Machine Learning Using Grammatical Evolution

Ranya El-Hwigi

University of Limerick

Faculty of Science and Engineering

Department of Computer Science and Information Systems

University of Limerick

Submitted to the University of Limerick

Final Year Project 2022

- Supervisor: Prof. Conor Ryan
  Department of Computer Science and Information Systems
  University *of* Limerick
  Ireland

- Project Implementation
  The implementation of the project can be found in the following GitHub Repository.

Declaration

I, the undersigned, hereby declare that this submission is entirely my own work, in my own words, and that all sources used in researching it are fully acknowledged and all quotations properly identified. It has not been submitted, in whole or in part, by me or another person, for the purpose of obtaining any other credit / grade.

Student Name: Ranya El-Hwigi

Student Number: 18227449

Signed

Limerick, 2022

Abstract

Evolutionary Algorithms have long been utilised in the detection and classification of the presence or absence of a disease. This project uses an Evolutionary Algorithm, Grammatical Evolution, to develop classifiers to predict whether a mammogram segment contains cancerous growth. Mammograms result from medical imaging used to see inside the breasts and are the most widely used method for detecting breast cancer. The aim is to develop classifiers with high accuracy and a low False Positive Rate to indicate a strong distinction between positive and negative instances.

A brand new variant of Grammatical Evolution, GRAPE, is used in the implementation along with Python libraries for Evolutionary Algorithms such as DEAP. For the set up of GRAPE, a grammar is produced as such that the classifiers developed use various mathematical functions and Haralick features that describe the mammogram to perform this classification. A fitness function is produced to use accuracy to measure and determine the classifier's fitness. Furthermore, multiple experiments are conducted to tune the hyperparameters and find the combination that gives the most favourable results.

The final results achieved on test data are an accuracy of 66.3% and a False Positive Rate of 46.8%. Results confidence is demonstrated through multiple runs of the program and a plot of the results which indicated slight variations in accuracy, attributed to the stochastic nature of Grammatical Evolution.

Contents

List of Tables

List of Figures

Listings

List of Abbreviations

AUC	Area Under the Curve
BDS	Biocomputing and Development Systems
BNF	Backus Normal Form
CAD	Computer Aided Detection
CADx	Computer Aided Diagnosis
CC	Cranio Caudal
CDV	Comma-separated values
CTG	Cardiotocography
DEAP	Distributed Evolutionary Algorithms in Python
EA	Evolutionary Algorithm
FNR	False Negative Rate
FN	False Negative
FPR	False Positive Rate
FP	False Positive
GB	Gigabyte
GE	Grammatical Evolution
GLCM	Grey Level Co-occurrence Matrix
GP	Genetic Programming

GRAPE	Grammatical Algorithm in Python for Evolution
MB	Megabyte
MLO	Mediolateral Oblique
ML	Machine Learning
SMOTE	Synthetic Minority Oversampling Technique
TNR	True Negative Rate
TN	True Negative
TPR	True Positive Rate
TP	True Positive

Chapter 1

Introduction

The project aims to use Grammatical Evolution (GE) to develop classifiers for mammograms to predict whether a mammogram contains cancerous growth. More specifically, using several thousand mammograms, the classifier should predict whether a segment of the mammogram contains cancerous growth. The classification of each segment will result in the presence or the absence of a suspicious region, therefore making this a binary classification issue and the resulting program a stage 1 Computer-Aided Detection (CAD) system.

A mammogram is an x-ray of the breast that radiologists use to determine whether or not the breast contains cancerous growth[12]. Computer-Aided Detection (CAD) systems have been developed to aid radiologists in detecting these growths. Recent studies have shown that CAD detection systems have improved radiologists' accuracy in breast cancer detection. However, these systems only act as a second reader and the final decision is made by the radiologist[2].

CAD algorithms consist of two stages; In stage one, the aim is to detect suspicious regions on the mammogram at a high sensitivity[2]. In stage two, the aim is to reduce the number of false positives by classifying the suspicious regions as mass or normal tissue[2]. *(See figure 1.1)*

Computer-Aided Diagnosis (CADx) systems are used after CAD to help radiologists with making recommendations for patient management[2]. For example

if a mass is suspected to be malignant, a biopsy is scheduled[2]. But if it's suspected to be benign, then a short-term follow up is scheduled[2]. Both CAD stage 2 and CADx are beyond the scope of this project.

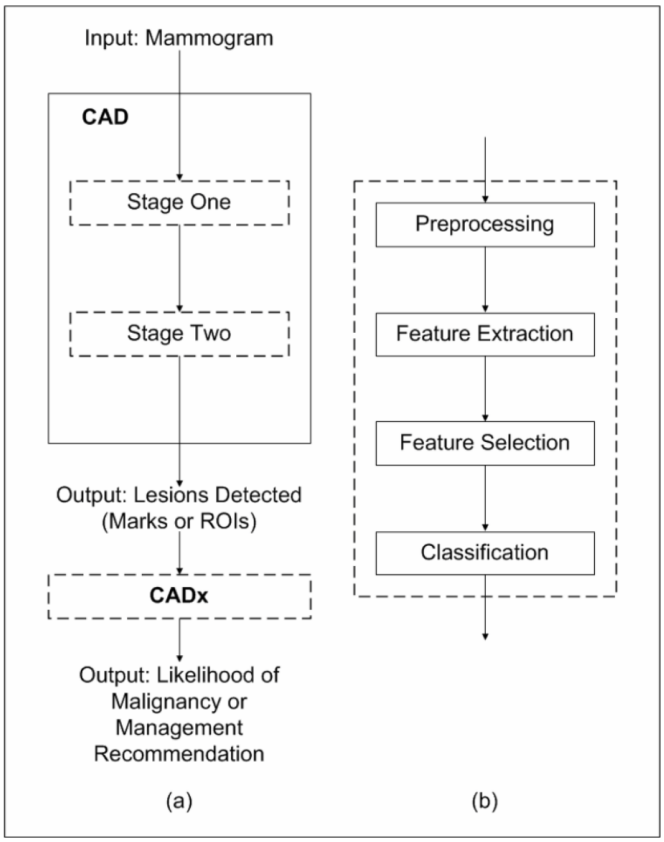

Figure 1.1: Flowchart showing the steps involved in the detection and diagnosis of mammographic abnormalities[2].

The complete dataset contains 405,280 segments from 5,066 mammograms, with 13 Haralick features extracted from the mammograms. Haralick features describe textural features for an image such as variance, entropy and correlation. Each segment has 104 columns to describe the different features, and of the 405,280 segments, 393,021 are positive, and 12,259 are negative.

GE is an evolutionary search algorithm similar to Genetic Programming (GP)

that evolves complete programs[13] It is made up of a context-free grammar like Backus Normal Form (BNF). BNF is a metasyntactic notation procedure that, among other things, is used to specify the syntax of computer programming languages.

GE will produce multiple individuals with binary genotypes. The phenotypes are then developed through a mapping technique that uses rules to map the genes to the grammar and produce a program, or segment of a program, that can be compiled and executed.

1.1 Motivations for Project

In 2020 breast cancer was the world's most prevalent cancer, with 2.3 million women being diagnosed with the disease and 685,000 deaths[14]. Breast cancer occurs in every country in the world, in women at any age after puberty making it a shared worry among women everywhere[14].

The importance of and motivation behind this project is the evidence that early detection of breast cancer can reduce the mortality rate of the disease. Early detection allows patients to get the help they need at an early stage of the cancer progression, and mammography is currently the most effective tool for early detection of breast cancer[2]. If cancer is located only in the breast, the 5-year survival rate of women with this disease is 99%. However, only 63% of women are diagnosed at this stage[15]. If cancer has spread to the regional lymph nodes, the mortality rate drops to 86%[15]. Then to 28% If it has spread to a distant part of the body[15].

