaiveBayes.py)

This model has been written in Python 3.10. It contains 12 methods, the most important of which is the cross-validation method (gaussianCrossValidation), as seen in *fig 4.6*. This method is used to validate the results of the Gaussian NB classifier. First, it takes all the program files and all the class labels as inputs. It then calls the ‘splitIntoFolds’ method, responsible for splitting the data into folds of equal lengths. After the data has been split, the folds are looped through, and each fold is used as the testing data exactly once, while all the other folds are used as the training data. Finally, the classification is done by the ‘runGNBClassifier’ method, which takes the training folds, the training label folds and the test fold as input. The implementation of this method can be seen in *fig 4.7* alongside the method for making predictions, ‘makePredictions’.

The remaining methods include the method for calculating the accuracy score (calculateAccuracyScore) – which calculates the percentage to which the predictions are equal to the actual labels – and the ‘getMean’ function that calculates the mean of a set of values. There is also the ‘separationFunction’ method that separates the data based on classes, and the ‘getStandardDev’ function that calculates the standard deviation of a set of values. Other methods include ‘collateStatistics’ and ‘collateClassStatistics’, which collate the statistics (mean and standard deviation) for each row and class.

Finally, the last two methods are the methods that calculate the Gaussian probability for a row and a class. These are called ‘getGaussianProbability’ and ‘getClassGaussianProbability’, respectively.

fig 4.6: The implementation of the cross-validation function on the Gaussian NB classifier

```
def gaussianCrossValidation(self, xValues, yValues):

    # split the data into folds of equal values:
    foldsX, foldsY = self.splitIntoFolds(xValues, yValues)
    accuracyScores = []

    # for each data fold, separate it from the rest of the folds
    # use the other folds as training data and use the current fold as testing data
    # calculate the and return the mean accuracy score when the prediction process is complete.
    for i in range(len(foldsX)):
        currentTrainX, currentY = foldsX[i], foldsY[i]
        allTrain, allY, allTest, allTestY = list(foldsX), list(foldsY), [], []

        allY.remove(currentY)
        allTrain.remove(currentTrainX)
        allTrain = sum(allTrain, [])
        allY = sum(allY, [])
        for j in range(len(currentTrainX)):
            trainY = currentY[j]
            trainX = currentTrainX[j]
            allTest.append(trainX)
            allTestY.append(trainY)

        predicted = self.runGNBClassifier(allTrain, allY, allTest)
        accuracyScore = self.calculateAccuracyScore(allTestY, predicted)
        accuracyScores.append(accuracyScore)
    return mean(accuracyScores)
```

fig 4.7: The methods to make predictions and carry out classification

```
def makePrediction(self, classStats, xValue):
    # Get the gaussian probabilities of the class based on its mean and standard deviation
    classProbabilities = self.getClassGaussianProbability(classStats, xValue)
    label, probability = 10, -1
    classProbs = classProbabilities.items()
    for value, prob in classProbs:
        if label == 10 or prob > probability:
            probability = prob
            label = value
    return label

def runGNBClassifier(self, xTrain, yTrain, xTest):
    # Get the mean and standard deviation for the training folds
    classStats = self.collateClassStatistics(xTrain, yTrain)
    predictions = []
    for x in xTest:
        prediction = self.makePrediction(classStats, x)
        predictions.append(prediction)

    return predictions
```

4.2.4.2.2 SGD, RF & SVM Classifiers (SKLearnClassifiers.py)

These classifiers have been implemented using the Scikit Learn machine learning library [48]. I chose this library because it provides easy-to-understand code and a wide range of classification models. Each of these three classifiers is declared as its own function. The functions here are SGDClassify, SVMClassify and rfClassify. Each method takes four inputs: a set of training data, training data labels, testing data and testing data labels. The first step in each method is to create a 'Pipeline' [49]. This simplifies the process of calling the classifier and saves time at runtime. After the pipeline is created, it is fitted on the training data and training data labels and the testing data is passed to the pipeline to make predictions. After the predictions are made, the accuracy of the predictions is calculated and returned.

4.2.5 Model Execution or NN Processing Stage

(textExperiments.py, treeExperiments.py and graphExperiments.py)
This stage is the final stage of the implementation phase of the project. I have created three separate files, each corresponding to one of the data structures. Each file calls its respective parser and data processor classes to get the training and testing data. After this data is collected, each model is run across all data structures using hashed and unhashed data.

These three files contain all the experiments used to get the results. These results are analyzed in the next section.

5 RESULTS, ANALYSIS AND EVALUATION

5.1 Minimum Criteria for Success

For this project, the metrics I will be examining as a measure of success are the accuracy of each model and the loss of each DL model.

5.1.1 Deep Learning Models

For each deep learning model, there are two different accuracy scores; Training Accuracy (TA), which is the accuracy of the model's predictions on seen training data, and Validation Accuracy (VA), which is the accuracy of the predictions on the testing data.

The TA is important because it shows how well the model has learnt the training dataset. Due to the relatively small training and testing datasets, the acceptable scores for the TA that denote a successful model will be any scores from 60%. Any TA scores below this range will be considered as too low or not accurate enough. This will indicate that the model needs to have its parameters adjusted, the data format applied to the model is leading to a loss of important information. It might also be an indicator that underfitting is occurring (where the model has not learnt the training data well enough).

The VA is important because it demonstrates how well the model works when classifying data it has not learnt (unseen data that is not included in the training dataset). The acceptable scores for the VA that indicate success are any values from 70%. Any scores below this will lead to the respective model being considered insufficiently accurate. Therefore, I consider the VA to be more important than the TA. This is not to say that the TA is unimportant, but rather, that the VA gives a better indication of how well each model performs when tested on new data.

Another metric usually observed in DL models is the loss [50]. A loss here refers to the level of error the model produces when carrying out classification. A higher value for loss means that the model makes more inaccurate predictions. The loss values are Training Loss (TL) and Validation Loss (VL). Because I am applying binary crossentropy [51] as my loss function, I expect the loss values to be relatively high. This is because of the nature of crossentropy functions where a single incorrectly classified value can cause the loss to increase exponentially. Usually, when working with NNs, the loss is expected to be as low as possible. Still, because I have applied crossentropy, I will consider a good loss value as anything below 1.0.

To conclude, the measure of a successful Deep Learning model is a TA of at least 60%, a VA of at least 70%, with a TL and VL below 1.0 each.

5.1.2 Non-Deep Learning Models

Due to the Scikit Learn classifiers (SGD, SVM & RF), the only score produced from these models is the Validation Accuracy (VA). A good VA here will be anything higher than 70%.

The Gaussian NB classifier uses cross-validation [34] to verify the accuracy of the models across each fold. As a result, the only metric to be observed here is the mean Validation Accuracy from running cross-validation on all the folds.

5.2 Setting Up and Executing the Experiments

I will run experiments on the nine models for each information storage structure. Each storage structure will contain all its experiments in the same file to make identifying the most accurate model easier.

To execute the models, I have created a Boolean variable called 'hashed'. If this value is set to 'True', the model is tested on the hashed version of the embeddings. If it is set to 'False', then the model is being tested on the version of the embeddings calculated using the vectorization function (unhashed). I have run experiments on both the hashed and unhashed versions of the datasets.

5.3 Analysis and Evaluation of Results

5.3.1 Result Overview

The general observation in this project is that non-deep learning models perform much better than conventional deep learning models when trained and tested on source code using the text data structure. Regardless, this project aims to determine which DL architecture is the most suitable for carrying out classification tasks on source code.

The tree-based models outperform the graph-based models. This indicates one of two things. The first is that the tree data structure is better than graphs for classifying programs based on what algorithm they implement. The second is that in vectorizing the graph-based data, not enough information is being learnt about each node, leading to a loss of important information in the node embeddings.

Unhashed nodes with trees produced the best results – several TA scores above 60% and several VA scores greater than or equal to 70%. None of the experiments on graphs had VA results that can be considered accurate enough, but some experiments produced TA scores in the acceptable range. I will explore these results in detail in sections 5.3.2, 5.3.3 and 5.3.4.

5.3.2 Results from Experiments on Text

The results from all the models are outlined in *table 5.1*. The results from running the models on text have proved to be very interesting. The DL models all perform relatively averagely, as expected. What is unexpected, however, is that the NDL models perform incredibly well when trained and tested using text-based data. An example is the Gaussian NB classifier which produced a VA of 86.88%, which was completely unexpected. This indicates that I have either built an exceptional model, or the Gaussian NB method for classification is well suited for classifying programs based on the algorithms they implement.

Even more interesting is the Random Forest classifier, which produces a VA of 100% when tested on text. RF classifiers are based on trees, so this could be a possible reason that this classifier works so well on the classification task for this project. In the future, it is worth exploring the architecture of RF classifiers to observe how they process input.

The SVM classifier produces a VA of 76%, which is also considered as good. In comparison, the SGD classifier is the least accurate NDL model, with a VA score of 64%, which is below the threshold for being considered a precise model. In the future, it will be worth exploring why this model performs worse than the other NDL models.

The conclusion to be drawn from this is that when using text-based data, non-deep learning models are better than deep learning models.

5.3.2.1 Statistics and Performance

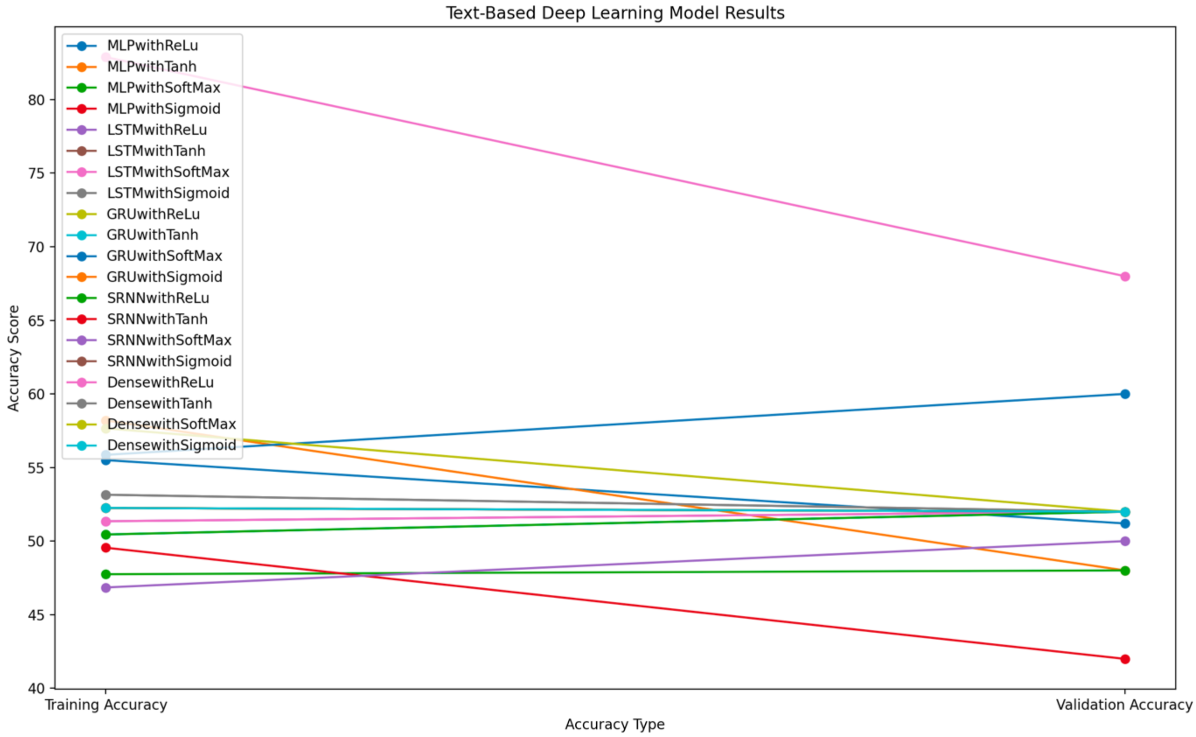
The Deep Learning model results for text are not as remarkable. The mean TA is 53.88%, while the mean VA is 52.16%. The best performing model was the Dense model, which produced a TA score of 82.88%, a VA score of 68%, a TL of 0.66 and a VL score of 0.83—using the ReLu activation of 256 neurons, a batch size of 10, and 10 epochs. This sharp drop in values between TA and VA is demonstrated in the graph pictured in *fig 5.1*. The accuracy and loss values are in the range to be considered a good model for classifying source code based on sorting algorithms. However, the VA score falling just outside the acceptable range for VA scores indicates that the model is either insufficient to classify unseen data or that overfitting might be occurring in the training phase, and the model is learning the training data too well.

The next best DL model was the GRU model with SoftMax activation, 256 neurons, and batches of 10, and 10 epochs. This model produced a TA score of 55.86%, VA of 60%, a TL of 0.68 and a VL of 0.69. This result further proves that converting source code to tokens as if it were text is insufficient for classification in DL models. The LSTM models all produced the same VA, indicating that the activation function used with this model is not a significant factor in determining the model's output. All the MLP models produce VA scores in the 48% - 52% range, supporting this assumption.

The worst performing DL model overall was the Simple RNN model with Tanh activation, 256 neuron units, and batch size and epoch size of 10 each. This model produced the lowest VA with 42% and a TA of 49.55% (the second lowest, the lowest overall was the Simple RNN model with SoftMax activation and a TA score of 46.85% but a VA score of 50%).

To sum up, I observed that the Simple RNN model performed the worst across the board, regardless of the activation function. This indicates that the updated versions of this model (the LSTM and the GRU) are better suited for carrying out the classification tasks in this project and may be better suited to classification tasks in general. The results also show that the activation function used only creates a marginal difference in results. These results have been plotted in a graph in *fig 5.1* to show the changes in TA and VA scores.

fig 5.1: Graph showing changes between TA and VA from text-based deep learning models



5.3.3 Results from Experiments on Graphs

The results derived from running each model are displayed in tables *5.2* and *5.3*. The headings indicate each metric for both padding (PTA – Padded Training Accuracy, PVA – Padded Validation Accuracy, PVL – Padded Validation Loss, and PTL – Padded Training Loss) and segmentation (STA – Segmented Training Accuracy, SVA – Segmented Validation Accuracy, SVL – Segmented Validation Loss and STL – Segmented Training Loss). In addition, the results from running the models on graphs with vectorized (unhashed) embeddings are displayed in *table 5.2. The* results from running the models using the hashed embeddings are in *table 5.3*.

The results from these experiments on graphs do not appear interesting at first glance. The hashed and unhashed embeddings provide very similar results. The padded and segmented graphs also give results that do not differ much.

5.3.3.1 Statistics and Performance

The mean TA for padded and hashed graphs is 57.64%, while the mean VA was 48.76%. The mean TA for segmented and hashed graphs is 53.49%, with a mean VA of 51.85%, the highest mean VA. The lowest average VA is from padding and unhashed graphs at 46.25%. One assumption that can be drawn from the similarity of these values (VA range = 5.2) is that with graphs, both padding and segmentation are insufficient for evening out the sizes of the graphs, and another alternative is necessary.

The model with the highest TA scores was the Dense model using ReLu activation, 256 neurons in the first hidden layer, 128 neurons in the second hidden layer, and 10 for both batch size and epochs. Its scores were 100% using padded graphs and 88.39% using segmented graphs. This might seem like a good result without further exploration, but a closer look at the VA scores indicates otherwise. The VA scores for padding were 40.82%, and 65.31% for segmentation. These high TA scores with low VA scores show that during the training phase, the model is learning the training data a little too well and has become unable to generalize to new data (overfitting). The Dense model also experiences overfitting when used with Tanh and Sigmoid activation functions. The Tanh activation function provides TA scores of 98.21% and 69.64% on padding and segmentation, respectively. The Tanh VA scores are much lower, with 48.98% on both padding and segmentation.

The Dense model with Sigmoid activation results in a TA score of 86.61% on padding and 56.25% on segmentation. Here, the segmentation model does not seem to experience overfitting, with a VA score of 59.18%. On the other hand, the padded model also experiences overfitting with a considerably lower VA of 48.98%.

In general, the best performing graph-based model was the Simple RNN with SoftMax activation, 256 neurons, 10 epochs and batch sizes of 10. For padding, it produced TA of 61.61% and VA of 67.35%, the highest VA of all the graph-based experiments. This contrasts with the text-based model, where the SRNN was generally the worst performing model across the board.

The loss values mainly were under 1.0, with a few exceptions. The GRU regularly produced the highest TL values with an average of 5.955. This indicates that the GRU model is experiencing the most incorrect classifications.

The NDL models do not perform very well, as seen in tables *5.2* and *5.3*. In comparison to their results on text, they perform much worse. This could be because these models are specifically designed to work with text-based data and do not generalize well to other types of data structures.

Overall, the results from these graphs experiments indicate that padding and segmentation are not well suited for carrying out classification tasks on graph-based source code, and a more accurate alternative is necessary. The results from the NDL models reinforce this conclusion due to the enormous drop in VA scores compared to their results on the text-based models. It would also be interesting to observe how all these scores increase or decrease when more data is added for training and testing.

5.3.4 Results from Experiments on Trees

To carry out segmentation on the trees, I initially experimented with all 11 options provided by TensorFlow and found that all the functions give approximately the same results, except the mean function, which produced better results than the others. As a result, I have chosen to use unsorted mean segmentation for my final experiments. The preliminary experiments to compare the results of the different segmentation methods have been included in a file called 'preliminaryTreeExperiments.py'. In addition, the results for the tree experiments using unhashed and hashed embeddings can be found in tables *5.4* and *5.5,* respectively.

5.3.4.1 Statistics and Performance

The statistics for the tree-based models with unhashed node embeddings are very promising. The mean TA for unhashing was 68.69%, with a mean VA of 61.71%. This indicates that with more data for training and testing, these scores could go up and be even more accurate. The hashed nodes do not produce as good results. The mean TA for hashing was 50.04%, with an average VA of 52.28%. The closeness of all these values indicates that there is little to no overfitting occurring in both the hashed and unhashed models. The changes between TA and VA for all the models using unhashing are plotted on the graph in *fig 5.2*.

The highest TA scores are from the Simple RNN model with 100% TA on SoftMax and Sigmoid activations. The SoftMax model produces a VA of 60.61%, while the Sigmoid model has a VA of 70%. The Sigmoid model has both values in the acceptable range for accuracy, making it the superior model. However, the SoftMax model has its VA score below the acceptable range, which indicates that in this model, either there is overfitting occurring, or the model needs to be tested using more validation data to increase the VA score.

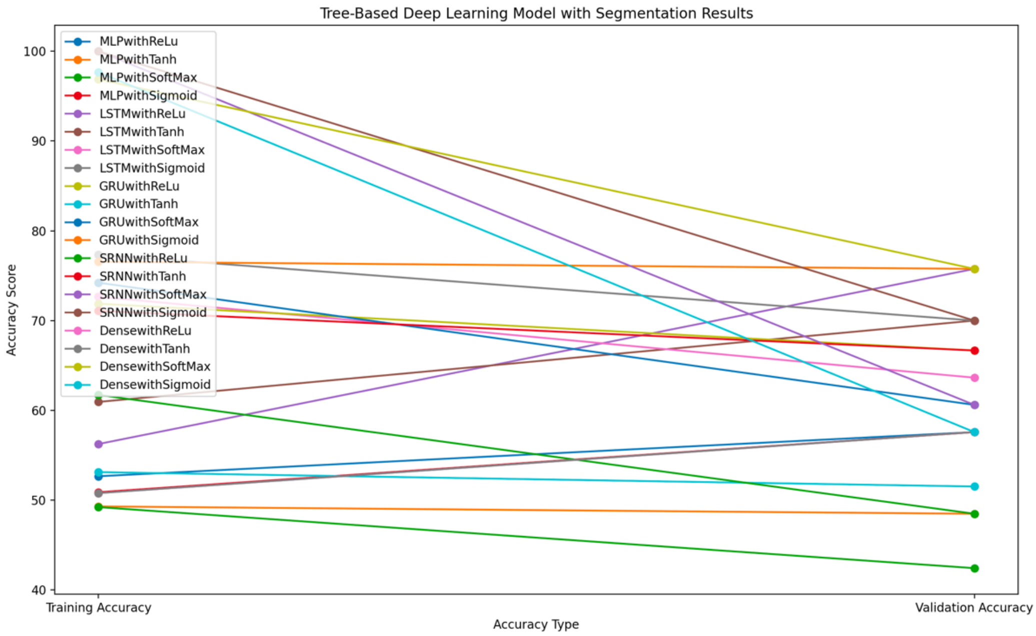
Other models that perform very well on both TA and VA include the GRU model with Sigmoid (TA 76.56%, VA 75.76%), the LSTM model with Tanh activation (TA 60.94%, VA 70%) and Sigmoid activation (TA 77.34% and VA 70%) and the Dense Model with SoftMax activation (96.88% TA, 75.76% VA). These four models prove the underlying theory behind this project that says that conventional neural network architectures can be made to process source code accurately using the right data structures. It appears that trees are an appropriate data structure for achieving this aim.

Other models with either a high TA and a low VA or a low TA and a high VA are the LSTM model with ReLu activation (TA 56.25% and VA 75.76%), the LSTM with SoftMax activation (72.66% for TA and 63.64% for VA), the GRU with ReLu (TA 71.88% and VA 66.67%) and the GRU with SoftMax (TA 74.22 % and VA 60.61%). There is also the Simple RNN, with Tanh activation, which has a TA score of 71.09% and VA score of 66.67%, and the Dense model with Sigmoid activation, which has a TA of 97.66% and VA of 57.58%. This high TA and low VA indicate that this dense model is overfitting itself on the training data and not generalizing well to the testing data.

The MLP underperformed across the board (on trees, text, and graphs), indicating that this model might be entirely unsuitable for carrying out classification tasks on source code, regardless of the data structure used to store the program. All the LSTM models had either a high TA or a high VA, or both. This indicates that these models might be even more accurate with more training and testing data. This is also the case with the SRNN model. Finally, the GRU model only had one activation that fell below the acceptable range in TA and VA. This is the Tanh activation function. This is interesting because GRUs and LSTMs usually work best with Tanh activation.

The best performing models for the tree data structure are the GRU model with Sigmoid activation and the Dense model with SoftMax activation. These models' performances show that deep learning models can be used to carry out learning tasks on source code, given that the source code has been embedded using the tree data structure. As with graphs, the NDL models generally do not perform well with trees except for the SGD classifier, which recorded a VA score of 81.82% using unhashed tress. This contrasts with its results on text (64%) and indicates that maybe the SGD classifier is better suited to work with tree-based data than text. This is a theory that needs to be explored beyond the scope of this project.

fig 5.2: Graph showing changes between TA and VA from tree-based deep learning models and unhashing



5.4 Overall Findings

The consensus from the results is that segmentation is better suited for scaling trees to be the same size and a suitable method for scaling graphs needs to be explored. Padding and segmentation are insufficient for graphs as they do not result in high training or validation accuracy levels. These results also show that unhashing in trees is a better option than hashing. This is likely because unhashing allows a node's embedding to embed information about the tree in general (e.g., tree depth, as in section 4.2.2.1.1) alongside information about its parent node, if one exists.

In addition, the research carried out for this project shows that on the classification task (see section 1.2.2), trees are better suited for representing source code than graphs when using deep learning models. On the other hand, non-deep learning models do not need the data to be converted into trees and graphs because their architectures allow the information stored in source code to be accurately captured.

Table 5.1: Results from all experiments on Text

Model	**Training Accuracy %**	**Validation Accuracy %**	**Training Loss**	**Validation Loss**
MLP with ReLu	55.50	51.2	0.91	N/A
MLP with Tanh	58.2	48.01	0.82	N/A
MLP with SoftMax	47.75	48.01	0.78	N/A
MLP with Sigmoid	52.25	52.0	0.76	N/A
LSTM with ReLu	51.35	52.0	0.70	0.70
LSTM with Tanh	52.25	52.0	0.71	0.70
LSTM with SoftMax	51.35	52.0	0.69	0.69
LSTM with Sigmoid	53.15	52.0	0.70	0.70
GRU with ReLu	52.25	52.0	0.70	0.70
GRU with Tanh	50.45	52.0	0.71	0.71
GRU with SoftMax	55.86	60.0	0.68	0.69
GRU with Sigmoid	52.25	52.0	0.69	0.70
SRNN with ReLu	50.45	52.0	6.95	6.67
SRNN with Tanh	49.55	42.0	2.65	1.30
SRNN with SoftMax	46.85	50.0	0.94	0.73
SRNN with Sigmoid	52.25	52.0	6.55	7.36
Dense with ReLu	82.88	68.0	0.66	0.80
Dense with Tanh	53.15	52.0	3.91	3.98
Dense with SoftMax	57.66	52.0	0.69	0.69
Dense with Sigmoid	52.25	52.0	0.69	0.69
Gaussian NB	N/A	86.88	N/A	N/A
RF Classifier	N/A	100	N/A	N/A
SGD Classifier	N/A	64.0	N/A	N/A
SVM Classifier	N/A	76.0	N/A	N/A

Table 5.2: Results from graphs with unhashed embeddings, padding and segmentation

Model	PTA %	STA %	PVA %	SVA %	PTL	STL	PVL	SVL
MLP with ReLu	53.21	51.61	44.69	55.31	100.29	1.61	N/A	N/A
MLP with Tanh	52.21	52.05	44.69	63.27	100.29	0.87	N/A	N/A
MLP with SoftMax	49.11	50.89	59.18	40.82	0.76	0.75	N/A	N/A
MLP with Sigmoid	58.04	47.32	46.94	59.18	0.69	0.81	N/A	N/A
LSTM with ReLu	50.89	46.43	40.82	51.02	7.53	0.70	9.08	0.71
LSTM with Tanh	48.21	51.79	40.82	51.02	0.70	0.69	0.70	0.71
LSTM with SoftMax	49.11	53.57	59.18	51.02	Nan	0.69	Nan	0.70
LSTM with Sigmoid	49.11	45.54	59.18	44.90	7.81	0.70	6.26	0.71
GRU with ReLu	48.21	59.82	57.14	44.9	7.71	0.68	7.60	0.73
GRU with Tanh	44.64	52.68	48.98	57.14	7.71	0.71	7.71	0.69
GRU with SoftMax	49.11	55.36	40.82	42.86	0.69	0.70	0.70	0.71
GRU with Sigmoid	48.21	46.43	48.98	44.90	7.71	0.71	7.63	0.73
SRNN with ReLu	49.11	47.32	44.9	51.02	5.52	7.72	3.68	7.70
SRNN with Tanh	50.0	42.86	51.02	53.06	3.11	2.64	3.06	0.99
SRNN with SoftMax	61.61	51.79	67.35	53.06	0.69	0.78	0.66	7.73
SRNN with Sigmoid	56.25	49.11	40.82	59.18	7.71	7.71	7.71	7.71
Dense with ReLu	100	88.39	40.82	65.31	1.13	0.40	2.97	1.31
Dense with Tanh	98.21	69.64	48.98	48.98	0.15	0.53	3.05	1.41
Dense with SoftMax	50.89	50.89	40.82	40.82	40.82	0.69	0.70	0.69
Dense with Sigmoid	86.61	56.25	48.98	59.18	48.98	0.83	0.72	0.69
Gaussian NB	N/A	N/A	55.0	51.25	N/A	N/A	N/A	N/A
RF Classifier	N/A	N/A	57.14	55.1	N/A	N/A	N/A	N/A
SGD Classifier	N/A	N/A	48.98	53.06	N/A	N/A	N/A	N/A
SVM Classifier	N/A	N/A	46.94	59.18	N/A	N/A	N/A	N/A

Table 5.3: Results from Graphs with hashed embeddings, padding and segmentation

Model	**PTA %**	**STA %**	**PVA %**	**SVA %**	**PTL**	**STL**	**PVL**	**SVL**
MLP with ReLu	50.89	53.56	49.39	47.55	5.03	10.69	N/A	N/A
MLP with Tanh	45.54	45.54	53.06	53.06	0.69	0.70	N/A	N/A
MLP with SoftMax	45.54	45.54	53.06	53.06	0.75	0.74	N/A	N/A
MLP with Sigmoid	45.54	45.54	53.06	53.06	0.69	0.76	N/A	N/A
LSTM with ReLu	53.57	52.68	46.94	46.94	7.71	7.71	7.71	7.71
LSTM with Tanh	55.36	43.75	46.94	53.06	7.71	7.71	7.71	7.71
LSTM with SoftMax	54.46	54.46	46.94	46.94	0.69	0.69	0.70	0.70
LSTM with Sigmoid	50.89	45.54	57.14	53.06	7.71	7.71	7.71	7.71
GRU with ReLu	56.25	47.32	46.94	46.94	7.71	7.71	7.71	7.71
GRU with Tanh	33.04	54.46	18.37	46.94	7.71	7.71	7.71	7.71
GRU with SoftMax	54.56	54.46	46.94	46.94	0.70	0.69	0.70	0.70
GRU with Sigmoid	40.18	48.21	24.49	53.06	7.71	7.71	7.71	7.71
SRNN with ReLu	45.54	53.57	53.06	46.94	8.35	7.71	7.20	7.71
SRNN with Tanh	54.46	53.57	46.94	46.94	Nan	7.71	Nan	7.71
SRNN with SoftMax	55.36	54.46	46.94	46.94	0.78	0.88	0.75	1.06
SRNN with Sigmoid	54.46	56.25	46.94	46.94	Nan	7.71	Nan	7.71
Dense with ReLu	54.46	45.54	46.94	53.06	7.71	7.71	7.71	7.71
Dense with Tanh	55.36	45.54	46.94	45.54	7.71	7.71	7.71	7.71
Dense with SoftMax	54.56	54.54	46.94	46.94	0.69	0.69	0.70	0.70
Dense with Sigmoid	43.75	46.43	46.94	46.94	0.70	0.70	0.70	0.70
Gaussian NB	N/A	N/A	53.13	50.63	N/A	N/A	N/A	N/A
RF Classifier	N/A	N/A	46.94	53.06	N/A	N/A	N/A	N/A
SGD Classifier	N/A	N/A	46.94	46.94	N/A	N/A	N/A	N/A
SVM Classifier	N/A	N/A	46.94	46.94	N/A	N/A	N/A	N/A

Table 5.4: Tree experiment results using Unsorted Max Segmentation and hashing

Model	**Training Accuracy %**	**Validation Accuracy %**	**Training Loss**	**Validation Loss**
MLP with ReLu	51.56	54.55	1.75	N/A
MLP with Tanh	51.56	54.55	0.69	N/A
MLP with SoftMax	51.56	54.55	0.69	N/A
MLP with Sigmoid	51.56	54.55	0.70	N/A
LSTM with ReLu	47.66	54.55	7.71	7.71
LSTM with Tanh	48.44	45.45	7.71	7.71
LSTM with SoftMax	50.78	54.55	0.69	0.69
LSTM with Sigmoid	51.56	54.55	0.69	0.69
GRU with ReLu	52.34	54.55	7.71	7.71
GRU with Tanh	55.47	45.45	7.71	7.71
GRU with SoftMax	51.56	54.55	0.70	0.69
GRU with Sigmoid	51.56	54.55	0.69	0.69
SRNN with ReLu	49.22	45.45	7.71	7.71
SRNN with Tanh	50.78	54.55	7.71	7.71
SRNN with SoftMax	54.69	54.55	0.69	0.69
SRNN with Sigmoid	42.97	54.55	0.75	0.69
Dense with ReLu	51.56	54.55	0.69	0.69
Dense with Tanh	41.41	36.36	7.71	7.71
Dense with SoftMax	51.56	54.55	7.71	7.71
Dense with Sigmoid	42.97	54.55	0.69	0.69
Gaussian NB	N/A	48.75	N/A	N/A
RF Classifier	N/A	45.45	N/A	N/A
SGD Classifier	N/A	54.55	N/A	N/A
SVM Classifier	N/A	54.55	N/A	N/A

Table 5.5: Tree experiment results using unsorted max segmentation and unhashing

Model	**Training Accuracy (%)**	**Validation Accuracy (%)**	**Training Loss**	**Validation Loss**
MLP with ReLu	52.66	57.58	495.12	N/A
MLP with Tanh	49.30	48.48	0.91	N/A
MLP with SoftMax	49.22	42.42	0.77	N/A
MLP with Sigmoid	50.78	57.58	Nan	N/A
LSTM with ReLu	56.25	75.76	4.05	3.33
LSTM with Tanh	60.94	70.00	2..01	2.49
LSTM with SoftMax	72.66	63.64	0.53	0.64
LSTM with Sigmoid	77.34	70.00	0.50	0.56
GRU with ReLu	71.88	66.67	0.62	1.48
GRU with Tanh	53.12	51.52	0.67	0.92
GRU with SoftMax	74.22	60.61	0.56	0.69
GRU with Sigmoid	76.56	75.76	0.53	0.57
SRNN with ReLu	61.72	48.48	4.32	5.63
SRNN with Tanh	71.09	66.67	1.92	2.54
SRNN with SoftMax	100	60.61	0.07	0.89
SRNN with Sigmoid	100	70.00	0.16	0.69
Dense with ReLu	50.78	57.58	7.71	7.71
Dense with Tanh	50.78	57.58	0.32	1.57
Dense with SoftMax	96.88	75.76	0.69	0.69
Dense with Sigmoid	97.66	57.58	0.16	0.80
Gaussian NB	N/A	61.88	N/A	N/A
RF Classifier	N/A	60.61	N/A	N/A
SGD Classifier	N/A	81.82	N/A	N/A
SVM Classifier	N/A	51.52	N/A	N/A

6 Legal, Social, Ethical and Professional Issues

6.1 Legal Issues

The research done in this project has been carried out in compliance with all the applicable laws and regulations. Where third-party work is made use of or applied, it has been referenced, clearly stating the source of the work. This project references third-party work and uses images collected from third-party websites. Where this is the case, credit is given according to professional standards.

6.2 Social and Ethical Issues

To my knowledge, the only social and ethical issue that could arise from this project is the development of neural networks to process and understand source code in a context where the source code being learnt is closed source, i.e., the developer has not been permitted to use that program for research or testing purposes. Using any of the ideas presented in this project in such a situation is unethical and is unintended in all cases.

6.3 Professional Issues

The professional issues this project could solve include building specific neural networks and the development of deep learning techniques to understand source code directly from the program's AST without needing to convert it into trees or graphs and embedding the nodes. This could be done by learning how compilers and interpreters process source code from the program AST at runtime and applying this knowledge to build an entirely new type of neural network architecture.

This development would see a huge leap forward in Programming Language Understanding (PLU). It would provide entirely new ways for programs to be understood by deep learning methods. It could even lead to further research interest in the field of PLU.

Additionally, this project is an addition to the PLU field that compares two ways of representing programs to classify them.

7 CONCLUSION

In conclusion, with the results of this project, I have determined if trees are more suited than graphs for representing source code which is to be learnt by a neural network. Furthermore, I have also achieved each specific objective outlined in section 3.1.

7.1 Specific Objectives Achieved

Objective 2: This project has successfully implemented a method for converting source code into a graph represented by a NetworkX Digraph object (section 4.2.1).

Objective 3: The method for converting the graph nodes from Objective 2 into node embeddings has been completed (section 4.2.2).

Objective 5: A method for converting a program into an AST and extracting the complex and important information in the program into a tree data structure has been implemented (section 4.2.1).

Objective 6: I designed a method for converting the tree data structure into a list of node embeddings that can be understood by a neural network (section 4.2.2).

Objectives 1, 4 and 7: I aimed to build 9 DL and NDL models to compare their results when trained and tested using source code represented with three different data structures (trees, graphs, and text), and I have achieved this aim (section 4.2.4).

Objective 8: I have compared the results of running the models from Objectives 1, 4 and 7 with the results of Objectives 3 and 6 and text embeddings to determine the most suitable for classifying programs based on the sorting algorithm they implement (section 4.5).

7.2 Future Work

One possible future improvement on this project could be to examine how an individual programmer's programming style (e.g., do they always use camelCase or snake_case, etc.) influences the ability of a deep learning model to understand the code. Another example could be to build a neural network model that can describe a program based on the functions it implements. Finally, work could be done to implement neural networks and machine learning models that can understand multiple programming languages. This could lead to models being trained in one language and tested in another. This does not have to be limited to deep learning models.

8 REFERENCES

[1] McCaffrey, J., 2006. *Scripting Languages vs. Non-Scripting Languages in Test Automation*. James D. McCaffrey. https://jamesmccaffrey.wordpress.com/2006/10/06/scripting-languages-vs-non-scripting-languages-in-test-automation/

[2] En.wikipedia.org. n.d. *Scripting language - Wikipedia*. https://en.wikipedia.org/wiki/Scripting_language

[3] En.wikipedia.org. n.d. *Procedural programming - Wikipedia*. https://en.wikipedia.org/wiki/Procedural_programming

[4] GeeksforGeeks. n.d. Introduction to Tree Data Structure - GeeksforGeeks. https://www.geeksforgeeks.org/introduction-to-tree-data-structure/

[5] Allamanis, M., Brockschmidt, M. and Khademi, M., 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*.

[6] Zola, A., n.d. *What is a strongly typed programming language?*. https://www.techtarget.com/whatis/definition/strongly-typed

[7] Nadkarni, P.M., Ohno-Machado, L. and Chapman, W.W., 2011. Natural language processing: an introduction. *Journal of the American Medical Informatics Association*, *18*(5), pp.544-551.

[8] Beck, D., Haffari, G. and Cohn, T., 2018. Graph-to-sequence learning using gated graph neural networks. *arXiv preprint arXiv:1806.09835*.

[9] Mou, L., Li, G., Zhang, L., Wang, T. and Jin, Z., 2016, February. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI conference on artificial intelligence*.

[10] Browne, M. and Ghidary, S.S., 2003, December. Convolutional neural networks for image processing: an application in robot vision. In *Australasian Joint Conference on Artificial Intelligence* (pp. 641-652). Springer, Berlin, Heidelberg.

[11] Browne, M., Ghidary, S.S. and Mayer, N.M., 2008. Convolutional neural networks for image processing with applications in mobile robotics. In *Speech, Audio, Image and Biomedical Signal Processing using Neural Networks* (pp. 327-349). Springer, Berlin, Heidelberg.

[12] Van Emden, E. and Moonen, L., 2002, October. Java quality assurance by detecting code smells. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* (pp. 97-106). IEEE.

[13] Alon, U., Zilberstein, M., Levy, O. and Yahav, E., 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, *3*(POPL), pp.1-29.

[14] Scarselli, F., Gori, M., Tsoi, A.C., Hagenbuchner, M. and Monfardini, G., 2008. The graph neural network model. *IEEE transactions on neural networks*, *20*(1), pp.61-80.

[15] Li, Y., Tarlow, D., Brockschmidt, M. and Zemel, R., 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*.

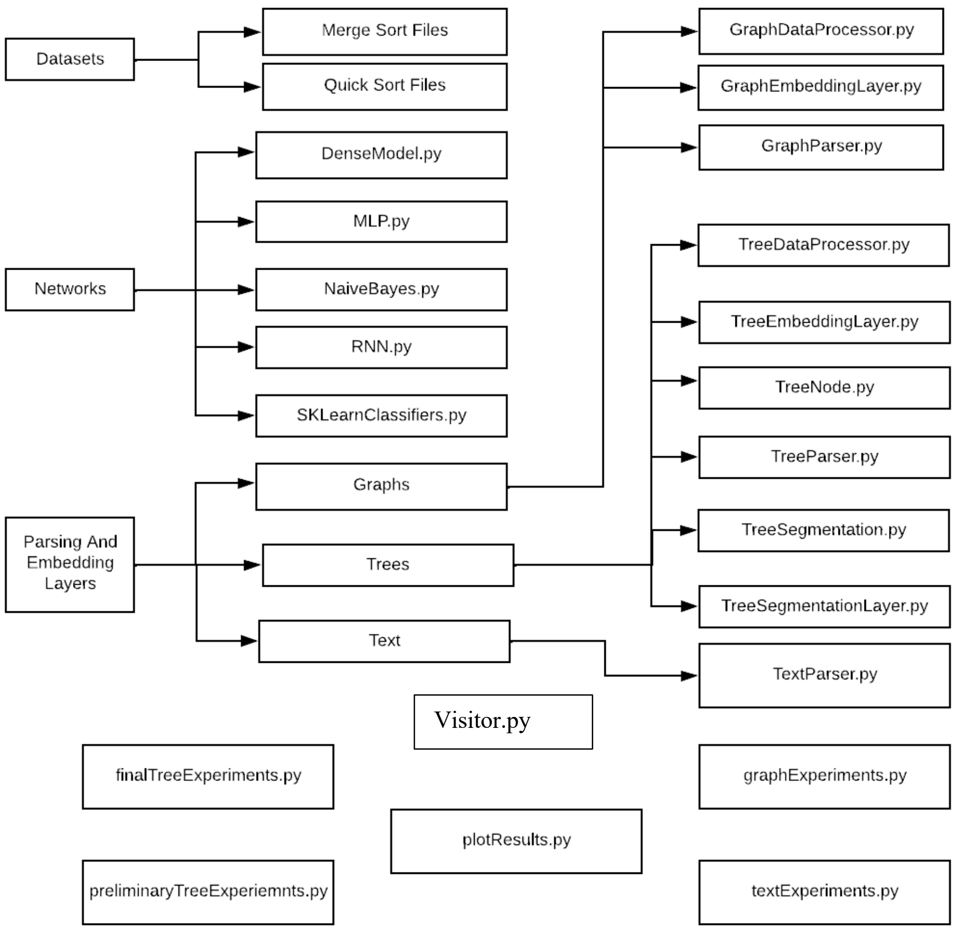
[16] Shi, Y., Li, Q. and Zhu, X.X., 2020. Building segmentation through a gated graph convolutional neural network with deep structured feature embedding. *ISPRS Journal of Photogrammetry and Remote Sensing*, *159*, pp.184-197.
[17] Ruiz, L., Gama, F. and Ribeiro, A., 2020. Gated graph recurrent neural networks. *IEEE Transactions on Signal Processing*, *68*, pp.6303-6318.
[18] GitHub. n.d. *Build software better, together*. https://github.com/search?l=Python&q=quick+sort&type=Repositories
[19] GitHub. n.d. *Build software better, together*. https://github.com/search?l=Python&q=merge+sort&type=Repositories
[20] Ramchoun, H., Ghanou, Y., Ettaouil, M. and Janati Idrissi, M.A., 2016. Multilayer perceptron: Architecture optimization and training.
[21] Bebis, G. and Georgiopoulos, M., 1994. Feed-forward neural networks. *IEEE Potentials*, *13*(4), pp.27-31.
[22] Kanal, L.N., 2003. Perceptron. In *Encyclopedia of Computer Science* (pp. 1383-1385).
[23] Hopfield, J.J., 1988. Artificial neural networks. *IEEE Circuits and Devices Magazine*, *4*(5), pp.3-10.
[24] Hecht-Nielsen, R., 1992. Theory of the back-propagation neural network. In *Neural networks for perception* (pp. 65-93). Academic Press.
[25] Verma, Y., 2021. *A Complete Understanding of Dense Layers in Neural Networks*. Analytics India Magazine. https://analyticsindiamag.com/a-complete-understanding-of-dense-layers-in-neural-networks/
[26] Hochreiter, S., 1998. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, *6*(02), pp.107-116.
[27] Schuster, M. and Paliwal, K.K., 1997. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, *45*(11), pp.2673-2681.
[28] Olah, C., 2015. *Understanding LSTM Networks -- colah's blog*. Colah.github.io. http://colah.github.io/posts/2015-08-Understanding-LSTMs/
[29] En.wikipedia.org. n.d. *Gated recurrent unit - Wikipedia*. https://en.wikipedia.org/wiki/Gated_recurrent_unit
[30] Dey, R. and Salem, F.M., 2017, August. Gate-variants of gated recurrent unit (GRU) neural networks. In *2017 IEEE 60th international midwest symposium on circuits and systems (MWSCAS)* (pp. 1597-1600). IEEE.
[31] Mikolov, T., Karafiát, M., Burget, L., Cernocký, J. and Khudanpur, S., 2010, September. Recurrent neural network based language model. In *Interspeech* (Vol. 2, No. 3, pp. 1045-1048).
[32] Abdelrahman, M.M.H.M., 2019. Analyzing robustness of models of chaotic dynamical systems learned from data with Echo state networks
[33] En.wikipedia.org. n.d. *Bayes' theorem - Wikipedia*. https://en.wikipedia.org/wiki/Bayes%27_theorem
[34] Brownlee, J., 2018. *A Gentle Introduction to k-fold Cross-Validation*. Machine Learning Mastery. https://machinelearningmastery.com/k-fold-cross-validation/
[35] En.wikipedia.org. n.d. *Random forest - Wikipedia*. https://en.wikipedia.org/wiki/Random_forest#/media/File:Random_forest_diagram_complete.png
[36] Kapil, D., 2019. *Stochastic vs Batch Gradient Descent*. Medium. https://medium.com/@divakar_239/stochastic-vs-batch-gradient-descent-8820568eada1