However, mammography is not perfect as the detection of suspicious abnormalities is a repetitive and fatiguing task[2]. For every one thousand cases analysed by a radiologist, only 3-4 are cancerous, and hence an abnormality might be overlooked[2]. Therefore, radiologists fail to detect 10-30% of cancers, and approximately two-thirds of these false-negative results are due to missed lesions

that are evident retrospectively[2]. Computer-Aided Detectors can help reduce the number of negative cases seen by the radiologists and, in turn, reduce the number of false negatives.

The world health organisation has a global breast cancer initiative that aims to reduce the global mortality of the disease by 2.5% each year[14]. They predict this will avert 2.5 million deaths between 2020 and 2040[14]. Treatment has significantly advanced, but the most important thing remains to be early detection, so two things are vital here:

1. Awareness, getting women to perform breast checks regularly.
2. Invite people to regular mammogram screenings and have a fast turnaround time of checking those mammograms and taking action when cancer is present.

1.2 Project Objectives

- To research and understand mammography to a limited extent, including the views and segments used in reporting mammography results but excluding the process of mammography.
- To research and gain an in-depth understanding of Grammatical Evolution.
- To gain experience implementing Grammatical Evolution as part of a solution, which consequently results in gaining experience in:
  - Developing a BNF grammar.
  - Implementing fitness functions.
- To gain experience in developing a binary classifier.
- To research and gain experience conducting experiments on a classifier to gain confidence in the results.
- To gain experience extracting and plotting results of a Machine Learning (ML) classifier.

- To document the process and results of development.

1.3 Report Scope

The remaining chapters of this report are as follows:

Chapter Two - introduces background information on aspects of the project such as mammography, classification and evolutionary algorithms.

Chapter Three - gives an overview of Grammatical Evolution and its general workflow.

Chapter Four - discusses four research papers that inspired aspects of the implementation of this project.

Chapter Five - gives an overview and introduction to the tools used in the project.

Chapter Six - gives an overview of the dataset used in the project and the information stored in the dataset.

Chapter Seven - discusses the methodology of the project, including data engineering, grammar produced, fitness function, and GE set-up.

Chapter Eight - outlines the results achieved and the confidence of the result.

Chapter Nine - outlines the setup and results of ten different experiments conducted on the tuning of hyperparameters.

Chapter Ten - discusses the problems encountered during the project, comparison of the results with previous projects, future work for the project, and concludes the project.

Chapter 2

Background

The following chapter discusses some concepts introduced in the Introduction 1 such as mammography and classification, gives an overview of Evolutionary Computing as it is relevant to the remainder of this report, and also presents past solutions to the problem statement of the project.

2.1 Mammography

Mammography is specialised medical imaging that uses low-dose x-ray systems to see inside the breasts in order to diagnose and treat medical conditions[16]. It is performed on asymptomatic women to detect early and clinically unsuspected breast cancer[2].

During the mammography process, two views of each breast are recorded:

1. The Cranio Caudal (CC) view, which is a top to bottom view[17]. *(See figure 2.1)*
2. The Mediolateral Oblique (MLO) view, which is a side view taken at an angle[17]. *(See figure 2.2)*

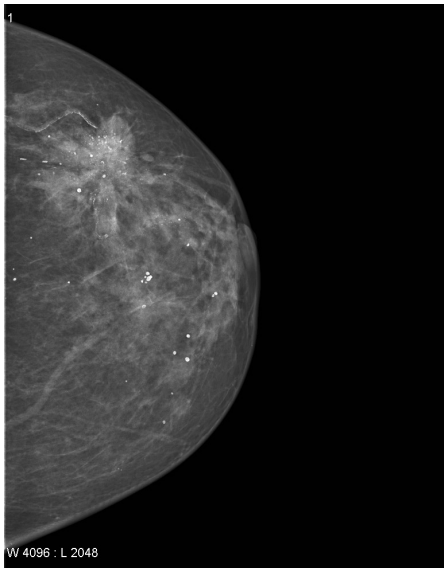

Figure 2.1: CC mammogram view [3]

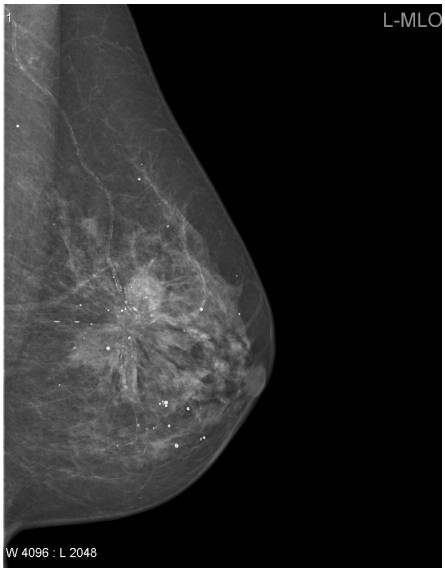

Figure 2.2: MLO mammogram view [3]

Breasts are made of two different types of tissues:

1. The parenchyma tissue, which is functional tissue and shows up white on mammograms[17].

2. The adipose tissue, which is non-functional fatty tissue and shows up nearly transparent[17].

A difficulty faced in mammography is dense breast tissue. Dense breast tissue refers to the appearance of breast tissue on a mammogram[18]. The non-dense breast tissue appears dark and transparent, making it easy to see through and detect cancer[18]. Whereas dense breast tissue appears as a solid white area, making it difficult to see through and increasing the chance cancer may go undetected with the dense tissue masking the cancer[18].

When analysing mammograms, radiologists are looking for two things:

1. Calcification's, which are calcium deposits and typically quite small white spots on mammograms[17]. Large, round or well-defined calcification's are

more likely to be non-cancerous. Whereas, tight clusters of tiny, irregular shaped calcification's might be cancerous[4]

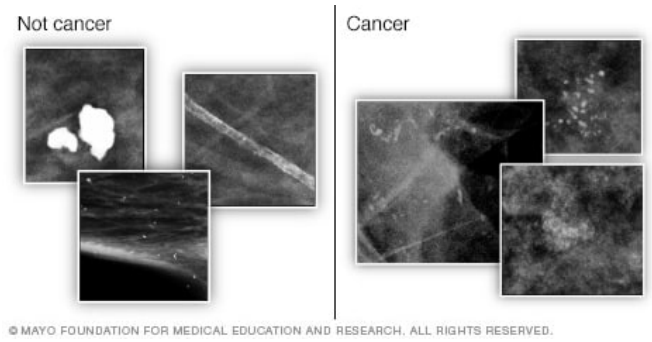

Figure 2.3: Mammograms comparing calcification's with cancerous abnormalities[4]

2. Masses, which are large space occupying lesions (abnormal changes in tissue)[17]. These masses differ from healthy breast tissue in their shape and edges.

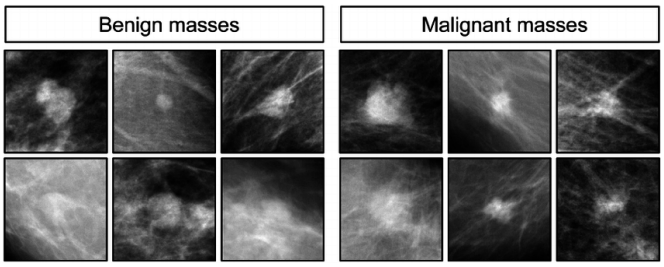

Figure 2.4: Mammograms comparing malignant (cancerous) and benign (not cancerous) masses[5]

2.2 Classification

Classification is the process of extracting meaningful and helpful information and then applying that information to support decisions made by the system[19]. In other words, it is the process of predicting the class of given data points[20]. The classes of data points are also known as targets, labels or categories[20]. This system is then referred to as a classifier.

When building a classifier, the goal is that after studying and training on a large labelled set of data, the classifier can now correctly label unseen data. For example, after studying and training on a large set of images of dogs and cats, the classifier should now be able to correctly predict whether an image, that it has not seen before, is of a dog or cat.