[37] Pupale, R., 2018. *Support Vector Machines(SVM) — An Overview*. Medium. https://towardsdatascience.com/https-medium-com-pupalerushikesh-svm-f4b42800e989
[38] Rentsch, T., 1982. Object oriented programming. *ACM Sigplan Notices*, *17*(9), pp.51-57.
[39] scikit-learn. n.d. *sklearn.feature_extraction.text.CountVectorizer*. https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html
[40] Docs.python.org. n.d. *ast — Abstract Syntax Trees — Python 3.10.5 documentation*. https://docs.python.org/3/library/ast.html
[41] Networkx.org. n.d. *NetworkX — NetworkX documentation*. https://networkx.org
[42] Kural, M., 2005. Tree traversal and word order. *Linguistic Inquiry*, *36*(3), pp.367-387.
[43] Minaee, S., Boykov, Y.Y., Porikli, F., Plaza, A.J., Kehtarnavaz, N. and Terzopoulos, D., 2021. Image segmentation using deep learning: A survey. *IEEE transactions on pattern analysis and machine intelligence*.
[44] Landsberg, J.M., 2012. Tensors: geometry and applications. *Representation theory*, *381*(402), p.3.
[45] Denoyer, L. and Gallinari, P., 2014. Deep sequential neural network. *arXiv preprint arXiv:1410.0510*.
[46] Brownlee, J., 2019. *How to Control the Stability of Training Neural Networks With the Batch Size*. Machine Learning Mastery. https://machinelearningmastery.com/how-to-control-the-speed-and-stability-of-training-neural-networks-with-gradient-descent-batch-size/
[47] Brownlee, J., 2018. *Difference Between a Batch and an Epoch in a Neural Network*. Machine Learning Mastery. https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/
[48] Scikit-learn.org. n.d. *scikit-learn: machine learning in Python — scikit-learn 1.1.1 documentation*. https://scikit-learn.org/stable/
[49] scikit-learn. n.d. *sklearn.pipeline.Pipeline*. https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html
[50] Wang, Q., Ma, Y., Zhao, K. and Tian, Y., 2022. A comprehensive survey of loss functions in machine learning. *Annals of Data Science*, *9*(2), pp.187-212.
[51] Saxena, S., 2021. *Binary Cross Entropy/Log Loss for Binary Classification*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2021/03/binary-cross-entropy-log-loss-for-binary-classification/

9 APPENDICES

9.1 Appendix A: Source Code File Structure

The structure of the files described in the implementation section of this report are displayed below:



- The Datasets file contains the training and testing data files. These are divided into Merge Sort and Quick Sort.

- The Networks file contains the implementations of the nine DL and NDL models used for the classification tasks. This folder contains 5 different files. One each for the Dense Model, the MLP model and the Naïve Bayes classifier. 'RNN.py' contains the implementations for the LSTM, the GRU and the Simple RNN. 'SKLearnClassifiers.py' contains the implementations for the SGD, SVM and RF classifiers.

- Parsing And Embedding Layers is divided into three subfolders.

  - Graphs: Contains all the files for running the background operations for the graph data structure. These files are 'GraphDataProcessor.py', 'GraphEmbeddingLayer.py' and 'GraphParser.py'. The contents and functions of these files have been described in detail in section 4.2.

  - Trees: This holds the files for running the background operations on the abstract tree data structure. The files here are 'TreeEmbeddingLayer.py', 'TreeDataProcessor.py', 'TreeNode.py', 'Treeparser.py', 'TreeSegmentation.py' and 'TreeSegmentationLayer.py'.

- The remaining files are for running the experiments and plotting their results. These are:

  - 'Visitor.py': This is where the AbstractVisitor, HashVisitor and Visitor are implemented.

  - 'preliminaryTreeExpeirments.py': This file implements the preliminary experiments for determining which segmentation function provides the best results. It uses the MLP to make this decision.

  - 'finalTreeExpeirments.py': This is where the final experiments for getting the results on the trees are contained.

  - 'graphExperiments.py': This file is where I implement the experiments on the graph-based data using all 9 models.

  - 'textExperiments.py': This file is the experiments on text are carried out.

  - 'plotResults.py': This file contains the code for plotting the graphs seen in *fig 5.1* and *5.2*.

  To run the experiments in the files above, open the file and click F5 (fn + F5 with a MacBook) or select the debug file option in the project menu.

9.2 Appendix B: Source Code Files

The project files outlined above have their source code shown in this section with the appropriate names.

9.2.1 Networks

9.2.1.1 DenseModel.py

```
import tensorflow as tf

def runDenseModel(x_train, y_train, x_test, y_test, activation: str, batchSize:
int, epochs: int, filename: str = None):
    """ The method to run the model made up of densely connected layers
    x_train - The training data
    y_train - The training data labels
    x_test - The testing data
    y_test - The testing data labels
    activation: str -  The activation function to be applied
    batchSize: int - The size of each batch for the model
    epochs: int - The number of epochs
    filename: str = None - THe name of the file to be saved """
    model = tf.keras.models.Sequential()
    #Insert the input layer
    model.add(tf.keras.layers.InputLayer(input_shape=(x_train.shape[1], )))

    # Add the densely connected layers
    model.add(tf.keras.layers.Dense(256, activation=activation))
    model.add(tf.keras.layers.Dense(64, activation=activation))
    model.add(tf.keras.layers.Dense(2, activation=activation))

    # Compile and train/fit the model
    model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
    model.summary()
    model.fit(x_train, y_train, epochs=epochs, batch_size=batchSize,
validation_data=(x_test, y_test))
    if filename is not None:
        model.save(filename)
```

9.2.1.2 MLP.py

```
import tensorflow as tf
import numpy as np
from typing import List

class MLP:
    def __init__(self, x_train: tf.Tensor, y_train: List, layers: List[int],
                 activationFunction: str, learningRate: float, epochs: int):
        """ The Multi Layer Perceptron Class
        x_train: tf.Tensor - The training data
        y_train: List - The training data labels
        layers: List[int] - The list of layer units
        activationFunction: str - The activation function to be used
        learningRate: float - The learning rate
        epochs: int - The number of times the data is passed back and forth in the
network"""
        self.x_train = x_train
        self.y_train = y_train
        self.layers = layers
        self.layerCount = len(self.layers)
        self.xCount = len(self.x_train)
        self.activationFunction = self.getActivationFunction(activationFunction)
        self.learningRate = learningRate
        self.epochs = epochs
        self.weights, self.bias, self.weightDeltas, self.biasDeltas = {}, {}, {},
{}
        self.hiddenLayerCount = len(layers)-2
        self.featureCount = layers[0]
        self.classCount = layers[-1]
        self.parameterCount = 0

        self.initialiseWeights()

        # Calculate the number of parameters in the model
        for i in range(1, self.layerCount):
            self.parameterCount += self.weights[i].shape[0] *
self.weights[i].shape[1]
            self.parameterCount += self.bias[i].shape[0]

        # Print a summary of the model
        print(self.featureCount, "features,", self.classCount, "classes,",
self.parameterCount, "parameters, and", self.hiddenLayerCount, "hidden layers",
"\n")
        for i in range(1, self.layerCount-1):
            print('Hidden layer {}:'.format(i), '{}
neurons'.format(self.layers[i]))

    def getActivationFunction(self, activationFunction: str):
        """ The method to seelct an activation function
```

```
        activationFunction: str - The string representation of the activation
function"""
        if activationFunction == 'softmax':
            return tf.nn.softmax
        elif activationFunction == 'relu':
            return tf.nn.relu
        elif activationFunction == 'tanh':
            return tf.tanh
        elif activationFunction == 'sigmoid':
            def logSigmoid(x):
                x = 1.0/(1.0 + tf.math.exp(-x))
                return x
            return logSigmoid
        else:
            return None

    def lossFunction(self, outputs: tf.Tensor, yValues: tf.Tensor):
        """ Calculate the loss between the actual and predicted outputs """
        return tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(yValues,
outputs))

    def makePrediction(self, x_test: tf.Tensor):
        """Make predictions on the unseen testing data
        x_test: tf.Tensor - The unseen testing data in tensor format"""
        predictions = []
        output = self.FFLayer(x_test)
        prediction = tf.argmax(tf.nn.softmax(output), axis=1)
        predictions.append(prediction)
        return tf.convert_to_tensor(predictions)

    def initialiseWeights(self):
        """Initialise the weights and biases for the model"""
        for i in range(1, self.layerCount):
            self.weights[i] = tf.Variable(tf.random.normal(shape=(self.layers[i],
self.layers[i-1])))
            self.bias[i] = tf.Variable(tf.random.normal(shape=(self.layers[i], 1)))

    def FFLayer(self, x: tf.Tensor):
        """ The Feed Forward layer for the model
        x: tf.Tensor - The data to be passed across the layers """
        for i in range(1, self.layerCount):
            x = tf.matmul(x, tf.transpose(self.weights[i])) +
tf.transpose(self.bias[i])
            x = self.activationFunction(x)
        return x

    def backPropagate(self, xValues: tf.Tensor, yValues: tf.Tensor):
        """ Run the backpropagation algorithm on the training data
        xValues: tf.Tensor - The training data
        yValues: tf.Tensor - The training data labels
        """
```

```
        with tf.GradientTape(persistent=True) as tape:
            output = self.FFLayer(xValues)
            loss = self.lossFunction(output, yValues)

        for i in range(1, self.layerCount):
            tape.watch(self.weights[i])
            # calcualte the gradients of the weights and biases
            self.weightDeltas[i] = tape.gradient(loss, self.weights[i])
            self.biasDeltas[i] = tape.gradient(loss, self.bias[i])

        del tape
        # update the weights after calculating the gradients
        self.updateWeights()
        return loss.numpy()

    def updateWeights(self):
        """The method that updates the weights and biases after backpropagation
        has been carried out"""
        for i in range(1, self.layerCount):
            if self.weightDeltas[i] is not None:
                # Subtract the learning rate * the gradient from the current
weight/bias
                self.weights[i].assign_sub(self.learningRate *
self.weightDeltas[i])

            if self.biasDeltas[i] is not None:
                self.bias[i].assign_sub(self.learningRate * self.biasDeltas[i])

    def runFFModel(self, x_train: tf.Tensor, y_train: tf.Tensor, x_test: tf.Tensor,
y_test: tf.Tensor):
        """ The method to implement the MLP
        x_train: tf.Tensor - The training data
        y_train: tf.Tensor - The training data labels
        x_test: tf.Tensor - The testing data
        y_test: tf.Tensor - The testing data labels"""
        index = 0
        loss = 0.0
        metrics = {'trainingLoss': [], 'trainingAccuracy': [],
'validationAccuracy': []}
        self.initialiseWeights()
        for i in range(self.epochs):
            print('Epoch {}'.format(i), end='........')
            predictions = []

            if index % 5 == 0:
                print(end=".")
            if index >= len(y_train):
                index = 0

            # First forward pass
            loss = self.backPropagate(x_train, y_train)
```

```
            # Second forward pass/Recurrent Loop with the updated weights
            newOutputs = self.FFLayer(x_train)

            # run softmax activation on the outputs to get the final prediction
            pred = tf.argmax(tf.nn.softmax(newOutputs), axis = 1)
            predictions.append(pred)
            index += 1

            predictions = tf.convert_to_tensor(predictions)

            # add the calculated loss to the list of metrics
            metrics['trainingLoss'].append(loss)

            # make predictions on the unseen testing data
            unseenPredictions = self.makePrediction(x_test)

            # Calculate the accuracies of the seen and unseen predictions
            metrics['trainingAccuracy'].append(np.mean(np.argmax(y_train, axis=1)
== predictions.numpy()))
            metrics['validationAccuracy'].append(np.mean(np.argmax(y_test, axis=1)
== unseenPredictions.numpy()))
            print('\tLoss:', metrics['trainingLoss'][-1], 'Accuracy:',
metrics['trainingAccuracy'][-1],
                  'Validation Accuracy:', metrics['validationAccuracy'][-1])
        return metrics
```

9.2.1.3 NaiveBayes.py

```
import random, math
from statistics import mean

class NBClassifier:
    def __init__(self, x: list, y: list):
        """A Naïve Bayes classifier model with cross-validation
        x: list - The x values (training data)
        y: list - The y values (training data labels)"""
        self.x = x
        self.y = y
        self.crossValFolds = 5 #the number of folds
        self.crossValFoldSize = int(len(self.x)/self.crossValFolds) # the size of
each fold

    def separationFunction(self, xValues: list, yValues: list):
        """ Separate the data based on classes to get a summary of the data
        xValues: list - The data
        yValues: list - The data labels """
        separatedDict = {} #separate the data into a dictionary with the labels as
the keys
```

```
        for i in range(len(xValues)):
            x = xValues[i]
            label = yValues[i]
            if label not in separatedDict:
                separatedDict[label] = []
            separatedDict[label].append(x)
        return separatedDict

    def splitIntoFolds(self, xValues, yValues):
        """Split the data and the labels into folds for cross-validation"""
        splitFolds, xVals = [], xValues #the folds for the data
        yFolds, yTrain = [], yValues # the folds for the labels
        for i in range(self.crossValFolds):
            currentFold, yFold = [], []
            while len(currentFold) < self.crossValFoldSize: #make sure the folds
are all the same size
                j = random.randrange(len(xVals)) #randomly select data to be in
each fold
                currentFold.append(xVals.pop(j))
                yFold.append(yTrain.pop(j)) #repeat for the label
            splitFolds.append(currentFold)
            yFolds.append(yFold)
        return splitFolds, yFolds

    def calculateAccuracyScore(self, values: list, predictions: list):
        """Get the overall accuracy score of the model
        values: list - the actual y values
        predictions: list - the predicted y values"""
        accuracyScore = 0.0
        for i in range(len(values)):
            if values[i] == predictions[i]:
                accuracyScore += 1.0

        accuracyScore = accuracyScore/float(len(values)) * 100.0
        return accuracyScore

    def getMean(self, values: list):
        """Get the mean of a set of values
        values: list - The lsit of values from which to calculate the mean"""
        return float(mean(values))

    def getStandardDev(self, values: list):
        """Calculate the standard deviation of a set of values
        values: list - The lsit of values from which to calculate the standard
deviation"""
        meanValue = self.getMean(values)
        var = sum([(i-meanValue) * (i-meanValue) for i in values]) /
float(len(values)-1)
        return math.sqrt(var)

    def collateStatistics(self, values: list):
```

```
        """Get the statistics for a set of values
        values: list - The list from which the statistics are to be collected"""
        return [(self.getMean(i), self.getStandardDev(i), len(i)) for i in
zip(*values)]

    def collateClassStatistics(self, xValues: list, yValues: list):
        """Get the statistics for an individual class
        xValues: list - The class data
        yValues: list - The class labels"""
        separatedDict = self.separationFunction(xValues, yValues) #separate the
data by class
        classStats = {}

        # collate the statistics for each separate class
        for i, j in separatedDict.items():
            classStats[i] = self.collateStatistics(j)

        return classStats

    def getGaussianProbability(self, val, valMean, valStd):
        """Calculate the Gaussian probability for a set of values
        val - Each individual value
        valMean - The mean of the set of values from which 'val' came from
        valStd - The standard deviation of the set of values from which 'val' came
from"""
        distFromMean = (val-valMean)  * (val-valMean) #the distance between the
value and its mean
        if valStd == 0:
            valStd = 1 # if the standard deviation is 0, set it to 1 so it can be
multiplied and divided

        # calculate the Gaussian probablilty using the above values
        stdSqr = 2 * valStd * valStd
        e = distFromMean/stdSqr
        piSqrt = 1/(math.sqrt(2 * math.pi) * valStd)
        gaussianProb = math.exp(-(e)) * (piSqrt)

        return gaussianProb

    def getClassGaussianProbability(self, classStats, values):
        """Get the Gaussian probability of a class of values
        classStats - The statistics of the individual class
        values - The values in the class
        """
        rowSum, probabilities = sum(classStats[y][0][2] for y in classStats), {}
        for value, stat in classStats.items():
            probabilities[value] = classStats[value][0][2]/float(rowSum)
            for i in range(len(stat)):
                valMean, valStd, valCount = stat[i]
                # get the probability of a value belonging to the current class
```

```
                probabilities[value] *= self.getGaussianProbability(values[i], valMean, valStd)

        return probabilities

    def makePrediction(self, classStats, xValue):
        """Make a prediction on a value in the testing fold based on its properties
        classStats - The statistics of all the classes
        xValue - The value who's class is to be predicted
        """
        # Get the gaussian probabilities of the class based on its mean and standard deviation
        classProbabilities = self.getClassGaussianProbability(classStats, xValue)
        label, probability = 10, -1
        classProbs = classProbabilities.items()
        for value, prob in classProbs:
            if label == 10 or prob > probability:
                probability = prob #select the highest probability as the final prediction
                label = value
        return label

    def runGNBClassifier(self, xTrain: list, yTrain: list, xTest: list):
        """Make classifications and predictions
        xTrain: list - The training data folds
        yTrain: list - The training label folds
        xTest: list - The testing data fold"""
        # Get the mean and standard deviation for the training folds
        classStats = self.collateClassStatistics(xTrain, yTrain)
        predictions = []
        for x in xTest:
            # make the predictions on the testing data fold
            prediction = self.makePrediction(classStats, x)
            predictions.append(prediction)

        return predictions

    def gaussianCrossValidation(self, xValues: list, yValues: list):
        """Run the cross-validation algorithm on all the data
        xValues: list - The data
        yValues: list - The data class labels"""
        # split the data into folds of equal values:
        foldsX, foldsY = self.splitIntoFolds(xValues, yValues)
        accuracyScores = []

        # for each data fold, separate it from the rest of the folds
        # use the other folds as training data and use the current fold as testing data
        # calculate the and return the mean accuracy score when the prediction process is complete.
        for i in range(len(foldsX)):
```

```
            currentTrainX, currentY = foldsX[i], foldsY[i]
            allTrain, allY, allTest, allTestY = list(foldsX), list(foldsY), [], []

            allY.remove(currentY)
            allTrain.remove(currentTrainX)
            allTrain = sum(allTrain, [])
            allY = sum(allY, [])
            for j in range(len(currentTrainX)):
                trainY = currentY[j]
                trainX = currentTrainX[j]
                allTest.append(trainX)
                allTestY.append(trainY)

            predicted = self.runGNBClassifier(allTrain, allY, allTest)
            accuracyScore = self.calculateAccuracyScore(allTestY, predicted)
            accuracyScores.append(accuracyScore)
        return mean(accuracyScores)
```

9.2.1.4 RNN.py

```
import tensorflow as tf

class RNN:
    def __init__(self, layerName: str, x_train: tf.Tensor, y_train: tf.Tensor,
    x_test: tf.Tensor, y_test: tf.Tensor, activationFunction: str = None,
    neurons:int = None, dropoutRate: float =  None):
        """The RNN class from where all RNN layers are called
        layerName: str - The string representation of the type of layer that has
been selected
        x_train: tf.Tensor - The training data
        y_train: tf.Tensor - The training data labels
        x_test: tf.Tensor - The testing data
        y_test: tf.Tensor - The testing data labels
        activationFunction: str = None - The activation function as a string
        neurons:int = None - The number of units each layer is to have
        dropoutRate: float =  None - The dropout rate"""
        self.layerName = layerName.lower()

        self.x_train = x_train
        self.y_train = y_train
        self.x_test = x_test
        self.y_test = y_test
        if activationFunction is not None:
            self.activationFunction =
self.getActivationFunction(activationFunction)

        if neurons is not None:
            self.neurons = neurons
```

```
        if dropoutRate is not None:
            self.dropoutRate = dropoutRate


    def RNNLayer(self, neurons: int, activationFunction: str, useBias: bool,
inputShape, returnSequences):
        """Get the Simple RNN layer"""
        activationFunction = self.getActivationFunction(activationFunction)
        return tf.keras.layers.SimpleRNN(neurons, activation=activationFunction,
use_bias=useBias, return_sequences=returnSequences, input_shape=inputShape)

    def LSTMLayer(self, neurons: int, activationFunction: str, useBias: bool,
inputShape, returnSequences):
        """Get the LSTM layer"""
        activationFunction = self.getActivationFunction(activationFunction)
        return tf.keras.layers.LSTM(neurons, activation=activationFunction,
use_bias=useBias, return_sequences=returnSequences, input_shape=inputShape)

    def GRULayer(self, neurons: int, activationFunction: str, useBias: bool,
inputShape, returnSequences):
        "Get the GRU layer"
        activationFunction = self.getActivationFunction(activationFunction)
        return tf.keras.layers.GRU(neurons, activation=activationFunction,
use_bias=useBias, return_sequences=returnSequences, input_shape=inputShape)

    def DropoutLayer(self, dropoutRate: float):
        """Get the Dropout Layer
        dropoutRate: float - The probability of a weight being dropped in the range
0 - 1
        """
        return tf.keras.layers.Dropout(dropoutRate)

    def DenseLayer(self, neurons: int, useBias: bool):
        """Get a densely connected layer to be used as the output layer
        neurons: int - The number of units the layer should have
        useBias: bool - Whether or not to use a bias"""
        activationFunction = self.activationFunction
        return tf.keras.layers.Dense(neurons, activationFunction, useBias)

    def getActivationFunction(self, activationFunction: str):
        """Retrieve the activation function based on the input to the method
        activationFunction: str - A string representation of the activation
function to be retrieved"""
        if activationFunction == 'softmax':
            return tf.nn.softmax
        elif activationFunction == 'relu':
            return tf.nn.relu
        elif activationFunction == 'tanh':
            return tf.tanh
        elif activationFunction == 'sigmoid':
            def logSigmoid(x):
```

```
                x = 1.0/(1.0 + tf.math.exp(-x))
                return x
            return logSigmoid
        else:
            return None

    def runModel(self, layerType: str, neurons: int, epochs: int, batchSize: int,
filename: str = None):
        """ Run the RNN model
        layerType: str - The type of layer: LSTM, GRU or RNN
        neurons: int - The number of units for the chosen layer type
        epochs: int - THe number of times the data will be passed back and forth in
the network
        batchSize: int - The size of each training batch in the model
        filename: str = None - The name of the file to save the model into
        """
        print("USING THE RECURRENT NEURAL NETWORK")

        inputLayer = tf.keras.layers.InputLayer(input_shape=(self.x_train.shape[1],
1))
        inputShape=(self.x_train.shape[0], )

        dropout = self.DropoutLayer(0.3)
        output = self.DenseLayer(2, False)

        print("USING", layerType.upper(), "LAYERS")
        model = tf.keras.models.Sequential()
        model.add(inputLayer)
        if layerType == "lstm":
            print("USING LSTM LAYERS")
            lstmLayer = self.LSTMLayer(neurons, self.activationFunction, True,
inputShape, True)
            model.add(tf.keras.layers.Bidirectional(lstmLayer))
            model.add(tf.keras.layers.LSTM(256))
        elif layerType == "gru":
            print("USING GRU LAYERS")
            gruLayer = self.GRULayer(neurons, self.activationFunction, False,
inputShape, True)
            model.add(tf.keras.layers.Bidirectional(gruLayer))
            model.add(tf.keras.layers.GRU(10))
        elif layerType =="rnn":
            print("USING SRNN LAYERS")
            rnnLayer = self.RNNLayer(neurons, self.activationFunction, False,
inputShape, True)
            model.add(tf.keras.layers.Bidirectional(rnnLayer))
            model.add(tf.keras.layers.SimpleRNN(256))

        model.add(output)
        model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
        model.summary()
```

```
        model.fit(self.x_train, self.y_train, epochs=epochs, batch_size=batchSize,
validation_data=(self.x_test, self.y_test))
        if filename is not None:
            model.save(filename)
```

9.2.1.5 SKLearnClassifiers.py

```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import SGDClassifier
from sklearn.svm import LinearSVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

def SGDClassify(x_train: list, y_train: list, x_test: list, y_test: list):
    """A Stochastic Gradient Classifier
    x_train: list - The training data
    y_train: list - The training data labels
    x_test: list - The testing data
    y_test: list - The testing data labels
    """
    classifier = Pipeline([('clf', SGDClassifier())]) #create a classification
pipeline
    classifier.fit(x_train, y_train)

    predictions = classifier.predict(x_test)
    accuracyScore = accuracy_score(y_test, predictions)
    return accuracyScore

def SVMClassify(x_train: list, y_train: list, x_test: list, y_test: list):
    """A Support Vector Machine Classifier
    x_train: list - The training data
    y_train: list - The training data labels
    x_test: list - The testing data
    y_test: list - The testing data labels
    """
    classifier = Pipeline([('clf', LinearSVC())])
    classifier.fit(x_train, y_train)

    predictions = classifier.predict(x_test)
    accuracyScore = accuracy_score(y_test, predictions)
    return accuracyScore

def rfClassify(x_train: list, y_train: list, x_test: list, y_test: list):
    """A RANDOM FOREST CLASSIFIER
    x_train: list - The training data
    y_train: list - The training data labels
    x_test: list - The testing data
    y_test: list - The testing data labels
    """
    classifier = Pipeline([('clf', RandomForestClassifier())])
```

```
    classifier.fit(x_train, y_train)
    predictions = classifier.predict(x_test)
    accuracyScore = accuracy_score(y_test, predictions)
    return accuracyScore
```

9.2.2 Parsing And Embedding Layers

9.2.2.1 Graphs

9.2.2.1.1 GraphDataProcessor.py

```
import tensorflow as tf
from ParsingAndEmbeddingLayers.Graphs.GraphParser import GraphParser
from ParsingAndEmbeddingLayers.Graphs.GraphEmbeddingLayer import
GraphEmbeddingLayer
from os.path import dirname, join

class GraphDataProcessor:
    def __init__(self, hashed: bool):
        """
        A Graph Data Processor Class. This is where all the data for the graph
        data structure is processed before the model is run on the data
        hashed: bool - Whether or not we are working with hashed data
        """
        self.hashed = hashed
        self.parser = GraphParser(self.hashed) #initialise the parser object
        self.segmentCount = 40 #the humber of segments for segmentation

    def splitTrainTest(self, x, matrices, y):
        """
        Split the data into training and testing
        x - The graph data
        matrices - The matrix representation of the graph data
        y - The graph data labels

        returns:
        x_train - The training data
        x_train_matrix - The training data in matrix form
        y_train - The training data labels
        x_test - The testing data
        x_test_matrix - The testing data in matrix form
        y_test - The testing data labels
        """
        split = int(0.7 * len(x)) #split 70-30

        # The training data
        x_train = x[:split]
```

```
        x_train_matrix = matrices[:split]
        y_train = y[:split]

        # The testing data
        x_test = x[split:]
        x_test_matrix = matrices[split:]
        y_test = y[split:]

        return x_train, x_train_matrix, y_train, x_test, x_test_matrix, y_test

    def splitTrainTestNoMatrices(self, x, y):
        """
        An alternative to splitTrainTest that does not involve the use of matrices
        x - The graph data
        y - The graph data labels

        returns:
        x_train - The training data
        y_train - The training data labels
        x_test - The testing data
        y_test - The testing data labels
        """
        split = int(0.7*len(x))

        x_train = x[:split]
        y_train = y[:split]

        x_test = x[split:]
        y_test = y[split:]

        return x_train, y_train, x_test, y_test

    def runSegmentation(self, nodeEmbeddings: tf.Tensor, numSegments: int):
        """
        Run the unsorted segment mean algorithm on the node embeddings
        nodeEmbeddings: tf.Tensor - The set of embeddings from each graph
        numSegments: int - The number of segments to be used

        Returns:
        segFunc - The result of segmentation on the graph
        """
        segments = tf.constant([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
        11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
        22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39])
        segFunc = tf.math.unsorted_segment_mean(nodeEmbeddings, segments,
num_segments = numSegments)
        return segFunc

    def getMaxLen(self, x):
        """
        Get the maximum graph length for padding
```

```
        x - The list of all graphs in both the training and testing data

        Returns:
        maxLen - The number of nodes in the largest available graph
        """
        maxLen = 0
        for i in x:
            if len(i) > maxLen:
                maxLen = len(i)

        return maxLen

    def padGraphs1(self, x, maxLen: int):
        """
        A method to pad the graphs and return a tensor representation of the padded
graphs
        This is intended for use with the deep learning models
        x - The list of graphs to be padded
        maxLen: int - The number of nodes in the largest graph

        Returns:
        x - The padded graphs
        """
        length = len(x)
        for i in range(length):
            if len(x[i]) < maxLen:
                padCount = maxLen - len(x[i])
                x[i] = list(x[i])
                for j in range(padCount):
                    x[i].append(0.0)
            x[i] = tf.convert_to_tensor(x[i])
        return x

    def padGraphs2(self, x, maxLen):
        """
        A method to pad the graphs and return a list representation of the padded
graphs
        This is intended for use with the non deep learning models
        x - The list of graphs to be padded
        maxLen: int - The number of nodes in the largest graph

        Returns:
        x - The padded graphs
        """
        length = len(x)
        for i in range(length):
            if len(x[i]) < maxLen:
                padCount = maxLen - len(x[i])
                for j in range(padCount):
                    x[i].append(0.0)
        return x
```

```
    def runProcessor1(self):
        """
        Processor 1: This processor is run when working with deep learning models
and padded graphs

        Returns:
        x_train - The padded training data in tensor format
        y_train - The training data labels in categorical tensor format
        x_test - The padded testing data in tensor format
        y_test - The testing data labels in categorical tensor format
        """
        # Collect all the required data
        x_train, y_train, x_test, y_test = self.getData()

        total_x = x_train + x_test #combine training and testing to find the
largest graph
        maxLen = self.getMaxLen(total_x)
        # Pad the training and testing graphs
        x_train = self.padGraphs1(x_train, maxLen)
        x_test = self.padGraphs1(x_test, maxLen)

        # Convert the padded graphs to tensors for use in the deep learning models
        x_train = tf.convert_to_tensor(x_train)
        y_train = tf.keras.utils.to_categorical(y_train)
        x_test = tf.convert_to_tensor(x_test)
        y_test = tf.keras.utils.to_categorical(y_test)

        return x_train, y_train, x_test, y_test

    def runProcessor2(self):
        """
        Processor 2: This processor is run when working with non deep learning
models and padded graphs

        Returns:
        xTrain2 - The padded training data in list format
        yTrain - The training data labels in list format
        xTest2 - The padded testing data in list format
        yTest - The testing data labels in list format
        """
        xTrain, yTrain, xTest, yTest = self.getData()

        totalX = xTrain + xTest
        maxLen = self.getMaxLen(totalX)

        xTrain = self.padGraphs2(xTrain, maxLen)
        xTest = self.padGraphs2(xTest, maxLen)

        xTrain2, xTest2 = [], []
```

```
        # Convert the tensors into Python list form
        for i in xTrain:
            xTrain2.append(i)

        for i in xTest:
            xTest2.append(i)
        return xTrain2, yTrain, xTest2, yTest

    def runProcessor3(self):
        """
        Processor 3: This processor is run when working with deep learning models
and segmented graphs

        Returns:
        x_train2 - The segmented training data in tensor format
        y_train - The training data labels in categorical tensor format
        x_test2 - The segmented testing data in tensor format
        y_test - The testing data labels in categorical tensor format
        """
        xTrain, yTrain, xTest, yTest = self.getData()
        x_train2, x_test2 = [], [] #empty list to copy the segmented graphs into

        # Run segmentation on the training data
        for x in xTrain:
            x = tf.convert_to_tensor(x)
            x = [x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x,
            x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
            x = self.runSegmentation(x, self.segmentCount)
            x = tf.reshape(x, (len(x[0]), self.segmentCount))
            x_train2.append(x[0]) #add the segmented graph to a new list

        # Run segmentation on the testing data
        for x in xTest:
            x = tf.convert_to_tensor(x)
            x = [x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x,
            x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
            x = self.runSegmentation(x, self.segmentCount)
            x = tf.reshape(x, (len(x[0]), self.segmentCount))
            x_test2.append(x[0])

        # COnvert all the data and labels into tensors for use in the deep learning
models
        x_train2 = tf.convert_to_tensor(x_train2)
        y_train = tf.keras.utils.to_categorical(yTrain)
        x_test2 = tf.convert_to_tensor(x_test2)
        y_test = tf.keras.utils.to_categorical(yTest)

        return x_train2, y_train, x_test2, y_test

    def runProcessor4(self):
        """
```

```
        Processor 4: This processor is run when working with non deep learning
models and segmented graphs

        Returns:
        xTrain2 - The segmented training data in list format
        yTrain - The training data labels in list format
        xTest2 - The segmented testing data in list format
        yTest - The testing data labels in list format
        """
        xTrain, yTrain, xTest, yTest = self.getData()
        x_train2, x_test2 = [], []
        for x in xTrain:
            x = tf.convert_to_tensor(x)
            x = [x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x,
            x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
            x = self.runSegmentation(x, self.segmentCount)
            x = tf.reshape(x, (len(x[0]), self.segmentCount))
            x_train2.append(x[0])

        for x in xTest:
            x = tf.convert_to_tensor(x)
            x = [x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x,
            x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
            x = self.runSegmentation(x, self.segmentCount)
            x = tf.reshape(x, (len(x[0]), self.segmentCount))
            x_test2.append(x[0])

        xTrain2, xTest2 = [], []
        for i in x_train2:
            #get the actual value and add it to the list to be returned
            xTrain2.append(list(i.numpy()))

        for i in x_test2:
            xTest2.append(list(i.numpy()))
        return xTrain2, yTrain, xTest2, yTest

    def runEmbeddingLayer(self):
        """
        Run the Graph Embedding Layer on each graph/set of nodes
        This is the method that performs 'unhashing' on the nodes
        """
        index = 0 #tracker value
        gp = GraphParser(False) #Create a GraphParser object with hashed = False
        x, x_graph, matrices, labels = gp.readFiles()

        embeddings = [] #an empty list for embeddings
        print("Collecting Graph Embeddings:")
        for graph in x_graph:
            if index % 5 == 0:
                print(end = ".")
            embed = GraphEmbeddingLayer(graph)
```

```
            embeddings.append(embed.vectors)
            index += 1

        # split into training and testing data after embedding
        x_train, y_train, x_test, y_test =
self.splitTrainTestNoMatrices(embeddings, labels)

        # write the data into files for easy retrieval
        self.writeToFiles(x_train, y_train, x_test, y_test)

    def runHashLayer(self):
        """
        The alternative to runEmbeddingLayer that runs the hashing algorithm on the
nodes
        """
        gp = GraphParser(True) #Create a GraphParser object with hashed = True
        x, x_graph, matrices, labels = gp.readFiles()

        embeddings = []
        for graph in x:
            # Hashed = true returns the graph with the label in the final position
so the actual graph is at :-1
            g = graph[:-1]
            embeddings.append(g)

        # split into training and testing data and write to files for easier
processing
        x_train, y_train, x_test, y_test =
self.splitTrainTestNoMatrices(embeddings, labels)
        self.writeToFiles(x_train, y_train, x_test, y_test)

    def runParser(self, processor: int):
        """
        Run the parser object on each processor for the padded graphs
        processor: int - The number of the processor to be selected

        Returns:
        (If processor is 1)
        x_train_graph - The training data in graph form
        x_train_matrix - The training data in matrix form
        y_train - The training data labels
        x_test_graph - The testing data in graph form
        x_test_matrix - The testing data in matrix form
        y_test - The testing data labels

        OR
        (If processor is 2)
        x_train_nodes - The list of nodes from the training data
        y_train - The training data labels
        x_test_nodes - The list of nodes from the testing data
        y_test - The testing data labels
```

```
        """

        x, matrices, labels = self.parser.readFiles()
        x_train_all, x_train_matrix, y_train, x_test_all, x_test_matrix, y_test =
self.splitTrainTest(x, matrices, labels)

        x_train_nodes = [] #empty list for the node representations
        x_train_graph = [] #empty list for the graph representations

        for i in range(len(x_train_all)):
            x_train_nodes.append(x_train_all[i][0]) #get the nodes
            x_train_graph.append(x_train_all[i][1]) #get the entire graph

        x_test_nodes = []
        x_test_graph = []
        for i in range(len(x_test_all)):
            x_test_nodes.append(x_test_all[i][0])
            x_test_graph.append(x_test_all[i][1])

        if processor == 1:
            return x_train_graph, x_train_matrix, y_train, x_test_graph,
x_test_matrix, y_test
        elif processor == 2:
            return x_train_nodes, y_train, x_test_nodes, y_test

    def getFileNames(self):
        """
        Get the names of the files to be read

        Returns:
        xTrain - The name of the file containing the appropriate training data
        yTrain - The name of the file containing the appropriate training data
labels
        xTest - The name of the file containing the appropriate testing data
        yTest - The name of the file containing the appropriate testing data labels
        """
        current_dir = dirname(__file__)
        if self.hashed is False:
            # if hashed is false, return the unhashed file names
            xTrain = join(current_dir, "./Graph Data/graph_x_train.txt")
            yTrain = join(current_dir, "./Graph Data/graph_y_train.txt")
            xTest = join(current_dir, "./Graph Data/graph_x_test.txt")
            yTest = join(current_dir, "./Graph Data/graph_y_test.txt")
        else:
            # if hashed is true, return the hashed file names
            xTrain = join(current_dir, "./Graph Data/graph_x_train_hashed.txt")
            yTrain = join(current_dir, "./Graph Data/graph_y_train_hashed.txt")
            xTest = join(current_dir, "./Graph Data/graph_x_test_hashed.txt")
            yTest = join(current_dir, "./Graph Data/graph_y_test_hashed.txt")

        return xTrain, yTrain, xTest, yTest
```

```
    def writeToFiles(self, x_train, y_train, x_test, y_test):
        """
        Write the embeddings into files
        x_train - The training data to be written
        y_train - The training data labels to be written
        x_test - The testing data to be written
        y_test - The testing data labels to be written
        """
        xTrain, yTrain, xTest, yTest = self.getFileNames() #collect the file names
        with open(xTrain, 'w') as writer:
            for i in x_train:
                writer.write(str(i) + "\n")

        with open(yTrain, 'w') as writer:
            for i in y_train:
                writer.write(str(i) + "\n")

        with open(xTest, 'w') as writer:
            for i in x_test:
                writer.write(str(i) + "\n")

        with open(yTest, 'w') as writer:
            for i in y_test:
                writer.write(str(i) + "\n")

    def readFiles(self, filePath, yFile: bool):
        """
        Read the contents of a file from a given file path
        filePath - The path of the file to be read
        yFile: bool - Whether or not the file contains labels

        Returns:
        values - The dformatted data read from the files
        """
        with open(filePath, 'r') as reader:
            values = reader.readlines()

        # If it is a file of labels, convert into integers form from string
        if yFile is True:
            values = [int(i[0]) for i in values]
        else:
            # if not a file of labels, remove brackets and commas and newlines and
convert bact to floats
            for x in range(len(values)):
                values[x] = values[x].replace("[", "").replace("]", "").strip("\n")
                values[x] = values[x].split(",")
                values[x] = [float(i) for i in values[x]]

        return values
```

```
    def getData(self):
        """
        Get the data for running the models

        Returns:
        x_train - The training data that has been read
        y_train - The training data labels that have been read
        x_test - The testing data that has been read
        y_test - The testing data labels that have been read
        """

        self.runHashLayer()
        self.runEmbeddingLayer()
        xTrain, yTrain, xTest, yTest = self.getFileNames()

        x_train, y_train, x_test, y_test = [], [], [], []
        # Call the readfiles method to read all the files
        x_train = self.readFiles(xTrain, False)
        y_train = self.readFiles(yTrain, True)

        x_test = self.readFiles(xTest, False)
        y_test = self.readFiles(yTest, True)

        return x_train, y_train, x_test, y_test
```

9.2.2.1.2 GraphEmbeddingLayer.py

```
import random
import tensorflow as tf
import networkx as nx
from networkx import DiGraph

class GraphEmbeddingLayer:
    def __init__(self, graph: DiGraph):
        """
        The Graph Embedding Layer class that carries out vectorization/unhashing on
nodes
        graph: DiGraph - The graph who's nodes are to be vectorized
        """
        self.graph = graph
        self.nodes = list(graph.nodes)
        self.nodeCopy = list(graph.nodes)
        self.edges = list(graph.edges)
        self.root = self.nodes[0]
        self.rootEmbedding = random.random() #set the root node's embedding to a
random float
        self.vectors = [self.rootEmbedding]
        self.nodeCopy.remove(self.root)
        self.unVectorised = self.nodeCopy #create a list of unvectorized nodes
```

```
        self.weights = {}
        for i in range(len(self.nodes)):
            #initiailise a set of random weights for each node
            self.weights[i] = tf.Variable(tf.random.normal(shape=(1, 1)))

        self.embeddingFunction(self.root, None)

    def embeddingFunction(self, node, parent):
        """
        The embedding function from where the vectorization function is called on
each node recursively
        node - The node to be vectorized
        parent - The parent node of 'node'
        """

        # run the recursive method as long as the unvectorized list is not empty
        if len(self.unVectorised) == 0:
            return self.vectors

        if parent is None: #if node is the root node
            # get all the children of the current node
            childNodes = nx.dfs_successors(self.graph, node)
            for child in childNodes:
                if child in self.unVectorised: #check that the node has not been
vectorized
                    self.unVectorised.remove(child)
                    rootIndex = self.nodes.index(self.root) #index of the root node
                    childIndex = self.nodes.index(child) #index of the current node
                    parentChildCount = len(self.getChildNodes(self.root)) #the
number of children the root node has
                    childNodeChildCount = len(self.getChildNodes(child)) # the
number of children this child node has

                    # run the vectorization function on the node given the above
variables
                    vec = self.vecFunction(parentChildCount, rootIndex, childIndex,
childNodeChildCount)
                    self.vectors.append(vec)
                    for childNode in self.getChildNodes(child):
                        # recursively call the method until all the nodes are
vectorized
                        self.embeddingFunction(childNode, child)
        else: #if the current node is not the root node
            if node in self.unVectorised:
                self.unVectorised.remove(node)
                parentIndex = self.nodes.index(parent) #index of the parent node
                childIndex = self.nodes.index(node)
                parentChildCount = len(self.getChildNodes(parent))
                childNodeChildCount = len(self.getChildNodes(node))
```

```
                vec = self.vecFunction(parentChildCount, parentIndex, childIndex,
childNodeChildCount)
                self.vectors.append(vec)
                for childNode in self.getChildNodes(node):
                    self.embeddingFunction(childNode, node)

    def getChildNodes(self, node):
        """
        Get the all the child nodes of a node
        node - The node who's children are to be collected

        Returns
        children - All the child nodes of 'node'
        """
        edges = self.edges #all the edges in the graph
        children = []
        for edge in edges:
            #if the right side of the edge is the current node, then the left side
is a child node
            if edge[0] == node:
                children.append(edge[1])

        return children

    def vecFunction(self, parentChildCount, parentIndex, childIndex,
childNodeChildCount):
        """
        The function in which vectorization is carried out
        parentChildCount - The number of children of the current node's parent node
        parentIndex - The index of the current node's parent node in the node list
        childIndex - The index of the current node in the node list
        childNodeChildCount - The number of children the current node has

        Returns
        result.numpy() - The numpy representation of the final result of
vectorization
        """
        pre = 0.0
        if childNodeChildCount > 0: #if the current node has one or more children
            pre = (parentChildCount/childNodeChildCount) *
(self.weights[parentIndex] + self.weights[childIndex])
        else: #if the current node does not have any children
            pre = (self.weights[parentIndex] + self.weights[childIndex])
        result = tf.reduce_logsumexp(pre) * 0.1
        if result < 0:
            result = result * -1.0
        return result.numpy()
```