Since the data labels are visible to classifiers, classification belongs to the machine learning category of supervised learning. For this project, the labels of the data are [0, 1], where 0 refers to a negative case (no suspicious mass detected) and 1 refers to a positive case (suspicious mass detected).

There are many different types of classification:

- Binary Classification, here we have precisely two class labels, and only one label can be predicted for each example[20].
- Multi-Class Classification, here we have more than two class labels, and only one label can be predicted for each example[20].
- Multi-Label Classification, here we have two or more class labels, but one or more labels can be predicted for each example[20].
- Imbalanced Classification, we have this type of classification when the number of examples in each class is unequally distributed[20].

The types of classifications this project falls under are binary classification and imbalanced classification.

2.2.1 Binary Classification

Binary classification refers to classification tasks that have two class labels[20]. One class is the normal state which is labelled 0, and the other class is the abnormal state which is labelled 1[20].

In the case of this project, the normal case would be that there is no suspicious mass detected in the mammogram segment and therefore is labelled a 0. And

then the abnormal case would be that there is a suspicious mass detected in the mammogram and therefore is labelled a 1.

2.2.2 Imbalanced Classification

Imbalanced classification refers to classification tasks that have an unequal number of examples for each class[20]. Imbalanced classification typically happens with binary classification tasks with the majority of the examples in the data set belonging to the normal class (no suspicious mass detected)[20].

There are specialised techniques that can be used to solve the class imbalance problem by changing the composition of samples. There are three approaches to rectify the class imbalance issue:

- Oversample the data: this approach duplicates or generates new synthetic data from the minority/positive class and appends it to the dataset. It continues until both classes are of a similar amount or a specified ratio.
- Undersample the data: this approach removes or merges data from the majority/negative class in the dataset. It continues until the number of cases in the majority class meets the number of cases in the minority class or a specified ratio.
- Combine oversampling and undersampling: this approach oversamples the minority class to a specified ratio, then proceeds to undersample the majority class also to a specified ratio.

Within the approaches described above, many techniques can be used. The most popular of them are:

- RandomUnderSampling: randomly selects samples from the majority class and removes them from the dataset.
- RandomOverSampling: randomly selects samples from the minority class, duplicates them, and appends them to the dataset.

- SMOTE: stands for Synthetic Minority Oversampling Technique and works by generating new synthetic examples. SMOTE works by drawing a random sample from the minority class, then identifying the k-nearest neighbours of the point, followed by taking one of those neighbours and identifying the vector between the current data point and the selected neighbour, then multiplying the vector by a random number between 0 and 1, and finally adding this to the current data point[21].

In the case of this project, we can see that there is an imbalance in the distribution of the class labels as of the 405,280 samples in the data, 393,021 belong to class 0, and 12,259 belong to class 1.

2.3 Evolutionary Algorithms

Evolutionary Algorithms (EAs) are Machine Learning (ML) algorithms based on the simulation of evolution that are used to solve many different problems. In biology, evolution is the change in the characteristics of a species over several generations, which relies on the process of natural selection and survival of the fittest[22]. EAs try to replicate this process while solving a problem by maintaining a population of solutions, known as individuals, then using a fitness function to determine the fittest individuals, i.e. the individuals with the most desirable genes. Then recombining the best solutions to create increasingly better ones, producing a population on average fitter than the previous one.

2.3.1 Fitness Functions

Fitness functions are used to determine how fit an individual (solution) is by calculating how close it is to the optimum solution of the problem at hand[23]. Hence, fitness functions are tailored to the given problem. The output of this function helps EAs determine which individuals will be used to populate the next generation.

Some generic requirements of a fitness function are:

- It should be clearly defined so that the reader can clearly understand how the fitness is calculated[23].
- It should be implemented efficiently; otherwise, it can become the bottleneck of the algorithm, causing a reduction in the overall efficiency of the algorithm[23].
- It should quantitatively measure how fit a given solution is in solving the problem[23].
- The results it produces should be intuitive[23].

Some examples of fitness functions are:

- *Accuracy:* describes how well a solution performs across all labels. Its result is the ratio of correct predictions to the total number of predictions[24].
- *Area Under the Curve (AUC):* represents the degree or measure of separability; it tells how much the model is capable of distinguishing between classes[25]. An AUC of 0 indicates the classifier is performing worse than a random guess, a score of 0.5 indicates the classifier is no better than a random guess, and a score of 1 is the perfect classifier.
- *True Positive Rate (TPR):* True Positive (TP) is the number of positive classes correctly classified as positive. TPR is the probability that a positive case is detected and is calculated as the ratio between the number of Positive samples correctly classified to the total number of samples classified as Positive[24].
- *True Negative Rate (TNR):* True Negative (TN) is the number of negative classes there were correctly classified as negative. TNR is the probability that a negative case is detected and is calculated as the ratio between the number of negative samples correctly classified to the total number of samples classified as negative.

- *False Positive Rate (FPR):* FP is the number of negative classes that were incorrectly classified as positive. FPR is the probability that a negative case is missed and is calculated as the ratio of incorrectly classified negative cases to the total number of negative cases.

- *False Negative Rate (FNR):* FN is the number of positive classes that were incorrectly classified as negative. FNR is the probability that a positive case is missed and is calculated as the ratio of incorrectly classified positive cases to the total number of positive cases.

- *Confusion Matrix:* summarises prediction results in a classification problem. It calculates the TP, TN, FP and FN of the predictions. It is further illustrated below *(See figure 8.5)*

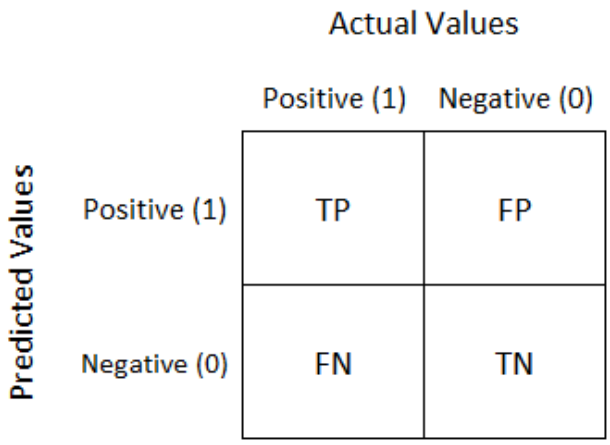

Figure 2.5: Confusion Matrix [6].

2.3.2 Evolutionary Loop

The evolutionary loop goes as follows:

1. *Initialisation:* we initialise the population by generating individuals to make up our first generation. The individuals can be represented as binary strings, syntax tree's etc., depending on the EA used.

2. *Fitness:* we measure the fitness of the individuals using a fitness function to determine how good a solution is. Usually, the population's fitness is low at the start, and as the population evolves, it gets increasingly better.

3. *Selection:* this is the first step in creating the next generation. We select individuals that we will use to create the next generation. Selection is a key part of the algorithm as we want to select the best performing individuals, so the more fit an individual is, the more like it will be chosen.

4. *Recombination (crossover):* this is the second step in creating the next generation. We take two individuals at a time and create an offspring by crossing over the individual elements (genes) from both individuals at a specified crossover point. Crossover continues until the next generation is formed. *(See figure 2.6)*

5. *Mutation:* this is the third and final step in creating the next generation. We pick individuals that have resulted from the previous step at random and make a small change to their genetic material. *(See figure 2.6)*

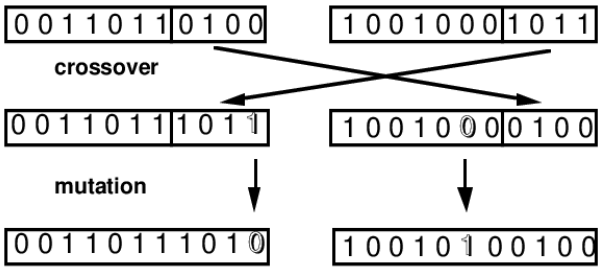

Figure 2.6: Example of crossover and mutation on binary represented individuals[7]

6. We now have our new generation, which is offspring from recombination, mutants and the best performers from the parent population.

7. We repeat steps 2-6 until we hit a completion tag or exhausted the amount of time we were willing to spend on this problem.

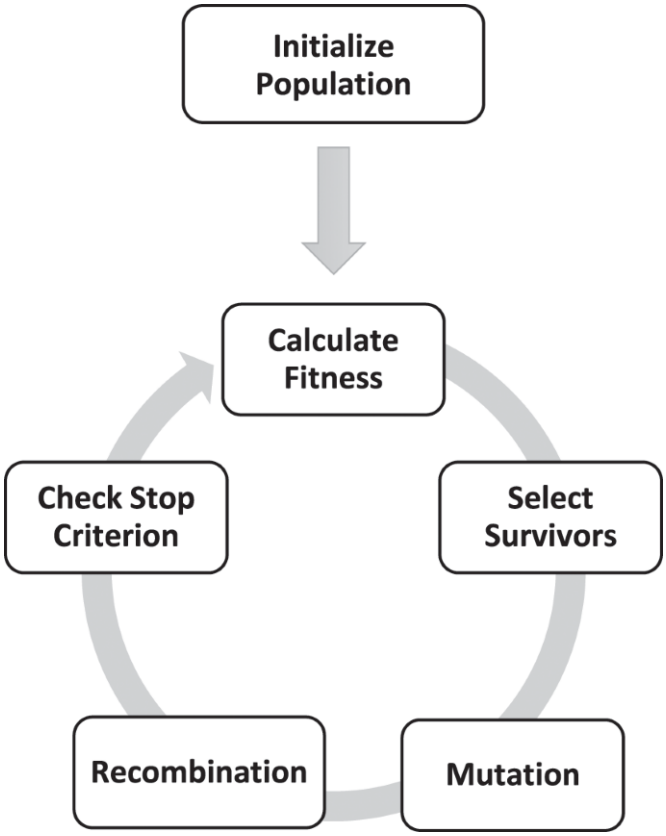

Figure 2.7: Diagram of the evolutionary loop[8]

There are many different EAs; among them are Genetic Algorithms (GA), Genetic Programming (GP), and Grammatical Evolution (GE). These algorithms can vary in structure, representation, evaluation etc., but essentially all EAs follow the same evolutionary loop discussed above.

2.4 Previous Solutions

2.4.1 Past FYP 2021

An existing solution to this project is previous work completed by Sean Morrissey as part of his final year project last year. Sean Morissey used ponyGE to develop a Stage-1 CAD system and received a true positive rate of 82%, and an Area

Under the Curve score of 0.63.

2.4.2 Fornells-Herrera et al. paper

Another existing solution is research completed by two research groups in Spain who also used grammatical evolution to classify mammograms. The research paper "Decision Support System For The Breast Cancer Diagnosis by Meta-learning Approach Based On Grammar Evolution"[26] proposed "Meta-learning Grammatical Evolution" which uses the Grammatical Evolution approach. Meta-learning can be seen as a black box that learns from outputs generated by other learners in order to make an improved prediction with a higher confidence level[26]. One of the most challenging tasks is the design of this black-box[26].

So they proposed what they called MGE, which is meta-learning grammatical evolution, as an intuitive way to define the relationships between level-0 models. By the end of their experiments, their MGE model had an error (percentage of misclassifications) of 28.57%.

Chapter 3

Grammatical Evolution

Grammatical Evolution (GE) is an evolutionary computation approach pioneered by Conor Ryan, J.J. Collins, and Michael O'Neill in 1998 at the Biocomputing and Development Systems (BDS) Group at the University of Limerick. GE is essentially a genetic algorithm and evolutionary computation technique that evolves complete programs[13].

A GE system consists of a grammar describing the structure of programs that can be developed or evolved, a fitness function to test how good those programs are, and a search engine, typically a GA, to produce variable-length binary strings. Using those elements, GE is then able to produce an output. Both the grammar and fitness function are catered to the problem at hand and provided as input to GE by the user. The GE process will be further explained in the following chapter.

3.1 Grammar

Grammar is used to specify legal sentences in a language. Usually, grammars are used to check if a program's syntax is legal. However, in GE, the grammar is used in a generative way, i.e. it is used to generate individuals in the population.

The grammar is an essential element in GE as it can have a significant impact on the quality of the solution depending on how well catered it is to the problem being solved. There are many grammar forms available, but a popular option, and the one GE uses, is Backus Naur Form (BNF).

A grammar is described by the set ⟨S,T,N,P⟩, where S is the start symbol, T is the terminal symbols, N is the Non-terminal symbols, and P is the production rules.

- Start symbol specifies where to start from when producing a program/program segment. It usually consists of terminal and non-terminal symbols.
- Terminal symbols are the specific elements of the program, e.g., 'if' for if statements. These cannot be broken down further.
- Non-terminal symbols are used in the generation of the program but do not appear in the program. These get broken down until they reach a terminal symbol.
- Production rules specify which symbols can be used to substitute others and create new symbol sequences.

Example grammar

⟨e⟩ := ⟨v⟩ |⟨o⟩⟨e⟩⟨e⟩
⟨o⟩ := + | - | * | /
⟨v⟩ := x | y | z

Where ⟨e⟩ is the start symbol, ⟨o⟩ and ⟨v⟩ are non-terminal symbols specifying the terminal symbols x, y, z, +, -, *, and /.

3.2 Mapping

In GE, individuals are binary strings; therefore, to calculate their fitness, the binary string is mapped to a program using the grammar specified. First, the

individual's genome is broken up into separate 8-bit long segments. That 8-bit long segment is converted to a decimal number and a list of decimals is generated to represent the entire genome. To get the phenotype, the mapping process starts with the start symbol in the grammar; then, takes the first (leftmost) decimal and mods it with the number of production rules that the start symbol has; the result is a number to indicate which production rule to choose. The mapping then continues by evaluating the leftmost non-terminal each time until we only have terminal symbols are left.

After mapping is complete the program is ready, it is compiled (if necessary), and its fitness is evaluated according to the fitness function specified.

Example mapping

Genome: 11010001001101101011000101010101
First 8-bit string: 11010001
Decimal: 209

Using grammar above, start symbol has 2 choices. 209%2 is 1 so GE goes with second choice as 0 would be the first choice. And continue on as such.

3.3 Evolution

Two essential elements of the evolutionary process in EAs are the initialisation of the population and the crossover of the individuals to create the offspring. GE implements initialisation, crossover and mutation differently from other EAs; this is further explained in this section.

3.3.1 Initialisation

We have three forms of initialisation in GE:

- Random initialisation: the initial populations' individuals are created by starting with the start symbol of the grammar, randomly picking between

the production rules, and continuing with the mapping process. The problem with this is that a large portion of the population will be very similar or even identical, leading to a very trivial population.

- Sensible initialisation: the initial populations' individuals are created by specifying a desired depth of the tree, an individual is then grown until it reaches that desired depth. The first node chooses randomly between production rules, and then depending on its choice, it may or may not have a specific production rule forced on its next node. However, this can lead to a left-handed or right-handed bias depending on when the depth is measured.

- Position Independent Grow initialisation: The initial population individuals are created by starting with the start symbol and then randomly choosing non-terminals to map. This is an improved version of sensible initialisation, its non-recursive and results in a better balance of trees.

3.3.2 Crossover and Mutation

GE uses a crossover called effective crossover and a form of mutation called effective mutation. Because GE has varying length individuals, the length of the genome can be broken down into different sections:

- Actual length: the full length of the genome.

- Effective length: the length to where the mapping was completed.

- Tail: the leftover part of the genome that was not included in the mapping.

Therefore, there is an issue of performing crossover or mutation on the tail, which has not had the fitness evaluated. Effective crossover ensures that the crossover is performed on the effective length and effective mutation ensures that mutation is performed within the effective length.