9.2.2.1.3 GraphParser.py

```
import os, ast, random, sys
import networkx as nx
from ParsingAndEmbeddingLayers.Visitor import Visitor, HashVisitor
from os.path import dirname, join

class GraphParser:
    def __init__(self, hashed: bool):
        """
        Initliaise the GraphParser object with a truth value for hashed
        hashed: bool - Whether or not we are to work with hashed data
        """
        self.hashed = hashed

    def convertToGraph(self, filePath):
        """
        Read a file and convert its contents into its graph representation

        filePath - The file that is to be read

        Returns:
        graph: The graph representation of the contents of the file
        """
        programAST = ''
        with open (filePath, "r") as file:
            programAST = ast.parse(file.read())

        # determine if the values are to be hashed or unhashed
        if self.hashed is True:
            visitor = HashVisitor()
            visitor.generic_visit(programAST)
        else:
            visitor = Visitor()
            visitor.generic_visit(programAST)
        graph = visitor.convertToGraph()
        return graph

    def assignLabels(self, filePath):
        """
        Assign class labels to each file and its corresponding graph based on
        the sorting algorithm it implements. 0 for Merge Sort and 1 for Quick Sort.

        filePath - The path to the set of files

        Returns:
        graphs - The graph representations of the file contents
        labels - The class labels corresponding to the graphs
        """
        current_dir = dirname(__file__)
        filePath = join(current_dir, filePath)
```

```
        graphs, labels = [], []
        os.chdir(filePath)
        for file in os.listdir():
            if file.endswith(".py"):
                path = f"{filePath}/{file}"
                graphData = self.convertToGraph(path)
                graphs.append(graphData)
                if filePath.find("Merge") != -1:
                    labels.append(0)
                elif filePath.find("Quick") != -1:
                    labels.append(1)
        return graphs, labels

    def assignLabelsToFiles(self, file1, file2):
        """
        Call the assignLabels method on both the merge and quick sort files

        file1: The file path of the merge sort
        file2: The file path of the quick sort

        Returns:
        x - The graphs from all the sorting algorithms
        y - The class labels from all the files
        """
        graph1, labels1 = self.assignLabels(file1)
        graph2, labels2 = self.assignLabels(file2)

        x = graph1 + graph2
        y = labels1 + labels2

        return x, y

    def convertToMatrix(self, x):
        """
        Convert an individual graph to its matrix representation using the NetworkX
API

        x: The graph to be converted into a matrix

        Returns:
        matrices: The matrix representation of the graph in x
        """
        matrices = []
        for graph in x:
            matrix = nx.to_numpy_array(graph)
            matrices.append(matrix)
        return matrices

    def extractNodes(self, graphs):
        """
        Given a set of NetworkX graphs, extract the nodes from each graph
```

```
        graphs - The graphs from which nodes are to be extracted

        Returns:
        xNodes - The nodes extracted from the graphs in graphs
        """
        xNodes = []
        for i in range(len(graphs)):
            currentGraph = list(graphs[i].nodes)
            xNodes.append(currentGraph)
        return xNodes

    def readFiles(self):
        """
        Tie all the above methods together and parse the files

        Returns;
        x - The list of graphs and nodes in pairs of tuples
        x_graph - The lsit of graphs on its own
        matrices - THe matrix representations of the graphs in x_graphs
        labels - The class labels
        """
        merge = "./Datasets/Merge Sort"
        quick = "./Datasets/Quick Sort"

        currentDirectory = dirname(__file__) #the current working directory on the
device
        pathSplit = "/ParsingAndEmbeddingLayers"
        head = currentDirectory.split(pathSplit) #split the path into two separate
parts
        path = head[0]

        merge = join(path, merge) #join the directory path to the absolute path
        quick = join(path, quick)

        x_graph, y  = self.assignLabelsToFiles(merge, quick)

        matrices = self.convertToMatrix(x_graph)
        x_list = self.extractNodes(x_graph)
        x = []
        for i in range(len(x_list)):
            x.append([x_list[i], x_graph[i]])

        for i in range(len(x)):
            x[i][0].append(y[i])

        random.shuffle(x)
        labels = []
        for i in range(len(x)):
            labels.append(x[i][0][-1])
            x[i][0].pop()
```

```
        return x, x_graph, matrices, labels
```

9.2.2.2 Text

9.2.2.2.1 Textparser.py

```
import os
import tensorflow as tf
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer
from os.path import dirname, join

merge = "./Datasets/Merge Sort"
quick = "./Datasets/Quick Sort"

currentDirectory = dirname(__file__) #the current working directory on the device
pathSplit = "/ParsingAndEmbeddingLayers"
head = currentDirectory.split(pathSplit) #split the path into two separate parts
path = head[0]
# print(path)

merge = join(path, merge) #join the directory path to the absolute path
quick = join(path, quick)

class TextParser:
    def readTextFile(self, filePath):
        """
        Read the contents of a file
        filePath - THe file to be read

        Returns
        f.read() - The contents of the file in 'filePath'
        """
        with open(filePath, 'r') as f:
            return f.read()

    def assignTextLabels(self, filePath, fileList, labelList):
        """
        Assign class labels to each file based on the sorting algorithm it
implements
        filePath - The path of the file to be read
        fileList - The list to save the read files into
        labelList - The list to save the class labels into
        """
        os.chdir(filePath)
        for file in os.listdir():
```

```
            # Check whether file is in text format or not
            if file.endswith(".py"):
                path = f"{filePath}/{file}"
                # call read text file function
                fileList.append(self.readTextFile(path))
                if filePath.find("Merge") != -1:
                    labelList.append(0)
                elif filePath.find("Quick") != -1:
                    labelList.append(1)

    def getTextData(self):
        """
        Call the assignTextLabels method and splut the data into training and
testing data

        Returns
        x_train - The training data
        y_train - The training data labels
        x_test - The testing data
        y_test - The testing data labels
        """
        mergeList, mergeLabels, quickList, quickLabels  = [], [], [], [] #the
training and testing data
        self.assignTextLabels(merge, mergeList, mergeLabels)
        self.assignTextLabels(quick, quickList, quickLabels)
        x_train, y_train, x_test, y_test = [], [], [], []
        x_train = mergeList[:int(0.7*len(mergeList))] +
quickList[:int(0.7*len(quickList))]
        x_test = mergeList[int(0.7*len(mergeList)):] +
quickList[int(0.7*len(quickList)):]
        y_train = mergeLabels[:int(0.7*len(mergeList))] +
quickLabels[:int(0.7*len(quickList))]
        y_test = mergeLabels[int(0.7*len(mergeList)):] +
quickLabels[int(0.7*len(quickList)):]

        return x_train, y_train, x_test, y_test

    def getVectorizedTextData(self):
        """
        Run vectorization on the training and testing data

        Returns
        x_train - The vectorized form of the training data
        y_train - The training data labels
        x_test - The vectorized form of the testing data
        y_test - The testing data labels
        """
        x_train, y_train, x_test, y_test = self.getTextData()

        # use the Scikit Learn vectorizer to fit the data on the training set
        vectorizer = CountVectorizer(tokenizer=lambda doc:doc, min_df=2)
```

```
        vectorizer.fit(x_train)
        x_train = vectorizer.transform(x_train)
        x_train = x_train.toarray()/255.
        x_train = tf.convert_to_tensor(x_train,  dtype=np.float32)
        y_train = tf.keras.utils.to_categorical(y_train)

        x_test  = vectorizer.transform(x_test)
        x_test = x_test.toarray()/255.
        x_test = tf.convert_to_tensor(x_test, dtype=np.float32)
        y_test = tf.keras.utils.to_categorical(y_test)

        return x_train, y_train, x_test, y_test
```

9.2.2.3 Trees

9.2.2.3.1 TreeDataProcessor.py

```
import random
import tensorflow as tf
from ParsingAndEmbeddingLayers.Trees.TreeEmbeddingLayer import TreeEmbeddingLayer
from ParsingAndEmbeddingLayers.Trees.TreeParser import TreeParser
from os.path import dirname, join

merge = "./Datasets/Merge Sort"
quick = "./Datasets/Quick Sort"

currentDirectory = dirname(__file__) #the current working directory on the device
pathSplit = "/ParsingAndEmbeddingLayers"
head = currentDirectory.split(pathSplit) #split the path into two separate parts
path = head[0]

merge = join(path, merge) #join the directory path to the absolute path
quick = join(path, quick)

def attachLabels(x, y):
    """
    Pair up the class labels with the actual data and randomly shuffle the result

    x: The list containing the actual data
    y: The class labels

    Returns
    pairs - A list containing data and the labels in tuples
    """
    pairs = []
    for index in range(len(x)):
        pairs.append([x[index], y[index]])
    random.shuffle(pairs)
```

```
    return pairs

def getFileNames(hashed: bool):
    """
    Get the names of the files to be read depending on whether we are working
    with hashed data or not

    hashed: bool - Whether or not we are working with hashed data

    Returns
    xTrain - The name of the file containing the training data
    yTrain - The name of the file containing the training data labels
    xTest - The name of the file containing the testing data
    yTest - The name of the file containing the testing data labels
    """
    current_dir = dirname(__file__)
    if hashed is False:
        xTrain = join(current_dir, "./Tree Data/tree_x_train.txt")
        yTrain = join(current_dir, "./Tree Data/tree_y_train.txt")
        xTest = join(current_dir, "./Tree Data/tree_x_test.txt")
        yTest = join(current_dir, "./Tree Data/tree_y_test.txt")
    else:
        xTrain = join(current_dir, "./Tree Data/tree_x_train_hashed.txt")
        yTrain = join(current_dir, "./Tree Data/tree_y_train_hashed.txt")
        xTest = join(current_dir, "./Tree Data/tree_x_test_hashed.txt")
        yTest = join(current_dir, "./Tree Data/tree_y_test_hashed.txt")

    return xTrain, yTrain, xTest, yTest

def saveData(train, test):
    """
    Save the unhashed/vectorized training and testing data into lists

    train - The training data
    test - The testing data
    """
    print("\nCollecting training data", end="......")
    x_train, y_train = [], []
    for i in range(len(train)):
        if i % 2 == 0:
            print(end=".")
        current = train[i]
        embedding = TreeEmbeddingLayer(current) #embed all the nodes in the current
tree
        x_train.append(embedding.vectors)
        y_train.append(embedding.label)

    print("\nCollecting testing data", end="......")
    x_test, y_test = [], []
    for i in range(len(test)):
        if i % 5 == 0:
```

```
            print(end=".")
        embedding = TreeEmbeddingLayer(test[i])
        x_test.append(embedding.vectors)
        y_test.append(embedding.label)

    # write the contents of each list into the appropriate files with hashed =
false
    writeToFiles(x_train, y_train, x_test, y_test, False)

def saveHashData(train, test):
    """
    Save the hashed training and testing data into lists

    train - The hashed training data
    test - The hashed testing data
    """
    print("\nCollecting training data", end="......")
    x_train, y_train = [], []
    for i in range(len(train)):
        if i % 5 == 0:
            print(end=".")
        current = train[i]
        tree = current[0].getTreeEmbeddings(current[0]) #get the embedding of each
node in the tree
        x_train.append(tree)
        y_train.append(train[i][1])

    # Repeat for the testing data
    print("\nCollecting testing data", end="......")
    x_test, y_test = [], []
    for i in range(len(test)):
        if i % 5 == 0:
            print(end=".")
        current = test[i]
        tree = current[0].getTreeEmbeddings(current[0])
        x_test.append(tree)
        y_test.append(test[i][1])

    # write the contents of the training and testing data into files with hashed =
true
    writeToFiles(x_train, y_train, x_test, y_test, True)

def writeToFiles(x_train, y_train, x_test, y_test, hashed):
    """
    Write the contents of the inputs to this method into appropriate files

    x_train - The training data
    y_train - The training data labels
    x_test - The testing data
    y_test - The testing data labels
    hashed - Whether or not hashed is true
```

```
    """
    #collect the right file names based on the value of hashed
    xTrain, yTrain, xTest, yTest = getFileNames(hashed)

    # Write to each file
    with open(xTrain, 'w') as writer:
        for i in x_train:
            writer.write(str(i) + "\n")

    with open(yTrain, 'w') as writer:
        for i in y_train:
            writer.write(str(i) + "\n")

    with open(xTest, 'w') as writer:
        for i in x_test:
            writer.write(str(i) + "\n")

    with open(yTest, 'w') as writer:
        for i in y_test:
            writer.write(str(i) + "\n")

def readXFiles(filePath):
    """
    Read the contents of a file containing either training or testing data

    filePath - The path to the file containing the appropriate embeddings

    Returns
    values - The formatted contents of 'filePath'
    """
    with open(filePath, 'r') as reader:
        values = reader.readlines()

    for x in range(len(values)):
        # remove all unnecessary characters
        values[x] = values[x].replace("[", "").replace("]", "").strip("\n")
        values[x] = values[x].split(",")
        values[x] = [float(i) for i in values[x]]

    return values

def readYFiles(filePath):
    """
    Read the contents of a file containing either training labels or testing data
labels

    filePath - The path to the file containing the appropriate embeddings

    Returns
    values - The formatted contents of 'filePath'
    """
```

```
    with open(filePath, 'r') as reader:
        values = reader.readlines()

    for y in range(len(values)):
        # strip newlines and brackets and convert back to float
        values[y] = values[y].replace("[", "").replace("]", "").strip("\n")
        values[y] = values[y].split(" ")
        values[y] = [float(i) for i in values[y]]

    return values

def getData(hashed: bool):
    """
    This method is called by external classes when they want to access the contents
of the saved files
    It is also called by external methods when running the final experiments

    hashed: bool - Whether or not hashed is true

    Returns
    x_train - The formatted contents of the training data file
    y_train - The formatted contents of the training data labels file
    x_test - The formatted contents of the testing data file
    y_test - The formatted contents of the testing data labels file
    """
    saveHashedFiles()
    print()
    saveUnhashedFiles()

    xTrain, yTrain, xTest, yTest = getFileNames(hashed)

    x_train, y_train, x_test, y_test = [], [], [], []
    x_train = readXFiles(xTrain)
    y_train = readYFiles(yTrain)

    x_test = readXFiles(xTest)
    y_test = readYFiles(yTest)

    return x_train, y_train, x_test, y_test

def saveUnhashedFiles():
    """
    Call the saveData method on the unhashed/vectorized files
    """
    parser = TreeParser(False)
    mergeTree, mergeLabels = parser.parse(merge)
    quickTree, quickLabels = parser.parse(quick)

    x = mergeTree + quickTree
    y = mergeLabels + quickLabels
```

```
    pairs = attachLabels(x, y) #attach class labels
    split = int(0.8 * len(pairs)) #split 80-20 for training and testing
    train, test = pairs[:split], pairs[split:]
    saveData(train, test)

def saveHashedFiles():
    """
    Call the saveHashData method on the hashed files
    """
    hashParser = TreeParser(True)
    mergeHashTree, mergeLabels = hashParser.parse(merge)
    quickHashTree, quickLabels = hashParser.parse(quick)

    x_hash = mergeHashTree + quickHashTree
    y_hash = mergeLabels + quickLabels

    hashedPairs = attachLabels(x_hash, y_hash) #attach class labels
    split_hash = int(0.8 * len(hashedPairs))
    train_hash, test_hash = hashedPairs[:split_hash], hashedPairs[split_hash:]
    saveHashData(train_hash, test_hash)

def tensorToList(xValues: tf.Tensor):
    """
    Convert a tensor to a list for processing by the Non-Deep Learning models

    xValues: tf.Tensor - The data values to be converted into a list
    """
    x = []
    for i in xValues:
        x.append(list(i.numpy()))

    return x

def floatToInt(y):
    """
    Convert a tensor to a list and a float to an int for processing by the Non-Deep
Learning models

    y - The data values to be converted into a list of integers
    """
    y = list(y)
    for i in range(len(y)):
        j = y[i]
        j = list(j)
        j = j[0]
        y[i] = int(j)
    return y
```

9.2.2.3.2 TreeEmbeddingLayer.py

```
import random
import tensorflow as tf
from typing import List

class TreeEmbeddingLayer():
    def __init__(self, values: list[List, List]):
        """
        The stage where the nodes in each tree are vectorized/embedded
        values: list[List, List] - A list containing the trees and their class
labels
        """
        self.root = values[0] #get the root node
        self.nodes =  self.root.preOrderTraversal(self.root) #get all the nodes in
the tree
        self.label = values[1] #the class label
        self.rootVec = random.random() #set the root node's embedding to a random
float
        self.weights = {}
        self.bias = {}
        self.vectorEmbeddings = [[self.root, self.rootVec]]
        self.vectors = [self.rootVec] #the list of vectorized embeddings
        self.treeDepth = self.getTreeDepth(self.root) #the tree depth
        self.unVectorised = self.root.preOrderTraversal(self.root) #all the non-
root nodes are currently unvectorized
        self.rootIndex = self.nodes.index(self.root) #the index of the root node
        #the root node has been vectorized so remove it from the unvectorized list
        self.unVectorised.remove(self.root)

        self.initialiseInputWeights()
        self.embeddingFunction(self.root, None)

    def getTreeDepth(self, root):
        """
        Get the depth of the tree
        The depth is the maximum length of a single branch in the tree
        """
        if root is None:
            # return 0 is the root is empty
            return 0
        maxDepth = 0
        for child in root.children:
            # recursively loop through the tree to find the longest branch
            maxDepth = max(maxDepth, self.getTreeDepth(child))
        return maxDepth + 1

    def initialiseInputWeights(self):
        """
        Initialise a set of random weights and biases for each node to be
        used in calculating the final vectorized form of the node
```

```
        """
        for i in range(len(self.nodes)):
            self.weights[i] = tf.Variable(tf.random.normal(shape=(1, 1)))
            self.bias[i] = tf.Variable(tf.random.normal(shape=(1, 1)))

    def embeddingFunction(self, node, parent):
        """
        A recursive function that calls the vectorization function of individual
nodes in the tree

        node - The current node to be embedded
        parent - The parent node of 'node'

        Returns
        self.vectors - The list of all the vecotirzed nodes in the tree
        """
        if len(self.unVectorised) == 0:
            return self.vectors

        if parent is None: #when working with the root node
            functionNodes = node.children  #all the functions in the program file
            for function in functionNodes:
                if function in self.unVectorised: #while there are still
unvectorized function nodes
                    self.unVectorised.remove(function)
                    rootIndex = self.nodes.index(self.root) #the index of the
root/parent node
                    functionIndex = self.nodes.index(function) #the index of the
current node

                    # call the vectorization function on the function node and add
it to the vectors list
                    vec = self.vecFunction(len(functionNodes), rootIndex, function,
functionIndex)
                    self.vectors.append(vec)
                    self.vectorEmbeddings.append([function, vec])

                    # repeat for all the children of the current function node
                    for child in function.children:
                        self.embeddingFunction(child, function)
        else:
            # When working with child nodes deeper into the tree
            if node in self.unVectorised:
                self.unVectorised.remove(node)
                parentIndex = self.nodes.index(parent)
                childIndex = self.nodes.index(node)
                vec = self.vecFunction(len(parent.children), parentIndex, node,
childIndex)
                self.vectors.append(vec)
                self.vectorEmbeddings.append([node, vec])
                for child in node.children:
```

```
                    self.embeddingFunction(child, node)

    def vecFunction(self, parentChildCount, parentIndex, child, index):
        """
        The vectorization function where 'unhashing' is carried out

        parentChildCount - The number of children the current node's parent has
        parentIndex - The index of the current node's parent in the node list
        child - The current node to be vectorized
        index - The index of 'child' in the node list

        Returns
        result.numpy() - The vectorized form of the current node
        """
        childCount = len(child.children) #the number of children the current node
has
        pre = 0.0
        if childCount > 0: #if and only if the current node has any children
            pre = float(self.treeDepth) * (parentChildCount/childCount) *
(self.weights[parentIndex] + self.weights[index])
        else: #what to do if the current node does not have any children
            pre = float(self.treeDepth) *  (self.weights[parentIndex] +
self.weights[index])
        a = pre + self.bias[index]
        result = tf.reduce_logsumexp(a) * 0.1
        if result < 0:
            result = result * -1.0 #convert the result to a positive float
        return result.numpy()

    def findNodeEmbedding(self, node):
        """
        In the list of vector embeddings, find the embedding corresponding to a
particular node

        node: The node who's embedding is to be found

        Returns
        embedding - The embedding corresponding to 'node'
        """
        count, embedding = 0, 0.0
        for i in self.vectorEmbeddings:
            n = i[0] #the node
            e = i[1] #the embedding
            if n == node:
                embedding = e
            count += 1
        return embedding
```