Chapter 4

Literature Review

4.1 Building a Stage 1 Computer Aided Detection for Breast Cancer using Genetic Programming

Building a Stage 1 Computer-Aided Detection for Breast Cancer using Genetic Programming[17] was research completed by Conor Ryan, Krzysztof Krawiec, Una-May O'Reilly and Jeannie Fitzgerald.

The paper described a fully automated workflow for performing stage 1 detection of breast cancer with genetic programming at its foundation. They operated with raw images and then extracted features to present to the GP, which then evolved the classifiers. The approach Ryan et al. took was to compare textural breast asymmetry as an indicator of cancerous growths. The results were impressive, with 100% accuracy on unseen test data and an FPR of only 1.5.

This paper helped with gaining an understanding of the data that is used in this project, as it is very similar to the data Ryan et al. used. However, Ryan et al.'s workflow is positioned at the data collection stage, so they generated the textural features themselves, whereas the textural features are already generated for this project data.

4.2 Classification of Fetal Heart Rate using Grammatical Evolution

Classification of fetal heart rate using grammatical evolution[9] is research that has been completed by Dimitris Gavrilis, Geroge Georgoulas, Ioannis G. Tsoulos, and Euripidis Glavas.

Feature extraction and classification of CTG[1] is a difficult task for obstetricians. This paper proposes an innovative method for feature construction for the classification of CTG, which is based on information extracted from the Fetal Heart Rate signal. The proposed method uses GE to construct new features from existing ones through nonlinear transformations. Once the new features are constructed, a new train and test set are created according to the new features. The new feature is then given a fitness value, the test error of a supervised classifier, which determines whether it will continue to the next generation. The method was tested on a data set of intrapartum cases, and it achieved an accuracy of 92.5%.

S ::= <expr>

<expr> ::= <expr> <op> <expr> | <func> (<expr>)

| <digit> | x_1 | x_2 | ... | x_n

<op> ::= + | - | * | /

<func> ::= sin | cos | exp | log

<digit> ::= 0 | 1 | 2 | ... | 9

Figure 4.1: BNF grammar to construct new features from existing data[9]

This paper inspired the grammar used in the projects implementation as both projects deal with medical classification of numerical data.

[1]Cardiotocography (CTG) uses sound waves to monitor a baby's heart rate and a mother's contractions during pregnancy.[27]

4.3 Computer-Aided Detection and Diagnosis in Mammography

Computer-Aided Detection and Diagnosis in Mammography[2] is research that has been completed by Mehul P. Sampat, Mia K. Markey, and Alan C. Bovik.

This paper focuses primarily on the different technological methods available for detecting and diagnosing cancer in mammograms. They detail computer-aided detection (CAD) and computer-aided diagnosis (CADx) of mammographic abnormalities. Each is also broken down into the detection of masses and calcifications separately. Sampat et al. also discuss the commercial CAD and CADx systems available and recent advances, and future directions in the field.

This paper helped with realising and deciding on a scope for the project as it broke down the different stages mammogram classification goes through.

4.4 Hyper-Parameter Optimization for Improving the Performance of Grammatical Evolution

Hyper-Parameter Optimization for Improving the Performance of Grammatical Evolution[1] is research that was completed by Hao Wang, Yitan Lou, and Thomas Back.

Along with defining a BNF grammar and a fitness function, GE also has several hyper-parameters that the user must tune to best fit their solution. In this paper, Wang et al. optimise these hyper-parameters for a PoneyGE2-system[1] using a variant of the Efficient Global Optimisation (EGO) algorithm.

Wang et al.'s approach was tested on four different problems, including banknote classification. Their results showed that the suggested approach improved

[1]PoneyGE2 is an earlier implementation than GRAPE of GE in python.

Hyperparameter	Type	No. of Options
Initialisation	Discrete	3
Crossover Rate	Real [0,1]	-
Crossover Type	Discrete	4
Mutation Rate	Real [0,1]	-
Mutation Type	Discrete	3
Selection Type	Discrete	2
Tournament Size	Integer {1,..,50}	50
Selection Proportion	Real [0,1]	-
Max INIT Tree Depth	Integer {5,..,25}	20
Max Tree Depth	Integer {20,..,100}	80
Max Genome Length	Integer {100,..,1000}	900

Table 4.1: Hyper-parameters considered in the tuning process[1].

Problem	Settings	Performance Measure	Relative Improvement
StringMatch	Default	4.8802	-
	LHS	1.0421	78%
	MIP-EGO	0	100%
Regression	Default	0.0326	-
	LHS	0.0306	6.1%
	MIP-EGO	0.0167	48.8%
Classification	Default	0.3798	-
	LHS	0.4560	20.1%
	MIP-EGO	0.4742	24.9%
Pymax	Default	357.1	-
	LHS	415.04	16.2%
	MIP-EGO	958.0	168.4%

Table 4.2: The mean performance of final hyper-parameter settings found by the MIP-EGO algorithms is compared to 1) the hyper-parameter obtained in the initial design of the experiment and 2) the default setting in PoneyGE2[1].

GE's performance between 25% and 168% on all test problems. They also found that the overall resulting best hyper-parameters differed from the defaults used in PonyGE2.

Although the improvement achieved for the classification problem was relatively small and not very helpful with my results, this paper helped with gaining a better understanding of all the hyper-parameters in GE and how the values they are set to affect the outcome.

Chapter 5

Environment and Tools

Various tools and environments were use in the development of this project, they are: GRAPE, Python, Jupyter Notebook, DEAP, Pandas, Matplotlib, and GitHub. Each were instrumental in bringing this project together for various reason which are discussed in the chapter.

5.1 GRAPE

GRAPE stands for GRammatical Algorithm in Python for Evolution. It is a GE system that's brand new as it was only developed in the summer of 2021 at the University of Limerick.

GRAPE is an implementation of Grammatical Evolution in Python *(further discussed below in 5.2)* and DEAP *(further discussed below in 5.4)*. Typically the search engine used in GE is a GA. However, GRAPE uses DEAP as its search engine giving us access to any algorithm that DEAP uses to produce binary strings. As well as that, GRAPE provides a mapper, crossover, and mutation.

5.2 Python

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics[28]. It is among the most popular programming languages ranking first in a 2021

ranking of top computer languages[29]. Python was used as the main development language during the project for the following reasons:

- It gave access to many useful libraries for the project like DEAP, GRAPE, Pandas etc.
- It is simple and easy to understand and allows for fast data validation[30].
- Python for Machine Learning is versatile; it can run on any platform[30].
- To gain experience developing in Python because of its growing popularity.
- The Grammatical Evolution library used, GRAPE, is developed in Python.

5.3 Jupyter Notebook

Jupyter Notebook is a web-based interactive computing platform that has become very popular for use by data scientists and machine learning projects[31].

A Jupyter Notebook comprises of cells that can either contain code or text. When running a code cell, the code is passed to a backend kernel which runs the code and returns the results. If that piece of code has an output it will be output directly after the cell.

Jupyter Notebook was used for the project because it provides an easy-to-use, interactive data science environment that does not only work like an IDE (Integrated Development Environment) but also as a presentation tool[32]. It also allows users to run specific cells of code instead of rerunning the entire code, to include text between those cells adding detail and explanation to the code, and it also saves the output of the code under the respective cells in the notebook since its last run, so there is no need to rerun the code. Therefore, it is the perfect environment for presenting a project that has an equal emphasis on both coding and research.

5.4 DEAP

DEAP stands for Distributed Evolutionary Algorithms in Python. It is an evolutionary computation framework that is used for rapid prototyping and testing of ideas[33].

There's two key parts to DEAP:

- Creator Module, which helps us create classes and types for the evolutionary algorithm we want to create. E.g., create the individuals, create the fitness function etc.
- Toolbox Module, which contains all our evolutionary algorithm operators. E.g., the fitness function, the selection method etc.