9.2.2.3.3 TreeNode.py

```
class TreeNode:
    def __init__(self, embedding: float):
        """
        A class to represent individual nodes in a tree

        embedding: float - The embedding representation of a node
        """
        self.children = [] #placeholder for the child nodes of the current TreeNode
object
        self.embedding = embedding

    def preOrderTraversal(self, root):
        """
        Run the pre-order traversal algorithm on a root node to get all the nodes
present
        in a tree as well as their respective children

        root: The root node of the tree (Must be a TreeNode object)
        """
        objectTree = [] #placeholder for all the nodes in the tree
        if root is not None: #while the root node in the recursion is not none
            objectTree.append(root)
            for i in range(len(root.children)):
                # recursively add nodes to the objectTree
                objectTree = objectTree + self.preOrderTraversal(root.children[i])
        return list(set(objectTree))

    def getTreeEmbeddings(self, root):
        """
        Run preorder traversal on a root node and get the embeddings of all the
nodes in the tree

        root: The root node of the tree

        Returns
        embeddings - The list of embeddings of all the nodes in the tree
        """
        fullTree = self.preOrderTraversal(root)
        embeddings = []
        for i in fullTree:
            embeddings.append(i.embedding)

        return embeddings
```

9.2.2.3.4 Treeparser.py

```
import os, ast
from ParsingAndEmbeddingLayers.Trees.TreeNode import TreeNode
from ParsingAndEmbeddingLayers.Visitor import Visitor, HashVisitor

class TreeParser():
    def __init__(self, hashed: bool):
        """
        The Tree Parser class where the files are parsed for processing into trees

        hashed: bool - Whether or not hashing is to be used
        """
        self.hashed = hashed

    def convertToTree(self, filePath):
        """
        Convert the contents of a file into a tree

        filePath - The file who's contents are to be converted into a tree data
structure

        Returns
        tree - The tree representation of the contents of 'filePath'
        """
        programAST = '' #placeholder for the file contents
        with open (filePath, "r") as file:
            programAST = ast.parse(file.read())

        if self.hashed is True: #if we're working with hashed data, use the
HashVisitor class
            visitor = HashVisitor()
            visitor.generic_visit(programAST)
            #construct a tree from the edges provided by the HashVisitor
            tree = self.createTreeFromEdges(visitor.edges)
        else: #if we're working with unhashed data, use the Visitor class
            visitor = Visitor()
            visitor.generic_visit(programAST)
            tree = self.createTreeFromEdges(visitor.edges)
        return tree

    def parse(self, filePath):
        """
        Call the 'convertToTree' method on the contents of a file and assign class
labels

        filePath - The file to be converted into a tree

        Returns
        trees - The list of trees from each file
        labels - The list of class labels for each tree
```

```
        """
        trees, labels = [], []
        os.chdir(filePath)
        for file in os.listdir():
            # Check whether file is in text format or not
            if file.endswith(".py"):
                path = f"{filePath}/{file}"
                # call read text file function
                tree = self.convertToTree(path)
                trees.append(tree)
                if filePath.find("Merge") != -1:
                    labels.append(0)
                elif filePath.find("Quick") != -1:
                    labels.append(1)
        return trees, labels

    def createTreeFromEdges(self, edges):
        """
        Given a set of tree edges, create an abstract tree

        edges - The edges from which to create a tree
        """
        individualNodes = [] #placeholder for the nodes in the edges list
        for edge in edges: #for each pair of nodes that make an edge
            for i in edge: # for each node in a pair of nodes
                if i not in individualNodes: # prevent duplicate nodes
                    individualNodes.append(i)

        # create a set of all the nodes as TreeNode objects
        nodes = {i: TreeNode(i) for i in individualNodes}

        for parent, child in edges: #for each pair of edges
            # make sure the child node is not added to the list of child nodes
twice
            if child not in nodes[parent].children:
                nodes[parent].children.append(nodes[child])
            if child in individualNodes:
                # remove a child from the overall node list once it has been added
to its parent's child list
                individualNodes.remove(child)
        for edge in individualNodes:
            return nodes[edge] #return the remaining nodes in the edge list
```

9.2.2.3.5 TreeSegementation.py

```
import tensorflow as tf
from ParsingAndEmbeddingLayers.Trees import TreeDataProcessor as tdp
from ParsingAndEmbeddingLayers.Trees.TreeSegmentationLayer import
TreeSegmentationLayer

segmentCount = 40 #the number of segments to be used
segmentationLayer = TreeSegmentationLayer() #the segmentation layer object

def getUnsortedSegmentTrainData(hashed: bool):
    """
    Run unsorted segmentation on the training data

    hashed: bool - Whether or not the embeddings have been hashed

    Returns
    x_train_usum - The results of unsorted sum segmentation
    x_train_umean - The results of unsorted mean segmentation
    x_train_umax - The results of unsorted max segmentation
    x_train_umin - The results of unsorted min segmentation
    x_train_uprod - The results of unsorted product segmentation
    y_train - The class labels
    """
    x_train, y_train, x_test, y_test = tdp.getData(hashed)
    y_train = tf.keras.utils.to_categorical(y_train)

    x_train_usum, x_train_umean, x_train_umax, x_train_umin, x_train_uprod = [],
[], [], [], []

    for i in x_train: #for each tree in the list of trees
         #duplicate the embedding list into a 40-dimensional version of itself
        i = [i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i,
i,
        i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i]
        i = tf.convert_to_tensor(i)

        # unsorted sum segmentation
        uSum = segmentationLayer.segmentationLayer("unsorted_sum", i, segmentCount)
        uSum = tf.reshape(uSum, (len(uSum[0]), segmentCount))
        x_train_usum.append(uSum[0])

        # unsorted mean segmentation
        uMean = segmentationLayer.segmentationLayer("unsorted_mean", i,
segmentCount)
        uMean = tf.reshape(uMean, (len(uMean[0]), segmentCount))
        x_train_umean.append(uMean[0])

        # unsorted max segmentation
        uMax = segmentationLayer.segmentationLayer("unsorted_max", i, segmentCount)
        uMax = tf.reshape(uMax, (len(uMax[0]), segmentCount))
```

```
        x_train_umax.append(uMax[0])

        # unsorted min segmentation
        uMin = segmentationLayer.segmentationLayer("unsorted_min", i, segmentCount)
        uMin = tf.reshape(uMin, (len(uMin[0]), segmentCount))
        x_train_umin.append(uMin[0])

        # unsorted product segmentation
        uProd = segmentationLayer.segmentationLayer("unsorted_prod", i,
segmentCount)
        uProd = tf.reshape(uProd, (len(uProd[0]), segmentCount))
        x_train_uprod.append(uProd[0])

    # convert the results into tensors for use in the Deep Learning models
    x_train_usum = tf.convert_to_tensor(x_train_usum)
    x_train_umean = tf.convert_to_tensor(x_train_umean)
    x_train_umax = tf.convert_to_tensor(x_train_umax)
    x_train_umin = tf.convert_to_tensor(x_train_umin)
    x_train_uprod = tf.convert_to_tensor(x_train_uprod)

    return x_train_usum, x_train_umean, x_train_umax, x_train_umin, x_train_uprod,
y_train

def getUnsortedSegmentTestData(hashed):
    """
    Run unsorted segmentation on the testing data

    hashed: bool - Whether or not the embeddings have been hashed

    Returns
    x_test_usum - The results of unsorted sum segmentation
    x_test_umean - The results of unsorted mean segmentation
    x_test_umax - The results of unsorted max segmentation
    x_test_umin - The results of unsorted min segmentation
    x_test_uprod - The results of unsorted product segmentation
    y_test - The class labels
    """
    x_train, y_train, x_test, y_test = tdp.getData(hashed) # used for when we want
to test using hashed data
    y_test = tf.keras.utils.to_categorical(y_test)

    x_test_usum, x_test_umean, x_test_umax, x_test_umin, x_test_uprod = [], [], [],
[], []

    for i in x_test:
        i = [i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i,
i,
        i, i, i, i, i, i,  i, i, i, i, i, i, i, i, i, i]
        i = tf.convert_to_tensor(i)

        uSum = segmentationLayer.segmentationLayer("unsorted_sum", i, segmentCount)
```

```
        uSum = tf.reshape(uSum, (len(uSum[0]), segmentCount))
        x_test_usum.append(uSum[0])

        uMean = segmentationLayer.segmentationLayer("unsorted_mean", i,
segmentCount)
        uMean = tf.reshape(uMean, (len(uMean[0]), segmentCount))
        x_test_umean.append(uMean[0])

        uMax = segmentationLayer.segmentationLayer("unsorted_max", i, segmentCount)
        uMax = tf.reshape(uMax, (len(uMax[0]), segmentCount))
        x_test_umax.append(uMax[0])

        uMin = segmentationLayer.segmentationLayer("unsorted_min", i, segmentCount)
        uMin = tf.reshape(uMin, (len(uMin[0]), segmentCount))
        x_test_umin.append(uMin[0])

        uProd = segmentationLayer.segmentationLayer("unsorted_prod", i,
segmentCount)
        uProd = tf.reshape(uProd, (len(uProd[0]), segmentCount))
        x_test_uprod.append(uProd[0])

    x_test_usum = tf.convert_to_tensor(x_test_usum)
    x_test_umean = tf.convert_to_tensor(x_test_umean)
    x_test_umax = tf.convert_to_tensor(x_test_umax)
    x_test_umin = tf.convert_to_tensor(x_test_umin)
    x_test_uprod = tf.convert_to_tensor(x_test_uprod)

    return x_test_usum, x_test_umean, x_test_umax, x_test_umin, x_test_uprod,
y_test


def getSortedSegmentTrainData(hashed):
    """
    Run sorted segmentation on the training data

    hashed: bool - Whether or not the embeddings have been hashed

    Returns
    x_train_sum - The results of sorted sum segmentation
    x_train_mean - The results of sorted mean segmentation
    x_train_max - The results of sorted max segmentation
    x_train_min - The results of sorted min segmentation
    x_train_prod - The results of sorted product segmentation
    y_train - The class labels
    """
    x_train, y_train, x_test, y_test = tdp.getData(hashed)
    y_train = tf.keras.utils.to_categorical(y_train)

    x_train_sum, x_train_mean, x_train_max, x_train_min, x_train_prod = [], [], [],
[], []
```

```
    for i in x_train:
        i = [i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i,
        i, i, i, i, i, i, i,  i, i, i, i, i, i, i, i, i, i]
        i = tf.convert_to_tensor(i)

        uSum = segmentationLayer.segmentationLayer("sorted_sum", i, segmentCount)
        uSum = tf.reshape(uSum, (len(uSum[0]), segmentCount))
        x_train_sum.append(uSum[0])

        uMean = segmentationLayer.segmentationLayer("sorted_mean", i, segmentCount)
        uMean = tf.reshape(uMean, (len(uMean[0]), segmentCount))
        x_train_mean.append(uMean[0])

        uMax = segmentationLayer.segmentationLayer("sorted_max", i, segmentCount)
        uMax = tf.reshape(uMax, (len(uMax[0]), segmentCount))
        x_train_max.append(uMax[0])

        uMin = segmentationLayer.segmentationLayer("sorted_min", i, segmentCount)
        uMin = tf.reshape(uMin, (len(uMin[0]), segmentCount))
        x_train_min.append(uMin[0])

        uProd = segmentationLayer.segmentationLayer("sorted_prod", i, segmentCount)
        uProd = tf.reshape(uProd, (len(uProd[0]), segmentCount))
        x_train_prod.append(uProd[0])

    x_train_sum = tf.convert_to_tensor(x_train_sum)
    x_train_mean = tf.convert_to_tensor(x_train_mean)
    x_train_max = tf.convert_to_tensor(x_train_max)
    x_train_min = tf.convert_to_tensor(x_train_min)
    x_train_prod = tf.convert_to_tensor(x_train_prod)

    return x_train_sum, x_train_mean, x_train_max, x_train_min, x_train_prod,
y_train


def getSortedSegmentTestData(hashed):
    """
    Run sorted segmentation on the testing data

    hashed: bool - Whether or not the embeddings have been hashed

    Returns
    x_test_sum - The results of sorted sum segmentation
    x_test_mean - The results of sorted mean segmentation
    x_test_max - The results of sorted max segmentation
    x_test_min - The results of sorted min segmentation
    x_test_prod - The results of sorted product segmentation
    y_test - The class labels
    """
    x_train, y_train, x_test, y_test = tdp.getData(hashed)
    y_test = tf.keras.utils.to_categorical(y_test)
```

```
    x_test_sum, x_test_mean, x_test_max, x_test_min, x_test_prod = [], [], [], [],
[]
    for i in x_test:
        i = [i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i, i,
        i, i, i, i, i, i, i, i, i,  i, i, i, i, i, i, i, i, i, i]
        i = tf.convert_to_tensor(i)

        uSum = segmentationLayer.segmentationLayer("sorted_sum", i, segmentCount)
        uSum = tf.reshape(uSum, (len(uSum[0]), segmentCount))
        x_test_sum.append(uSum[0])

        uMean = segmentationLayer.segmentationLayer("sorted_mean", i, segmentCount)
        uMean = tf.reshape(uMean, (len(uMean[0]), segmentCount))
        x_test_mean.append(uMean[0])

        uMax = segmentationLayer.segmentationLayer("sorted_max", i, segmentCount)
        uMax = tf.reshape(uMax, (len(uMax[0]), segmentCount))
        x_test_max.append(uMax[0])

        uMin = segmentationLayer.segmentationLayer("sorted_min", i, segmentCount)
        uMin = tf.reshape(uMin, (len(uMin[0]), segmentCount))
        x_test_min.append(uMin[0])

        uProd = segmentationLayer.segmentationLayer("sorted_prod", i, segmentCount)
        uProd = tf.reshape(uProd, (len(uProd[0]), segmentCount))
        x_test_prod.append(uProd[0])


    x_test_sum = tf.convert_to_tensor(x_test_sum)
    x_test_mean = tf.convert_to_tensor(x_test_mean)
    x_test_max = tf.convert_to_tensor(x_test_max)
    x_test_min = tf.convert_to_tensor(x_test_min)
    x_test_prod = tf.convert_to_tensor(x_test_prod)

    return x_test_sum, x_test_mean, x_test_max, x_test_min, x_test_prod, y_test
```

9.2.2.3.6 TreeSegmentationLayer.py

```
import tensorflow as tf

class TreeSegmentationLayer:
    def __init__(self):
        """
        The Tree Segmentation layer class where the functions for carrying out
        segmentation on trees are declared
        """
        pass

    def segmentationFunction(self, segmentationFunction: str):
```

```
        """
        Determine the type of segmentation function to use

        segmentationFunction: str - The string representation of the chosen
segmentation function

        Returns
        The tensorflow function corresponding to 'segmentationFunction'
        """
        segmentationFunction = segmentationFunction.split("_")
        if segmentationFunction[0] == "sorted":
            if segmentationFunction[1] == "sum":
                return tf.math.segment_sum
            if segmentationFunction[1] == "mean":
                return tf.math.segment_mean
            if segmentationFunction[1] == "max":
                return tf.math.segment_max
            if segmentationFunction[1] == "min":
                return tf.math.segment_min
            if segmentationFunction[1] == "prod":
                return tf.math.segment_prod
        elif segmentationFunction[0] == "unsorted":
            if segmentationFunction[1] == "sum":
                return tf.math.unsorted_segment_sum
            if segmentationFunction[1] == "mean":
                return tf.math.unsorted_segment_mean
            if segmentationFunction[1] == "max":
                return tf.math.unsorted_segment_max
            if segmentationFunction[1] == "min":
                return tf.math.unsorted_segment_min
            if segmentationFunction[1] == "prod":
                return tf.math.unsorted_segment_prod
        else:
            return None

    def segmentationLayer(self, segmentationFunction: str, nodeEmbeddings:
tf.Tensor, numSegments: int):
        """
        The segmentation function proper where the tree is segmented

        segmentationFunction: str - The string representation of the segmentation
function
        nodeEmbeddings: tf.Tensor - The nodes to be segmented
        numSegments: int - The number of segments to be used

        Returns
        The segmented representation of 'nodeEmbeddings'
        """
        seg = segmentationFunction.lower()
        segmentationFunction = self.segmentationFunction(segmentationFunction)
```

```
        if seg.split("_")[0] == "unsorted":
            return segmentationFunction(nodeEmbeddings, tf.constant([0, 1, 2, 3, 4,
5, 6, 7, 8, 9, 10, 11,
            12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
30, 31,
            32, 33, 34, 35, 36, 37, 38, 39]), numSegments)
        else:
            return segmentationFunction(nodeEmbeddings, tf.constant([0, 1, 2, 3, 4,
5, 6, 7, 8, 9, 10, 11,
            12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
30, 31,
            32, 33, 34, 35, 36, 37, 38, 39]))
```

9.2.3 Visitor.py

```
import ast, re
import networkx as nx
from abc import ABC, abstractmethod

class AbstractVisitor(ast.NodeVisitor, ABC):
    def __init__(self):
        """
        An Abstract Visitor object. This is the super class that defines the
        core functionalities of the Visitor and HashVisitor classes
        """
        self.nodes = []
        self.edges = []
        self.adjList = []
        self.hashedNodes = []

    @abstractmethod
    def generic_visit(self, node):
        raise NotImplementedError()

    def splitCamelCase(self, identifier: str):
        """
        Split an identifier based on 'camelCase'

        identifier: str - The identifier to split

        Returns
        The split identifier
        """
        splitIdentifier = re.finditer('.+?(?:(?<=[a-z])(?=[A-Z])|(?<=[A-Z])(?=[A-
Z][a-z])|$)', identifier)
        return [i.group(0) for i in splitIdentifier]
```

```
    def splitSnakeCase(self, identifier: str):
        """
        Split an identifier based on 'snake_case'

        identifier: str - The identifier to split

        Returns
        The split identifier
        """
        return identifier.split("_")

    def splitIdentifier(self, identifier):
        """
        Split an idenfifier regardless of whether it is in 'snake_case' or
'camelCase'

        identifier - The identifier to split

        Returns
        finalSplitID - The split identifier as a list of its parts
        """
        splitId = self.splitSnakeCase(identifier) #start by splitting based on
snake_case
        finalSplitID = []
        idParts = []
        for part in splitId:
            #if there are still more parts to split, split based on camelCase
            if len(part) > 0:
                idParts.append(self.splitCamelCase(part))

        if len(idParts) == 0:
            return [identifier]
        else:
            for i in idParts:
                for j in i:
                    finalSplitID.append(j)

        return finalSplitID

    def convertToGraph(self):
        """
        Convert the contents of an AST into a NetworkX DiGraph object

        Returns
        graph - The NetworkX DiGraph reepresentation of the AST
        """
        graph = nx.DiGraph()
        graph.add_edges_from(self.edges)
        return graph

    def createAdjList(self):
```

```
        """
        Create an adjacency list containing all the nodes in the tree using hashing
        """
        for node in self.nodes:
            children = list(ast.iter_child_nodes(node))
            if len(children) > 0:
                self.adjList.append([1/hash(node), [1/hash(child) for child in
children]])

    def visitSpecial(self, node):
        """
        Visit a node and return a string representation of the node
        node - The node to visit

        Returns
        A string representation of 'node'
        """
        if isinstance(node, ast.FunctionDef or ast.AsyncFunctionDef or
ast.ClassDef):
            return self.visitDef(node)
        elif isinstance(node, ast.Return):
            return self.visitReturn(node)
        elif isinstance(node, ast.Delete):
            return self.visitDelete(node)
        elif isinstance(node, ast.Attribute):
            return self.visitAttribute(node)
        elif isinstance(node, ast.Assign):
            return self.visitAssign(node)
        elif isinstance(node, ast.AugAssign or ast.AnnAssign):
            return self.visitAugAssign(node)
        elif isinstance(node, ast.Attribute):
            return self.visitAttribute(node)
        elif isinstance(node, ast.Name):
            return self.visitName(node)
        elif isinstance(node, ast.Constant):
            return self.visitConstant(node)
        else:
            className = 'value = ' + node.__class__.__name__
            return className

    def visitDef(self, node: ast.FunctionDef or ast.AsyncFunctionDef or
ast.ClassDef):
        """
        The special visitor class for Function and Class definition objects

        Returns
        The name of the function or class definition
        """
        return str(node.name)

    def visitReturn(self, node: ast.Return):
```

```
        """
        The special visitor class for Return objects

        Returns
        returnValue - The string representation of the value to be returned
        """
        returnValue = "return " + str(node.value)
        return returnValue

    def visitDelete(self, node: ast.Delete):
        """
        The special visitor class for Delete objects

        Returns
        returnValue - The string representation of the value to be deleted
        """
        returnValue = "delete " + str(node.targets)
        return returnValue

    def visitAssign(self, node: ast.Assign):
        """
        The special visitor class for Assign objects

        Returns
        returnValue - The string representation of the value to be assigned and its
targets
        """
        returnValue = "assign " + str(node.value) + " to " + str(node.targets)
        return returnValue

    def visitAugAssign(self, node: ast.AugAssign or ast.AnnAssign):
        """
        The special visitor class for Augmented and Annotated Assign objects

        Returns
        returnValue - The string representation of the value to be assigned and its
targets
        """
        returnValue = "assign " + str(node.value) + " to " + str(node.target)
        return returnValue

    def visitAttribute(self, node: ast.Attribute):
        """
        The special visitor class for Attribute objects

        Returns
        returnValue - The string representation of the attribute name and its value
        """
        returnValue = str(node.attr) + " = " + str(node.value)
        return returnValue
```

```
    def visitName(self, node: ast.Name):
        """
        The special visitor class for Name objects

        Returns
        The name of the object
        """
        return str(node)

    def visitConstant(self, node: ast.Constant):
        """
        The special visitor class for Constant objects

        Returns
        The value of the constant
        """
        return "value = " + str(node.value)



class Visitor(AbstractVisitor):
    def __init__(self):
        """
        The Visitor object visits all the nodes in an AST without hashing them
        """
        super().__init__()

    def generic_visit(self, node):
        """
        Recursively visit each node in the AST and add it to the node list
        """
        if node not in self.nodes:
            self.nodes.append(node)

        if isinstance(node, ast.AST):
            for child in list(ast.iter_child_nodes(node)): #loop through all the
children of the current node
                for child in list(ast.iter_child_nodes(node)):
                    self.edges.append([node, child])
                    if child not in self.nodes:
                        self.nodes.append(child)
                        self.generic_visit(child)

        elif isinstance(node, list):
            for child in list(ast.iter_child_nodes(node)):
                self.edges.append([node, child])
                if child not in self.nodes: #prevent duplicate nodes in the node
list
                    self.nodes.append(child)
                    self.generic_visit(child) #recursively visit all the nodes
```

```python
class HashVisitor(AbstractVisitor):
    def __init__(self):
        """
        The HashVisitor object visits all the nodes in an AST and performs hashing
on them
        """
        super().__init__()

    def generic_visit(self, node):
        """
        Recursively visit each node in the AST,
        visit that node specially,
        perform the hashing algorithm on it
        add it to the node list
        and add all its children in pairs to the list of edges
        """
        nodeEmbedding = self.visitSpecial(node)
        nodeEmbedding = 1/hash(node) + 1/hash(nodeEmbedding) * 0.005
        if node not in self.nodes:
            self.nodes.append(node)
            self.hashedNodes.append(nodeEmbedding)

        if isinstance(node, ast.AST):
            for child in list(ast.iter_child_nodes(node)):
                childEmbedding = self.visitSpecial(child)
                childEmbedding = 1/hash(child) + 1/hash(childEmbedding) * 0.005
                self.edges.append([nodeEmbedding, childEmbedding])
                if child not in self.nodes:
                    self.nodes.append(child)
                    self.hashedNodes.append(childEmbedding)
                    self.generic_visit(child)

        elif isinstance(node, list):
            for child in list(ast.iter_child_nodes(node)):
                childEmbedding = self.visitSpecial(child)
                childEmbedding = 1/hash(child) + 1/hash(childEmbedding) * 0.005
                self.edges.append([(node, nodeEmbedding), (child, childEmbedding)])
                if child not in self.nodes:
                    self.nodes.append(child)
                    self.hashedNodes.append(childEmbedding)
                    self.generic_visit(child)
```