DEAP also offers features to keep track of statistics on the evolutionary process. One exciting feature is "Hall of fame". It lists all the best individuals it has seen so far in a particular run throughout the generations. Hall of fame was helpful to check whether fit individuals are being lost.

DEAP can help simplify the process of developing evolutionary algorithms; therefore, it was used alongside Grammatical Evolution to develop the classifier and keep statistics on the evolutionary process.

5.5 Pandas

Pandas is a Python package that provides quick, flexible, and expressive data structures[34] and is most widely used for data science and machine learning tasks[35].

Pandas data structures have been designed to make working with relational or labelled data easy and intuitive[34]. It is built on top of Numpy, another Python package that provides support for multi-dimensional arrays[35].

Pandas was used because the data is labelled data, which Pandas data structures have been designed for. Also, it is one of the most popular data-wrangling packages, so it works well with many other data science modules in the Python ecosystem[35]. It also made it easy to do a lot of the time consuming and repetitive tasks which are associated with data, like data inspection, data cleansing, data visualisation etc[35].

5.6 Matplotlib

Matplotlib is a comprehensive library for creating static, animated, and interactive visualisations in Python using many different types of data[36].

Its goal is to allow users to create simple plots with a few comprehensive commands by skipping setup requirements needed by other libraries, which overcomplicate the process[37].

Matplotlib was used to make graphs to visualise both the data and the results of the developed classifier.

5.7 GitHub

GitHub is a code hosting platform for version control and collaboration[38]. It allows users to create repositories that are used to organise a single project. A repository contains folders, files, images, data sets and anything else the project needs[38]. Users can alter the code, or other files in the repository, locally on their machines and then push those changes to the remote GitHub repository after they are sure they want to make the changes.

Even though this is not a collaborative project, I used GitHub to host my code for the following reasons:

- Safety: if I made any mistakes locally, I still had the previous version of the code accessible to revert back to it.
- Portability: I was able to pull and run my code on any other machine and not just limited to mine.
- Data storage: the data for this project is quite large, so I stored it on GitHub and, through Pandas, I directly loaded it into my Jupyter Notebook from a link to the data file in the GitHub repository. Doing this saved me storage space on my machine.
- Presentation: GitHub offers ReadMe files which are then displayed nicely in the repository under each folder. Therefore, I used this feature to present my project; it gave me the ability to keep all the files relating to the project together and to thoroughly explain through ReadMe files what is stored in each folder.

Chapter 6

Dataset

The dataset used in this project was provided by the project supervisor. Two files were provided:

- A large (size ≈ 1GB) csv file containing the full dataset.
- A smaller (size ≈ 65MB) csv file containing a subset of the full dataset.

A general description of the dataset would be that each row corresponds to a specific segment of each breast at a specific mammogram view. Each column in a row then contains information on the patient, the study, the mammogram view performed, the segment it relates to, descriptions of many Haralick features at multiple rotations of the segment in question and the same segment on the opposite breast, and a label.

6.1 Overview of Dataset

6.1.1 Size

The only difference between the two datasets provided is the size, i.e. number of rows. The larger file contains 405,280 rows, and the smaller file contains 24,999 rows.

Each row corresponds to a segment of a breast of a patient; each breast has 20 segments *(discussed in 6.1.3)* and is recorded in multiple views but most

commonly in just two views CC and MLO *(discussed in 2.1)*. So approximately every 80 rows refer to one patient.

Given that 80 rows refer to one patient, approximately 5,066 patients are recorded in the entire dataset and 312 in the sample dataset.

6.1.2 Columns

The dataset contains 111 columns for each row. The columns are:

- *PatientID:* The id of the patient to whom this segment belongs.
- *StudyID:* The id of the study from which these mammograms were obtained.
- *PatientAge:* The age of the patient to whom this mammogram segment belongs.
- *ImgID:* The id of the mammographic image used in this row, i.e. the image this segment was taken from and Haralick features constructed from.
- *View:* refers to the mammogram view this image was taken in, i.e. CC, MLO, etc.
- *SegmentPosition:* The segment out of the possible 40 this row relates to. *(discussed in 6.1.3)*
- *x0 - x103:* 104 columns describe the Haralick features of two segments of the breast. These descriptions include the rotation of the segments at four different orientations. The first 52 columns x0-x51 relate the segment that is currently of interest on either the left or right breast, and the remaining 52 columns x52-x103 relate the same segment on the opposite breast.
- *Label:* The outcome of the screening of this segment. i.e. $0 \rightarrow$ not cancerous or $1 \rightarrow$ cancerous.

6.1.3 Mammogram segments

In general, most automated approaches to mammography divide the images into segments that can be used for analysis[17]. LesionPosition is a code that is similar to a spreadsheet cell address. It divides an image of a breast into columns A to E and rows 1 to 4. A special address "M" is reserved for the middle cell, referring to the areola/nipple.

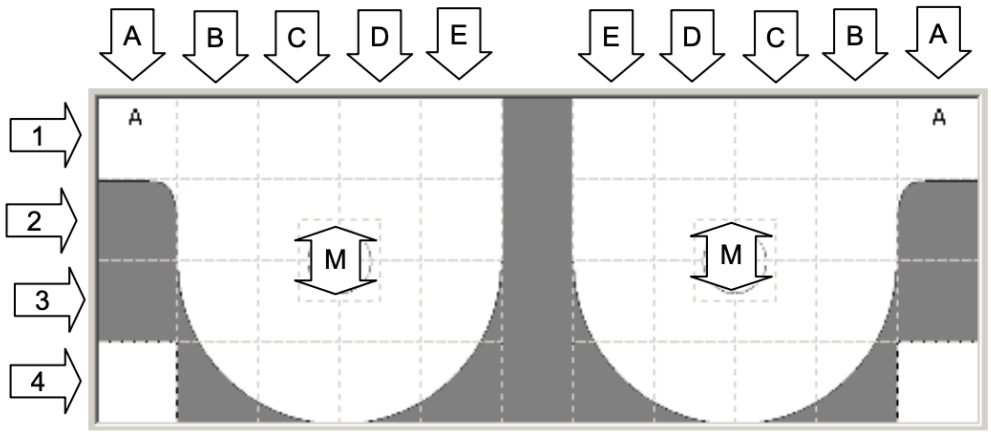

Figure 6.1: Mammogram segments using LesionPosition[10]

The dataset uses the LesionPosition code hence there are 40 possible segments: LA1, RA1, LB1, RB1, LC1, RC1, LD1, RD1, LE1, RE1, LA2, RA2, LB2, RB2, LC2, RC2, LD2, RD2, LE2, RE2, LA3, RA3, LB3, RB3, LC3, RC3, LD3, RD3, LE3, RE3, LA4, RA4, LB4, RB4, LC4, RC4, LD4, RD4, LE4, RE4.

6.1.4 Haralick Features

As mentioned in the section 6.1.2, 104 columns of the dataset are used to store information on the Haralick features of a breast segment taken from a mammogram image.

Haralick features describe textural features for an image such as variance, entropy and correlation. They are calculated from a grey level co-occurrence matrix (GLCM) which is a standard method to represent image texture, as it is simple to implement and results in a set of interpretable texture descriptors[39].

Texture analysis is a way of describing the spatial distribution of intensities, which makes it a valuable method for the classification of similar regions in different images[39]. It has gained popularity for many medical imaging applications, including oncology (the study of cancer)[39].

6.1.4.1 Grey Level Co-occurrence Matrix (GLCM)

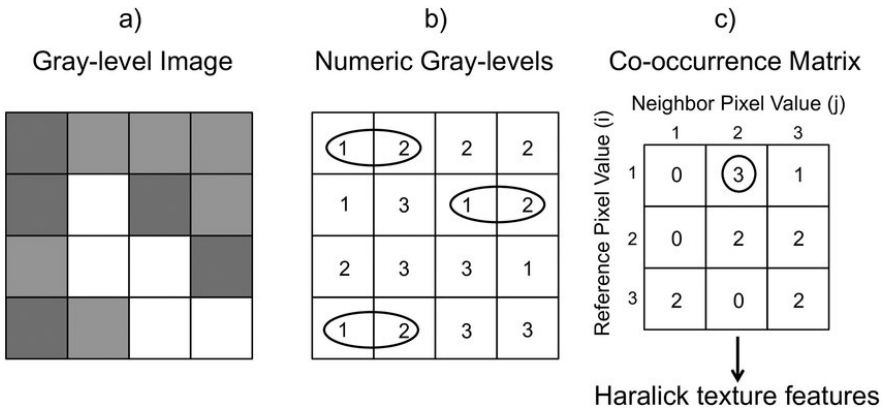

Figure 6.2: An example of the process of calculating a GLCM from a 4x4 image.[11]

GLCM is one of the earliest techniques for image texture analysis[40]. It examines the spatial relationship between pixels and defines how often a combination of pixels is present in an image given a specified direction and distance[41].

6.1.5 Label Distribution

The label tells us whether or not the segment of the mammogram in this row is suspicious/cancerous. Given that the normal case is that it is not suspicious, there are a lot more of the negative (0) labels than the positive (1) labels in both the datasets. Hence we have an imbalanced distribution.

For the large dataset, there are 393,021 negative cases (label 0) and 12,259 positive cases (label 1). Moreover, for the smaller dataset, there are 24,018 negative cases (label 0) and 981 positive cases (label 1).

Chapter 7

Methodology

The methodology of this experiments involved many elements, among them: Data Exploration, Data Preprocessing, Grammar production, Fitness Function production, and setting up and running the GE environment. Each of these elements is further discussed in this chapter. The implementation discussed is that of which received the best results.

7.1 Data Exploration

Data exploration is the first step of data analysis in which data analysts use a mix of manual and automatic techniques in order to understand the nature of the data better, and reveal patterns and points of interest[42]. It is an essential practice in ML so that we can correctly interpret the results achieved and because the model accuracy can suffer if data is not adequately explored first[42]. The ultimate goal is to provide data insights that will inspire subsequent feature engineering and the model building process[42].

Through data exploration, I understood and uncovered a lot of information about the data as I only received raw data files. Hence, what I discussed in section 6.1 I discovered through performing data exploration.

The steps I took to explore the data are as follows:

- First, I started off by loading both the large and small datasets into separate pandas (discussed further in section 5.5) dataframes from csv using the function *pandas.read_csv(⟨filepath⟩)*.

- Second, using the *head()* and *tail()* functions I took a look at and familiarised myself with the data.

- Third, using the *describe()* function, I got a statistical summary of the numerical data in the dataset. The statistical summary helped me discover if there were any missing values, outliers, etc., that I would need to deal with in the data preprocessing phase. I found that there were no missing values or outliers that needed to be dealt with.

	PatientAge	x0	x1	x2	x3	x4
count	405280.000000	405280.000000	405280.000000	405280.000000	405280.000000	405280.000000
mean	60.345638	0.546924	25.173811	2.148299	618.533917	4.655893
std	8.212397	0.453399	37.215509	1.555457	932.928751	5.137900
min	30.000000	0.000000	-1.000000	-0.000204	0.000000	-0.000000
25%	54.000000	0.001709	-0.309129	0.975508	0.000301	0.216874
50%	61.000000	0.821703	0.000000	0.995603	0.004511	0.952885
75%	66.000000	0.987784	46.411842	3.961258	1029.679274	10.832815
max	93.000000	1.000000	1974.782692	6.718097	12581.824994	14.739384

Figure 7.1: Output from describe function on larger dataset.

- Fourth, using Pandas *count()* function, I calculated and visualised the difference in size between the two datasets. As mentioned in 6.1.1, I found that the larger set consisted on 405,280 rows and the smaller set 24,999. I focused on the large dataset for the remainder of the steps.

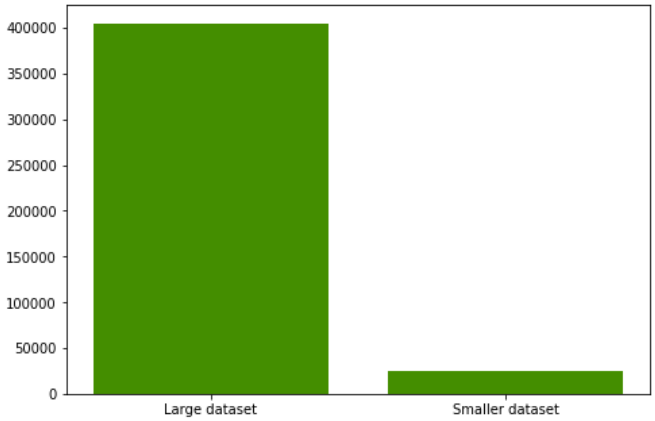

Figure 7.2: Visualisation of the difference in size between both datasets.

- Fifth, using Pandas *dtypes* function, I extracted all the different column names and their types. I found that there are 111 columns, 5 of which are of type object (which is a string in Pandas), 2 are of type integer and the remainder 104 of type float. I made note of the ones I think are not relevant to remove them later.

Column Name	Type
PatientID	object
StudyID	object
PatientAge	int64
ImgID	object
View	object
SegmentPosition	object
x0 - x103	float64
Label	int64

Table 7.1: Datasets column names and their corresponding types.

- Sixth, using Pandas *value_counts()* function, I calculated and visualised the label distribution in the dataset, I found that the dataset was imbalanced with 393,021 "0" labels and 12,259 "1" labels.

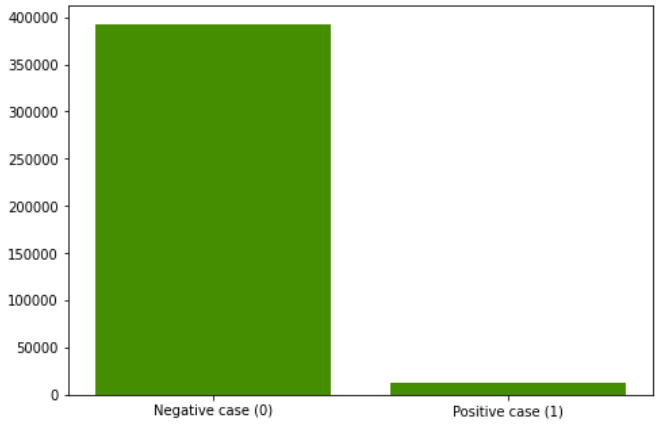

Figure 7.3: Visualisation of the label distribution in the large dataset.

- Seventh, using Pandas *unique()* function, I extracted the different views that were used for the data in the dataset and calculated their popularity. I found that there are 5 views included, they are: CC (*discussed in 2.1*), MLO (*discussed in 2.1*), XCCL[1], ML[2], LM[3] I discovered that CC and MLO dominate the dataset with 99.9% of the data being of either of those two views.

[1]XCCL is a supplementary view that is an exaggerated CC view. It is often done when a lesion is suspected on a MLO view but it can't be seen on the CC view[43].

[2]ML is a supplementary view that shows less breast tissue and pectoral muscle than MLO[44].

[3]LM is a supplementary view that allows the medial breast to be more carefully evaluated[45].