9.2.4 finalTreeExperiments.py

```
import numpy as np
from Networks.MLP import MLP
from Networks.RNN import RNN
from Networks.NaiveBayes import NBClassifier
from Networks.DenseModel import runDenseModel
from Networks.SKLearnClassifiers import SGDClassify, rfClassify, SVMClassify
from ParsingAndEmbeddingLayers.Trees.TreeSegmentationLayer import
TreeSegmentationLayer
from ParsingAndEmbeddingLayers.Trees import TreeDataProcessor as tdp
from ParsingAndEmbeddingLayers.Trees import TreeSegmentation as seg


# hashed = True
hashed = False
x_train_usum, x_train_umean, x_train_umax, x_train_umin, x_train_uprod, y_train =
seg.getUnsortedSegmentTrainData(hashed)
x_test_usum, x_test_umean, x_test_umax, x_test_umin, x_test_uprod, y_test =
seg.getUnsortedSegmentTestData(hashed)

lstm = "lstm"
gru = "gru"
simpleRNN = "rnn"

print("USING HASHED =", str(hashed).upper(), "DATA")
segmentCount = 40
segmentationLayer = TreeSegmentationLayer()
layers = [segmentCount, 128, 128, 2]
epochs = 10
lr = 0.001

def runUnsortedMLPModel(activationFunction: str):
    """
    Run the MLP on the tree data

    activationFunction: str - The activation function to apply
    """
    print("RUNNING RNN MODELS USING UNSORTED SEGMENTATION")
    model = MLP(x_train_umean, y_train, layers, activationFunction, lr, epochs)
    metrics = model.runFFModel(x_train_umean, y_train, x_test_umean, y_test)

    print("USING", activationFunction.upper())
    print("Loss:", np.average(metrics['trainingLoss']), "Training Accuracy:",
        np.average(metrics['trainingAccuracy']), "Validation accuracy:",
        np.average(metrics['validationAccuracy']), "\n")

def runLSTM(activationFunction: str):
    """
    Run the LSTM on the tree data
```

```
    activationFunction: str - The activation function to apply
    """
    model = RNN("lstm", x_train_umean, y_train, x_test_umean, y_test,
activationFunction)
    model.runModel(lstm, 12, 10, 70)

def runGRU(activationFunction: str):
    """
    Run the GRU on the tree data

    activationFunction: str - The activation function to apply
    """
    model = RNN("gru", x_train_umean, y_train, x_test_umean, y_test,
activationFunction)
    model.runModel(gru, 64, 20, 64)

def runSRNN(activationFunction: str):
    """
    Run the Simple RNN on the tree data

    activationFunction: str - The activation function to apply
    """
    model = RNN("rnn", x_train_umean, y_train, x_test_umean, y_test,
activationFunction)
    model.runModel(simpleRNN, 64, 10, 64)

def runUnsortedDenseModel():
    """
    Run the Densely connnected model on the tree data using all 4 activation
functions
    """
    print("DENSE UNSORTED MODEL AND SOFTMAX")
    runDenseModel(x_train_umean, y_train, x_test_umean, y_test, "softmax", 5, 10,
"denseSegmented.hdf5")
    print("DENSE UNSORTED MODEL AND RELU")
    runDenseModel(x_train_umean, y_train, x_test_umean, y_test, "relu", 5, 10,
"denseSegmented.hdf5")
    print("DENSE UNSORTED MODEL AND TANH")
    runDenseModel(x_train_umean, y_train, x_test_umean, y_test, "tanh", 5, 10,
"denseSegmented.hdf5")
    print("DENSE UNSORTED MODEL AND SIGMOID")
    runDenseModel(x_train_umean, y_train, x_test_umean, y_test, "sigmoid", 5, 10,
"denseSegmented.hdf5")

def runGaussianNBCUnsorted():
    """
    Run the Gaussian Naïve Bayes CLassifier on the tree data
    """
    # convert the training and testing data and their labels into lists from
tensors
```

```python
    x_train = tdp.tensorToList(x_train_umean)
    x_test = tdp.tensorToList(x_test_umean)

    yTrain = tdp.floatToInt(y_train)
    yTest = tdp.floatToInt(y_test)

    x, y = [], []
    for i in x_train:
        x.append(i)
    for i in x_test:
        x.append(i)

    for i in yTrain:
        y.append(i)
    for i in yTest:
        y.append(i)

    nbc = NBClassifier(x, y)
    print("Gaussian NB Classifier Accuracy:", nbc.gaussianCrossValidation(x, y))

def runSKLearnClassifiersUnsorted():
    """
    Run the SVM, SGD and RF classifiers on the tree data
    """
    yTrain = tdp.floatToInt(y_train)
    yTest = tdp.floatToInt(y_test)

    sgdUSumAccuracy = SGDClassify(x_train_umean, yTrain, x_test_umean, yTest)
    print("SGD CLASSIFIER AND UNSORTED:", sgdUSumAccuracy)

    rfUSumAccuracy = rfClassify(x_train_umean, yTrain, x_test_umean, yTest)
    print("RANDOM FOREST CLASSIFIER AND UNSORTED:", rfUSumAccuracy)

    svmUSumAccuracy = SVMClassify(x_train_umean, yTrain, x_test_umean, yTest)
    print("SVM CLASSIFIER AND UNSORTED:", svmUSumAccuracy)

# EXPERIMENTS AND RESULTS
runUnsortedMLPModel("relu")
runUnsortedMLPModel("tanh")
runUnsortedMLPModel("softmax")
runUnsortedMLPModel("sigmoid")

runLSTM("relu")
runLSTM("tanh")
runLSTM("softmax")
runLSTM("sigmoid")

runGRU("relu")
runGRU("tanh")
runGRU("softmax")
runGRU("sigmoid")
```

```
runSRNN("relu")
runSRNN("tanh")
runSRNN("softmax")
runSRNN("sigmoid")

runUnsortedDenseModel()

runGaussianNBCUnsorted()

runSKLearnClassifiersUnsorted()
```

9.2.5 graphExperiments.py

```
import numpy as np
from Networks.MLP import MLP
from Networks.RNN import RNN
from Networks.NaiveBayes import NBClassifier
from Networks.DenseModel import runDenseModel
from Networks.SKLearnClassifiers import SGDClassify, rfClassify, SVMClassify
import ParsingAndEmbeddingLayers.Graphs.GraphDataProcessor as GDP

hashed = True  # if you want to test with hashed graphs, set HASHED to True
# hashed = False # else, set to False
gdp = GDP.GraphDataProcessor(hashed)

def runMLPonPaddedGraphs():
      """RUNNING ON PADDED GRAPHS"""
      x_train, y_train, x_test, y_test = gdp.runProcessor1()

      layers = [len(x_train[0]), 128, 128, 2]
      epochs = 10
      lr = 0.001
      mlp1 = MLP(x_train, y_train, layers, "relu", lr, epochs)
      metrics1 = mlp1.runFFModel(x_train, y_train, x_test, y_test)

      mlp2 = MLP(x_train, y_train, layers, "tanh", lr, epochs)
      metrics2 = mlp2.runFFModel(x_train, y_train, x_test, y_test)

      mlp3 = MLP(x_train, y_train, layers, "softmax", lr, epochs)
      metrics3 = mlp3.runFFModel(x_train, y_train, x_test, y_test)

      mlp4 = MLP(x_train, y_train, layers, "sigmoid", lr, epochs)
      metrics4 = mlp4.runFFModel(x_train, y_train, x_test, y_test)

      print("USING THE MULTI-LAYER PERCEPTRON AND PADDED GRAPHS")
      print("USING RELU")
      print("Loss:", np.average(metrics1['trainingLoss']), "Training Accuracy:",
```

```
            np.average(metrics1['trainingAccuracy']), "Validation accuracy:",
            np.average(metrics1['validationAccuracy']), "\n")

      print("USING TANH")
      print("Loss:", np.average(metrics2['trainingLoss']), "Training Accuracy:",
            np.average(metrics2['trainingAccuracy']), "Validation accuracy:",
            np.average(metrics2['validationAccuracy']), "\n")

      print("USING SOFTMAX")
      print("Loss:", np.average(metrics3['trainingLoss']), "Training Accuracy:",
            np.average(metrics3['trainingAccuracy']), "Validation accuracy:",
            np.average(metrics3['validationAccuracy']), "\n")

      print("USING SIGMOID")
      print("Loss:", np.average(metrics4['trainingLoss']), "Training Accuracy:",
            np.average(metrics4['trainingAccuracy']), "Validation accuracy:",
            np.average(metrics4['validationAccuracy']), "\n")

def runMLPonSegmentedGraphs():
      """RUNNING ON SEGMENTED GRAPHS"""
      x_train, y_train, x_test, y_test = gdp.runProcessor3()

      layers = [len(x_train[0]), 128, 128, 2]
      epochs = 10
      lr = 0.001
      mlp1 = MLP(x_train, y_train, layers, "relu", lr, epochs)
      metrics1 = mlp1.runFFModel(x_train, y_train, x_test, y_test)

      mlp2 = MLP(x_train, y_train, layers, "tanh", lr, epochs)
      metrics2 = mlp2.runFFModel(x_train, y_train, x_test, y_test)

      mlp3 = MLP(x_train, y_train, layers, "softmax", lr, epochs)
      metrics3 = mlp3.runFFModel(x_train, y_train, x_test, y_test)

      mlp4 = MLP(x_train, y_train, layers, "sigmoid", lr, epochs)
      metrics4 = mlp4.runFFModel(x_train, y_train, x_test, y_test)


      print("USING THE MULTI-LAYER PERCEPTRON AND SEGMENTATION")
      print("USING RELU")
      print("Loss:", np.average(metrics1['trainingLoss']), "Training Accuracy:",
            np.average(metrics1['trainingAccuracy']), "Validation accuracy:",
            np.average(metrics1['validationAccuracy']), "\n")

      print("USING TANH")
      print("Loss:", np.average(metrics2['trainingLoss']), "Training Accuracy:",
            np.average(metrics2['trainingAccuracy']), "Validation accuracy:",
            np.average(metrics2['validationAccuracy']), "\n")

      print("USING SOFTMAX")
      print("Loss:", np.average(metrics3['trainingLoss']), "Training Accuracy:",
```

```
            np.average(metrics3['trainingAccuracy']), "Validation accuracy:",
            np.average(metrics3['validationAccuracy']), "\n")

      print("USING SIGMOID")
      print("Loss:", np.average(metrics4['trainingLoss']), "Training Accuracy:",
            np.average(metrics4['trainingAccuracy']), "Validation accuracy:",
            np.average(metrics4['validationAccuracy']), "\n")

def runDenseModelonPaddedGraphs():
      x_train, y_train, x_test, y_test = gdp.runProcessor1()

      print("DENSE PADDED MODEL AND SOFTMAX")
      runDenseModel(x_train, y_train, x_test, y_test, "softmax", 5, 10,
"densePaddedSoftmax.hdf5")
      print("DENSE PADDED MODEL AND RELU")
      runDenseModel(x_train, y_train, x_test, y_test, "relu", 5, 10,
"densePaddedRelu.hdf5")
      print("DENSE PADDED MODEL AND TANH")
      runDenseModel(x_train, y_train, x_test, y_test, "tanh", 5, 10,
"densePaddedTanh.hdf5")
      print("DENSE PADDED MODEL AND SIGMOID")
      runDenseModel(x_train, y_train, x_test, y_test, "sigmoid", 5, 10,
"densePaddedSigmoid.hdf5")

def runDenseModelonSegmentedGraphs():
      x_train, y_train, x_test, y_test = gdp.runProcessor3()

      print("DENSE SEGMENTED MODEL AND SOFTMAX")
      runDenseModel(x_train, y_train, x_test, y_test, "softmax", 5, 10,
"denseSegmented.hdf5")
      print("DENSE SEGMENTED MODEL AND RELU")
      runDenseModel(x_train, y_train, x_test, y_test, "relu", 5, 10,
"denseSegmented.hdf5")
      print("DENSE SEGMENTED MODEL AND TANH")
      runDenseModel(x_train, y_train, x_test, y_test, "tanh", 5, 10,
"denseSegmented.hdf5")
      print("DENSE SEGMENTED MODEL AND SIGMOID")
      runDenseModel(x_train, y_train, x_test, y_test, "sigmoid", 5, 10,
"denseSegmented.hdf5")

def runGaussianNBConPaddedGraphs():
      x_train, y_train, x_test, y_test = gdp.runProcessor2()
      x, y = [], []
      for i in x_train:
            x.append(i)
      for i in x_test:
            x.append(i)

      for i in y_train:
            y.append(i)
      for i in y_test:
```

```
            y.append(i)
      nbc = NBClassifier(x, y)
      print("Gaussian NB CLassifier Accuracy:", nbc.gaussianCrossValidation(x, y))

def runGaussianNBConSegmentedGraphs():
      x_train, y_train, x_test, y_test = gdp.runProcessor4()
      x, y = [], []
      for i in x_train:
            x.append(i)
      for i in x_test:
            x.append(i)

      for i in y_train:
            y.append(i)
      for i in y_test:
            y.append(i)
      nbc = NBClassifier(x, y)
      print("Gaussian NB CLassifier Accuracy:", nbc.gaussianCrossValidation(x, y))

def runLSTMonPaddedGraphs(activationFunction: str):
      """RUNNING LSTM ON PADDED GRAPHS"""
      x_train, y_train, x_test, y_test = gdp.runProcessor1()
      print(activationFunction.upper())
      graphLSTM = "lstm"
      lstmModel = RNN(graphLSTM, x_train, y_train, x_test, y_test,
activationFunction)
      lstmModel.runModel(graphLSTM, 256, 10, 5, "graphLSTMPadded.hdf5")

def runLSTMonSegmentedGraphs(activationFunction: str):
      """RUNNING LSTM ON PADDED GRAPHS"""
      x_train, y_train, x_test, y_test = gdp.runProcessor3()
      print(activationFunction.upper())
      graphLSTM = "lstm"
      lstmModel = RNN(graphLSTM, x_train, y_train, x_test, y_test,
activationFunction)
      lstmModel.runModel(graphLSTM, 256, 10, 5, "graphLSTMSegmented.hdf5")

def runGRUonPaddedGraphs(activationFunction: str):
      """RUNNING GRU ON PADDED GRAPHS"""
      x_train, y_train, x_test, y_test = gdp.runProcessor1()
      print(activationFunction.upper())
      gru = "gru"
      gruModel = RNN(gru, x_train, y_train, x_test, y_test, activationFunction)
      gruModel.runModel(gru, 256, 10, 5, "graphGRUPadded.hdf5")

def runGRUonSegmentedGraphs(activationFunction: str):
      """RUNNING GRU ON PADDED GRAPHS"""
      x_train, y_train, x_test, y_test = gdp.runProcessor3()
      print(activationFunction.upper())
      gru = "gru"
      gruModel = RNN(gru, x_train, y_train, x_test, y_test, activationFunction)
```

```
    gruModel.runModel(gru, 256, 10, 5, "graphGRUSegmented.hdf5")

def runSRNNonPaddedGraphs(activationFunction: str):
    """RUNNING GRU ON PADDED GRAPHS"""
    x_train, y_train, x_test, y_test = gdp.runProcessor1()
    print(activationFunction.upper())
    srnn = "rnn"
    srnnModel = RNN(srnn, x_train, y_train, x_test, y_test, activationFunction)
    srnnModel.runModel(srnn, 256, 10, 5, "graphSRNNPadded.hdf5")

def runSRNNonSegmentedGraphs(activationFunction: str):
    """RUNNING GRU ON PADDED GRAPHS"""
    x_train, y_train, x_test, y_test = gdp.runProcessor3()
    print(activationFunction.upper())
    srnn = "rnn"
    srnnModel = RNN(srnn, x_train, y_train, x_test, y_test, activationFunction)
    srnnModel.runModel(srnn, 256, 10, 5, "graphSRNNSegmented.hdf5")

def runSKLearnClassifiersOnPaddedGraphs():
    """RUNNING ON PADDED GRAPHS"""
    x_train, y_train, x_test, y_test = gdp.runProcessor2()

    sgdUSumPadAccuracy = SGDClassify(x_train, y_train, x_test, y_test)
    print("SGD CLASSIFIER AND PADDED GRAPHS:", sgdUSumPadAccuracy)

    rfUSumPadAccuracy = rfClassify(x_train, y_train, x_test, y_test)
    print("RANDOM FOREST CLASSIFIER AND PADDED GRAPHS:", rfUSumPadAccuracy)

    svmUSumPadAccuracy = SVMClassify(x_train, y_train, x_test, y_test)
    print("SVM CLASSIFIER AND PADDED GRAPHS:", svmUSumPadAccuracy)

def runSKLearnClassifiersOnSegmentedGraphs():
    """RUNNING ON SEGMENTED GRAPHS"""
    x_train, y_train, x_test, y_test = gdp.runProcessor4()

    sgdUSumSegAccuracy = SGDClassify(x_train, y_train, x_test, y_test)
    print("SGD CLASSIFIER AND SEGMENTED GRAPHS:", sgdUSumSegAccuracy)

    rfUSumSegAccuracy = rfClassify(x_train, y_train, x_test, y_test)
    print("RANDOM FOREST CLASSIFIER AND SEGMENTED GRAPHS:", rfUSumSegAccuracy)

    svmUSumSegAccuracy = SVMClassify(x_train, y_train, x_test, y_test)
    print("SVM CLASSIFIER AND SEGMENTED GRAPHS:", svmUSumSegAccuracy)

runMLPonPaddedGraphs()
runMLPonSegmentedGraphs()

runDenseModelonPaddedGraphs()
runDenseModelonSegmentedGraphs()

runGaussianNBConPaddedGraphs()
```

```
runGaussianNBConSegmentedGraphs()

runLSTMonPaddedGraphs("relu")
runLSTMonPaddedGraphs("tanh")
runLSTMonPaddedGraphs("sigmoid")
runLSTMonPaddedGraphs("softmax")

runLSTMonSegmentedGraphs("relu")
runLSTMonSegmentedGraphs("tanh")
runLSTMonSegmentedGraphs("sigmoid")
runLSTMonSegmentedGraphs("softmax")

runGRUonPaddedGraphs("relu")
runGRUonPaddedGraphs("tanh")
runGRUonPaddedGraphs("sigmoid")
runGRUonPaddedGraphs("softmax")

runGRUonSegmentedGraphs("relu")
runGRUonSegmentedGraphs("tanh")
runGRUonSegmentedGraphs("sigmoid")
runGRUonSegmentedGraphs("softmax")

runSRNNonPaddedGraphs("relu")
runSRNNonPaddedGraphs("tanh")
runSRNNonPaddedGraphs("sigmoid")
runSRNNonPaddedGraphs("softmax")

runSRNNonSegmentedGraphs("relu")
runSRNNonSegmentedGraphs("tanh")
runSRNNonSegmentedGraphs("sigmoid")
runSRNNonSegmentedGraphs("softmax")

runSKLearnClassifiersOnPaddedGraphs()
runSKLearnClassifiersOnSegmentedGraphs()
```

9.2.6 plotResults.py

```
import matplotlib as mpl
import matplotlib.pyplot as plt

mpl.use("Qt5Agg")

# PLOT TEXT-BASED DEEP LEARNING MODEL ACCURACIES
MLPwithReLu = [55.50,   51.2]
MLPwithTanh = [58.2, 48.01]
MLPwithSoftMax = [47.75,    48.01]
MLPwithSigmoid = [  52.25,  52.0]
LSTMwithReLu = [51.35, 52.0]
LSTMwithTanh = [52.25, 52.0]
```

```
LSTMwithSoftMax = [51.35, 52.0]
LSTMwithSigmoid = [53.15, 52.0]
GRUwithReLu = [52.25, 52.0]
GRUwithTanh = [50.45, 52.0]
GRUwithSoftMax = [55.86, 60.0]
GRUwithSigmoid = [52.25, 52.0]
SRNNwithReLu = [50.45,  52.0]
SRNNwithTanh = [49.55,  42.0]
SRNNwithSoftMax = [46.85, 50.0]
SRNNwithSigmoid = [ 52.25, 52.0]
DensewithReLu = [82.88, 68.0]
DensewithTanh = [53.15, 52.0]
DensewithSoftMax = [57.66, 52.0]
DensewithSigmoid = [52.25, 52.0]

textBasedTrainingMean = MLPwithReLu[0] + MLPwithTanh[0] + MLPwithSoftMax[0]  +
MLPwithSigmoid[0] + LSTMwithReLu[0] +  LSTMwithTanh[0] + LSTMwithSoftMax[0] +
LSTMwithSigmoid[0] + GRUwithReLu[0] + GRUwithTanh[0] + GRUwithSoftMax[0] +
GRUwithSigmoid[0] + SRNNwithReLu[0] + SRNNwithTanh[0] + SRNNwithSoftMax[0] +
SRNNwithSigmoid[0] + DensewithReLu[0] + DensewithTanh[0] + DensewithSoftMax[0] +
DensewithSigmoid[0]
textBasedValidationMean = MLPwithReLu[1] + MLPwithTanh[1] + MLPwithSoftMax[1]  +
MLPwithSigmoid[1] + LSTMwithReLu[1] +  LSTMwithTanh[1] + LSTMwithSoftMax[1] +
LSTMwithSigmoid[1] + GRUwithReLu[1] + GRUwithTanh[1] + GRUwithSoftMax[1] +
GRUwithSigmoid[1] + SRNNwithReLu[1] + SRNNwithTanh[1] + SRNNwithSoftMax[1] +
SRNNwithSigmoid[1] + DensewithReLu[1] + DensewithTanh[1] + DensewithSoftMax[1] +
DensewithSigmoid[1]
treeBasedTrainingMean = textBasedTrainingMean/20.0
treeBasedValidationMean = textBasedValidationMean/20.0

print("Avergae TA from Text-Based Deep Learning Models:", treeBasedTrainingMean)
print("Avergae VA from Text-Based Deep Learning Models:", treeBasedValidationMean)

xAxisLabels = ["Training Accuracy", "Validation Accuracy"]

fig, plot = plt.subplots(1)

plot.plot(MLPwithReLu, label="MLPwithReLu", marker="o")
plot.plot(MLPwithTanh, label="MLPwithTanh", marker="o")
plot.plot(MLPwithSoftMax, label="MLPwithSoftMax", marker="o")
plot.plot(MLPwithSigmoid, label="MLPwithSigmoid", marker="o")

plot.plot(LSTMwithReLu, label="LSTMwithReLu", marker="o")
plot.plot(LSTMwithTanh, label="LSTMwithTanh", marker="o")
plot.plot(LSTMwithSoftMax, label="LSTMwithSoftMax", marker="o")
plot.plot(LSTMwithSigmoid, label="LSTMwithSigmoid", marker="o")
plot.plot(GRUwithReLu, label="GRUwithReLu", marker="o")
plot.plot(GRUwithTanh, label="GRUwithTanh", marker="o")
plot.plot(GRUwithSoftMax, label="GRUwithSoftMax", marker="o")
plot.plot(GRUwithSigmoid, label="GRUwithSigmoid", marker="o")
plot.plot(SRNNwithReLu, label="SRNNwithReLu", marker="o")
```

```
plot.plot(SRNNwithTanh, label="SRNNwithTanh", marker="o")
plot.plot(SRNNwithSoftMax, label="SRNNwithSoftMax", marker="o")
plot.plot(SRNNwithSigmoid, label="SRNNwithSigmoid", marker="o")
plot.plot(DensewithReLu, label="DensewithReLu", marker="o")
plot.plot(DensewithTanh, label="DensewithTanh", marker="o")
plot.plot(DensewithSoftMax, label="DensewithSoftMax", marker="o")
plot.plot(DensewithSigmoid, label="DensewithSigmoid", marker="o")

plot.legend(loc="upper left")
xTickValues = [0, 1]
plt.title("Text-Based Deep Learning Model Results")
plt.xlabel('Accuracy Type')
plt.ylabel('Accuracy Score')
plt.xticks(ticks=xTickValues, labels=xAxisLabels)
plt.show()


# PLOT TREE-BASED DEEP LEARNING MODEL ACCURACIES USING UNHASHED NODES
fig2, plot2 = plt.subplots(1)

MLPwithReLu = [52.66, 57.58]
MLPwithTanh = [49.30, 48.48]
MLPwithSoftMax = [49.22, 42.42]
MLPwithSigmoid = [50.88, 57.58]

LSTMwithReLu = [56.25, 75.76]
LSTMwithTanh = [60.94, 70.00]
LSTMwithSoftMax = [72.66, 63.64]
LSTMwithSigmoid = [77.34, 70.00]

GRUwithReLu = [71.88, 66.67]
GRUwithTanh = [53.12, 51.52]
GRUwithSoftMax = [74.22, 60.61]
GRUwithSigmoid = [76.56, 75.76]

SRNNwithReLu = [61.72, 48.48]
SRNNwithTanh = [71.09,  66.67]
SRNNwithSoftMax = [100.0, 60.61]
SRNNwithSigmoid = [ 100.0, 70.0]

DensewithReLu = [50.78, 57.58]
DensewithTanh = [50.78, 57.58]
DensewithSoftMax = [96.88, 75.76]
DensewithSigmoid = [97.66, 57.58]


plot2.plot(MLPwithReLu, label="MLPwithReLu", marker="o")
plot2.plot(MLPwithTanh, label="MLPwithTanh", marker="o")
plot2.plot(MLPwithSoftMax, label="MLPwithSoftMax", marker="o")
plot2.plot(MLPwithSigmoid, label="MLPwithSigmoid", marker="o")
```

```
plot2.plot(LSTMwithReLu, label="LSTMwithReLu", marker="o")
plot2.plot(LSTMwithTanh, label="LSTMwithTanh", marker="o")
plot2.plot(LSTMwithSoftMax, label="LSTMwithSoftMax", marker="o")
plot2.plot(LSTMwithSigmoid, label="LSTMwithSigmoid", marker="o")
plot2.plot(GRUwithReLu, label="GRUwithReLu", marker="o")
plot2.plot(GRUwithTanh, label="GRUwithTanh", marker="o")
plot2.plot(GRUwithSoftMax, label="GRUwithSoftMax", marker="o")
plot2.plot(GRUwithSigmoid, label="GRUwithSigmoid", marker="o")
plot2.plot(SRNNwithReLu, label="SRNNwithReLu", marker="o")
plot2.plot(SRNNwithTanh, label="SRNNwithTanh", marker="o")
plot2.plot(SRNNwithSoftMax, label="SRNNwithSoftMax", marker="o")
plot2.plot(SRNNwithSigmoid, label="SRNNwithSigmoid", marker="o")
plot2.plot(DensewithReLu, label="DensewithReLu", marker="o")
plot2.plot(DensewithTanh, label="DensewithTanh", marker="o")
plot2.plot(DensewithSoftMax, label="DensewithSoftMax", marker="o")
plot2.plot(DensewithSigmoid, label="DensewithSigmoid", marker="o")
plot2.legend(loc="upper left")
xTickValues = [0, 1]
plt.title("Tree-Based Deep Learning Model with Segmentation Results")
plt.xlabel('Accuracy Type')
plt.ylabel('Accuracy Score')
plt.xticks(ticks=xTickValues, labels=xAxisLabels)
plt.show()
```

9.2.7 preliminaryTreeExperiments.py

```
import numpy as np
from Networks.MLP import MLP
from Networks.RNN import RNN
from Networks.NaiveBayes import NBClassifier
from Networks.DenseModel import runDenseModel
from Networks.SKLearnClassifiers import SGDClassify, rfClassify, SVMClassify
from ParsingAndEmbeddingLayers.Trees.TreeSegmentationLayer import
TreeSegmentationLayer
from ParsingAndEmbeddingLayers.Trees import TreeSegmentation as seg


hashed = True
# hashed = False
x_train_usum, x_train_umean, x_train_umax, x_train_umin, x_train_uprod, y_train =
seg.getUnsortedSegmentTrainData(hashed)
x_test_usum, x_test_umean, x_test_umax, x_test_umin, x_test_uprod, y_test =
seg.getUnsortedSegmentTestData(hashed)

x_train_sum, x_train_mean, x_train_max, x_train_min, x_train_prod, y_train =
seg.getSortedSegmentTrainData(hashed)
```

```
x_test_sum, x_test_mean, x_test_max, x_test_min, x_test_prod, y_test =
seg.getSortedSegmentTestData(hashed)

print("USING HASHED =", str(hashed).upper(), "DATA")
segmentCount = 40
segmentationLayer = TreeSegmentationLayer()
layers = [segmentCount, 64, 64, 2]
epochs = 30
lr = 0.05

# USING RELU ACTIVATION
print("RUNNING RNN MODELS USING UNSORTED SEGMENTATION AND HASHED NODES")
print("UNSORTED SEGMENT SUM AND RELU")
model1a = MLP(x_train_usum, y_train, layers, "relu", lr, epochs)
model1a.runFFModel(x_train_usum, y_train, x_test_usum, y_test)
print("UNSORTED SEGMENT MEAN AND RELU")
model1a.runFFModel(x_train_umean, y_train, x_test_umean, y_test)
print("UNSORTED SEGMENT MAX AND RELU")
model1a.runFFModel(x_train_umax, y_train, x_test_umax, y_test)
print("UNSORTED SEGMENT MIN AND RELU")
model1a.runFFModel(x_train_umin, y_train, x_test_umin, y_test)
print("UNSORTED SEGMENT PROD AND RELU")
model1a.runFFModel(x_train_uprod, y_train, x_test_uprod, y_test)
print()


# USING TANH ACTIVATION
print("UNSORTED SEGMENT SUM AND TANH")
model1b = MLP(x_train_usum, y_train, layers, "tanh", lr, epochs)
model1b.runFFModel(x_train_usum, y_train, x_test_usum, y_test)
print("UNSORTED SEGMENT MEAN AND TANH")
model1b.runFFModel(x_train_umean, y_train, x_test_umean, y_test)
print("UNSORTED SEGMENT MAX AND TANH")
model1b.runFFModel(x_train_umax, y_train, x_test_umax, y_test)
print("UNSORTED SEGMENT MIN AND TANH")
model1b.runFFModel(x_train_umin, y_train, x_test_umin, y_test)
print("UNSORTED SEGMENT PROD AND TANH")
model1b.runFFModel(x_train_uprod, y_train, x_test_uprod, y_test)
print()


# USING LOGSIGMOID ACTIVATION
print("UNSORTED SEGMENT SUM AND LOGSIGMOID")
model1c = MLP(x_train_usum, y_train, layers, "sigmoid", lr, epochs)
print("UNSORTED SEGMENT MEAN AND LOGSIGMOID")
model1c.runFFModel(x_train_umean, y_train, x_test_umean, y_test)
print("UNSORTED SEGMENT MAX AND LOGSIGMOID")
model1c.runFFModel(x_train_umax, y_train, x_test_umax, y_test)
print("UNSORTED SEGMENT MIN AND LOGSIGMOID")
model1c.runFFModel(x_train_umin, y_train, x_test_umin, y_test)
print("UNSORTED SEGMENT PROD AND LOGSIGMOID")
```

```
model1c.runFFModel(x_train_uprod, y_train, x_test_uprod, y_test)
print()

# USING SOFTMAX ACTIVATION
print("UNSORTED SEGMENT SUM AND SOFTMAX")
model1d = MLP(x_train_usum, y_train, layers, "softmax", lr, epochs)
model1d.runFFModel(x_train_usum, y_train, x_test_usum, y_test)
print("UNSORTED SEGMENT MEAN AND SOFTMAX")
model1d.runFFModel(x_train_umean, y_train, x_test_umean, y_test)
print("UNSORTED SEGMENT MAX AND SOFTMAX")
model1d.runFFModel(x_train_umax, y_train, x_test_umax, y_test)
print("UNSORTED SEGMENT MIN AND SOFTMAX")
model1d.runFFModel(x_train_umin, y_train, x_test_umin, y_test)
print("UNSORTED SEGMENT PROD AND SOFTMAX")
model1d.runFFModel(x_train_uprod, y_train, x_test_uprod, y_test)
print()



print("RUNNING RNN MODELS USING SORTED SEGMENTATION AND HASHED NODES")
# USING RELU ACTIVATION
print("SORTED SEGMENT SUM AND RELU")
model2a = MLP(x_train_usum, y_train, layers, "relu", lr, epochs)
model2a.runFFModel(x_train_sum, y_train, x_test_sum, y_test)
print("SORTED SEGMENT MEAN AND RELU")
model2a.runFFModel(x_train_mean, y_train, x_test_mean, y_test)
print("SORTED SEGMENT MAX AND RELU")
model2a.runFFModel(x_train_max, y_train, x_test_max, y_test)
print("SORTED SEGMENT MIN AND RELU")
model2a.runFFModel(x_train_min, y_train, x_test_min, y_test)
print("SORTED SEGMENT PROD AND RELU")
model2a.runFFModel(x_train_prod, y_train, x_test_prod, y_test)
print()


# USING TANH ACTIVATION
print("SORTED SEGMENT SUM AND TANH")
model2b = MLP(x_train_usum, y_train, layers, "tanh", lr, epochs)
model2b.runFFModel(x_train_sum, y_train, x_test_sum, y_test)
print("SORTED SEGMENT MEAN AND TANH")
model2b.runFFModel(x_train_mean, y_train, x_test_mean, y_test)
print("SORTED SEGMENT MAX AND TANH")
model2b.runFFModel(x_train_max, y_train, x_test_max, y_test)
print("SORTED SEGMENT MIN AND TANH")
model2b.runFFModel(x_train_min, y_train, x_test_min, y_test)
print("SORTED SEGMENT PROD AND TANH")
model2b.runFFModel(x_train_prod, y_train, x_test_prod, y_test)
print()


# USING LOGSIGMOID ACTIVATION
print("SORTED SEGMENT SUM AND LOGSIGMOID")
```

```
model2c = MLP(x_train_usum, y_train, layers, "sigmoid", lr, epochs)
model2c.runFFModel(x_train_sum, y_train, x_test_sum, y_test)
print("SORTED SEGMENT MEAN AND LOGSIGMOID")
model2c.runFFModel(x_train_mean, y_train, x_test_mean, y_test)
print("SORTED SEGMENT MAX AND LOGSIGMOID")
model2c.runFFModel(x_train_max, y_train, x_test_max, y_test)
print("SORTED SEGMENT MIN AND LOGSIGMOID")
model2c.runFFModel(x_train_min, y_train, x_test_min, y_test)
print("SORTED SEGMENT PROD AND LOGSIGMOID")
model2c.runFFModel(x_train_prod, y_train, x_test_prod, y_test)
print()

# USING SOFTMAX ACTIVATION
print("SORTED SEGMENT SUM AND SOFTMAX")
model2c = MLP(x_train_usum, y_train, layers, "softmax", lr, epochs)
model2c.runFFModel(x_train_sum, y_train, x_test_sum, y_test)
print("SORTED SEGMENT MEAN AND SOFTMAX")
model2c.runFFModel(x_train_mean, y_train, x_test_mean, y_test)
print("SORTED SEGMENT MAX AND SOFTMAX")
model2c.runFFModel(x_train_max, y_train, x_test_max, y_test)
print("SORTED SEGMENT MIN AND SOFTMAX")
model2c.runFFModel(x_train_min, y_train, x_test_min, y_test)
print("SORTED SEGMENT PROD AND SOFTMAX")
model2c.runFFModel(x_train_prod, y_train, x_test_prod, y_test)
print()
```

9.2.8 textExperiments.py

```
import numpy as np
from ParsingAndEmbeddingLayers.Text.TextParser import TextParser
from Networks.MLP import MLP
from Networks.RNN import RNN
from Networks.NaiveBayes import NBClassifier
from Networks.DenseModel import runDenseModel
from Networks.SKLearnClassifiers import SGDClassify, rfClassify, SVMClassify

tp = TextParser()
x_train, y_train, x_test, y_test = tp.getVectorizedTextData()
lstm = "lstm"
gru = "gru"
simpleRNN = "rnn"

def runMLPModels():
    """
    Run the Multilayer Perceptron Models on text
    """
    layers = [len(x_train[0]), 128, 128, 2]
    epochs = 10
    lr = 0.001
```

```
    mlp1 = MLP(x_train, y_train, layers, "relu", lr, epochs)
    metrics1 = mlp1.runFFModel(x_train, y_train, x_test, y_test)

    mlp2 = MLP(x_train, y_train, layers, "tanh", lr, epochs)
    metrics2 = mlp2.runFFModel(x_train, y_train, x_test, y_test)

    mlp3 = MLP(x_train, y_train, layers, "softmax", lr, epochs)
    metrics3 = mlp3.runFFModel(x_train, y_train, x_test, y_test)

    mlp4 = MLP(x_train, y_train, layers, "sigmoid", lr, epochs)
    metrics4 = mlp4.runFFModel(x_train, y_train, x_test, y_test)

    print("USING THE MULTI-LAYER PERCEPTRON")
    print("USING RELU")
    print("Average loss:", np.average(metrics1['trainingLoss']), "Average training
accuracy:",
    np.average(metrics1['trainingAccuracy']), "Average validation accuracy:",
    np.average(metrics1['validationAccuracy']), "\n")

    print("USING TANH")
    print("Average loss:", np.average(metrics2['trainingLoss']), "Average training
accuracy:",
    np.average(metrics2['trainingAccuracy']), "Average validation accuracy:",
    np.average(metrics2['validationAccuracy']), "\n")

    print("USING SOFTMAX")
    print("Average loss:", np.average(metrics3['trainingLoss']), "Average training
accuracy:",
    np.average(metrics3['trainingAccuracy']), "Average validation accuracy:",
    np.average(metrics3['validationAccuracy']), "\n")

    print("USING SIGMOID")
    print("Average loss:", np.average(metrics4['trainingLoss']), "Average training
accuracy:",
    np.average(metrics4['trainingAccuracy']), "Average validation accuracy:",
    np.average(metrics4['validationAccuracy']), "\n")

def runReluRNNs():
    """
    Run the RNN Models using ReLu Activation
    """
    reluModel = RNN("lstm", x_train, y_train, x_test, y_test, "relu")

    print("LSTM WITH RELU")
    reluModel.runModel(lstm, 256, 10, 10)

    print("GRU WITH RELU")
    reluModel.runModel(gru, 256, 10, 10)

    print("SRNN WITH RELU")
    reluModel.runModel(simpleRNN, 256, 10, 10)
```

```
def runSoftmaxRNNs():
    """
    Run the RNN Models using SoftMax Activation
    """
    softmaxModel = RNN("lstm", x_train, y_train, x_test, y_test, "softmax")

    print("LSTM WITH SOFTMAX")
    softmaxModel.runModel(lstm, 256, 10, 10)

    print("GRY WITH SOFTMAX")
    softmaxModel.runModel(gru, 256, 10, 10)

    print("SRNN WITH SOFTMAX")
    softmaxModel.runModel(simpleRNN, 256, 10, 10)

def runTanhRNNs():
    """
    Run the RNN Models using Tanh Activation
    """
    tanhModel = RNN("lstm", x_train, y_train, x_test, y_test, "tanh")

    print("LSTM WITH TANH")
    tanhModel.runModel(lstm, 256, 10, 10)

    print("GRU WITH TANH")
    tanhModel.runModel(gru, 256, 10, 10)

    print("SRNN WITH TANH")
    tanhModel.runModel(simpleRNN, 256, 10, 10)

def runSigmoidRNNs():
    """
    Run the RNN Models using Sigmoid Activation
    """
    sigmoidModel = RNN("lstm", x_train, y_train, x_test, y_test, "sigmoid")

    print("LSTM WITH SIGMOID")
    sigmoidModel.runModel(lstm, 256, 10, 10)

    print("GRU WITH SIGMOID")
    sigmoidModel.runModel(gru, 256, 10, 10)

    print("SRNN WITH SIGMOID")
    sigmoidModel.runModel(simpleRNN, 256, 10, 10)

def runDenseTextModels():
    """
    Run the densely connected models with the four different activation functiond
    """
    print("DENSE WITH RELU")
```

```python
    runDenseModel(x_train, y_train, x_test, y_test, "relu", 20, 10)

    print("DENSE WITH  SOFTMAX")
    runDenseModel(x_train, y_train, x_test, y_test, "softmax", 20, 10)

    print("DENSE WITH  TANH")
    runDenseModel(x_train, y_train, x_test, y_test, "tanh", 20, 10)

    print("DENSE WITH  SIGMOID")
    runDenseModel(x_train, y_train, x_test, y_test, "sigmoid", 20, 10)

def runGaussianNBC():
    """
    Run the Gaussian Naïve Bayes classifier on the text-based inputs
    """
    # convert the tensors to simple Python lists
    x, y = [], []
    for i in x_train:
        x.append(list(i.numpy()))
    for i in x_test:
        x.append(list(i.numpy()))

    # convert the y tensors into simple Python lists
    for i in y_train:
        i = list(i)
        label = i.index(1.0)
        y.append(int(label))
    for i in y_test:
        i = list(i)
        label = i.index(1.0)
        y.append(int(label))

    # Run the classifier on the converted lists
    nbc = NBClassifier(x, y)
    print("Gaussian NB Classifier Accuracy:", nbc.gaussianCrossValidation(x, y))
#86.875

def runSKLearnClassifiers():
    """
    Run the SGD, SVM and RF classifiers on the tex-based input
    """
    xTrain, yTrain, xTest, yTest = [], [], [], []
    for i in x_train:
        xTrain.append(list(i.numpy()))
    for i in x_test:
        xTest.append(list(i.numpy()))

    for i in y_train:
        i = list(i)
        label = i.index(1.0)
        yTrain.append(int(label))
```

```
    for i in y_test:
        i = list(i)
        label = i.index(1.0)
        yTest.append(int(label))

    print("SGD CLASSIFIER:", SGDClassify(xTrain, yTrain, xTest, yTest))
    print("RF CLASSIFIER:", rfClassify(xTrain, yTrain, xTest, yTest))
    print("SVM CLASSIFIER:", SVMClassify(xTrain, yTrain, xTest, yTest))


runMLPModels()
runReluRNNs()
runSoftmaxRNNs()
runTanhRNNs()
runSigmoidRNNs()
runDenseTextModels()
runGaussianNBC()
runSKLearnClassifiers()
```