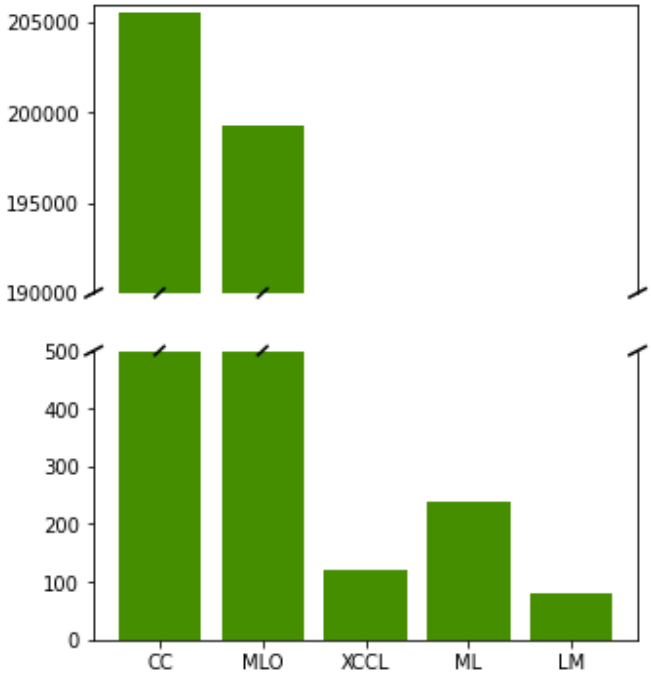

Figure 7.4: Visualisation of the mammogram views distribution in the large dataset.

- Finally, using Pandas *unique()* function again, I extracted the different segments that are included in the dataset. I found that there are 40 segments, the same as the ones mentioned in section 6.1.3.

7.2 Data Preprocessing

Data preprocessing is transforming raw data into an understandable format[46]. It involves preparing the raw data, making it suitable for building and training a machine learning model.

There are four majors tasks involved in preprocessing:

- Data cleaning: This is the process of removing incorrect, incomplete, and inaccurate data from the datasets[46].

- Data integration: is combining multiple sources into a single dataset. Need

to consider schema integration, entity identification problems, and detecting and resolving data value concepts[46].

- Data reduction: helps reduce the volume of the data making the analysis easier while still producing the same result. Some techniques are dimensionality reduction, numerosity reduction, and data compression[46].
- Data transformation: involves changing the format or structure of the data. Some methods are smoothing, aggregation, discretisation, and normalisation[46].

Through data exploration (section 7.1), I discovered the data is more or less already prepared and only requires reduction and transformation in some areas. The areas that require attention in this section are:

1. Columns that do not describe Haralick features, excluding the label column, should be excluded from the data fed to GE. Removing those extra columns can reduce the volume of the data without impacting the results, given that these columns provide no information on Haralick features.
2. The imbalanced label distribution in the data needs to be balanced. Balancing the labels is essential to help train the model without causing an unconscious bias towards a particular label in the outcome. A bias can affect test fitness by performing poorly in the minority class.
3. The unpopular views (XCCL, ML, LM) needed to be removed. Given that the two main views, CC and MLO, attribute to 99.9%, there is insufficient data on the remaining views to train a program properly. So to further reduce dimensionality, they were dropped, and all focus was put on CC and MLO views.
4. Finally, the data needs to be divided into test and train sets for GE to use.

7.2.1 Data Reduction

As mentioned previously, the dataset consists of 405,280 rows and 111 columns. Resulting in 44,986,080 cells containing data, so any reduction that can be made can help bring down that load.

First, I focused on dropping the unpopular mammogram views XCCL, ML, and LM. Using Pandas *isin()* function I was able to reduce the number of rows from 405,280 to 404,840, eliminating 48,840 cells.

Next I moved on to dropping the columns I deemed unnecessary for the classification of the segments. As a result, using Pandas *drop()* function, I dropped the following Columns:

- PatientID
- StudyID
- PatientAge
- ImgID
- View
- SegmentPosition

Hence, the remaining columns were the Haralick features x0-x103 and the Label, bringing down the total columns to 105 and eliminating 2,429,040 cells.

7.2.2 Data Transformation

Two transformation's needed to be made to the data. The data had to be either extended or shrunk to balance the occurrences of negative and positive cases, and it had to be split into train and test sets for GE to accept it as input.

7.2.2.1 Resampling

As mentioned previously, there is a significant imbalance in the data between the positive and negative cases, with a positive case occurring in only 3% of the data. This is problematic; class imbalance makes the classification of mammograms more difficult due to an inherent bias towards the majority class. Another problem with class balance is that the EA decides the "intelligent" thing to do

would be to classify everything as the majority class; this is further explored in section 10.1.2.

Given that the number of positive cases is small (12,259 cases), I decided that going with the undersampling approach and reducing the data to approximately 25,000 cases would result in a significant data loss. I decided against the combination of both undersampling and oversampling because, again, I wanted to preserve all the data I had. But also because I did not see it fitting to undersample and delete a large amount of real data of negative cases, and to then duplicate or synthetically generate new instances. Hence, I oversampled the data.

State	Negative Cases	Positive Cases	Total Cases
Original	393,021	12,259	405,280
Oversampled	393,021	393,021	786,042

Table 7.2: Result of oversampling the dataset.

For the oversampling technique, I decided to go with SMOTE as Random Oversampling would result in the 12,259 cases getting duplicated ≈ 32 times to reach the number of negative cases in the data. The problem with using Random Oversampling is that the high amount of duplication could result in overfitting[1] Whereas SMOTE oversamples the dataset by synthetically generating new data points which would be more realistic to a real balanced dataset of positive and negative mammogram cases.

7.2.2.2 Train and Test Sets

An important step in the data preprocessing phase is to split the data into four separate sets:

- X Train set: Is a set of the features that will be used during the training of the model.

[1]Overfitting is a modelling error that occurs when a model is too aligned to a limited set of data points.

- X Test set: Is a set of the features that will be used when testing the model after training.
- Y Train set: The labels of the train set.
- Y Test set: The labels of the test set.

When splitting the data, you must specify the split ratio. For my implementation, I went with a 70/30 ratio. That corresponds to 70% of the data being placed in the train set and the remaining 30% into the test set.

```
def split_data():
  X = np.zeros([number_of_samples, final_number_of_columns-1],
   dtype=float)
  Y = np.zeros([number_of_samples,], dtype=int)

  for i in range(number_of_samples):
      for j in range(final_number_of_columns-1):
          X[i,j] = data['x'+ str(j)].iloc[i]

  for i in range(number_of_samples):
      Y[i] = data['Label'].iloc[i]

  oversample = SMOTE()
  X, Y = oversample.fit_resample(X, Y)

  X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
   test_size=0.3, random_state=42)

  X_train = np.transpose(X_train)
  X_test = np.transpose(X_test)

  return X_train, X_test, Y_train, Y_test
```

Listing 7.1: Function for splitting and oversampling the data

Listing 7.1 above shows the code I used to split my data. A further explanation of the code is as follows:

- In lines 2 and 3, I create a two-dimensional NumPy[1] array and a one-dimensional NumPy array that are populated with zeros at the start. The X two dimensional array refers to the dataset's features, and the Y array to the labels of the dataset. X is set to contain a number of rows equal to the number of samples in the dataset and a number of columns equal to the number of columns in the dataset minus one, as I stored the label separately. Y is then set to contain a number of rows equal to the number of samples in the dataset.

- In lines 5 to 7, the for loop begins populating the X array with the features. The outer for loop focuses on looping through each row, and the inner for loop focuses on looping through each column. The data is then placed into the appropriate cell using the indexes 'i' and 'j' from the for loops.

- Lines 9 and 10 then populate the Y array with the labels.

- Lines 12 and 13 oversamples the data using SMOTE as shown in table 7.2

- Line 15 splits the X array into X_train and X_test and the Y array into Y_train and Y_test with the specified ratio of 70/30.

- lines 17 and 18 get the transpose of the train arrays. Getting the transpose of a two-dimensional array flips the array over its diagonal so that the rows become the columns and the columns become the rows, thus changing the shape of our arrays. I had to do this because of how NumPy arrays axis works. When creating a one-dimensional array such as our labels, NumPy uses one axis called axis-0, which runs horizontally like, for example, the x-axis. However, when creating a two-dimensional array, this axis-0 changes, so instead of horizontally, it runs vertically like the y-axis, and a new axis is introduced, axis-1, which runs horizontally. Therefore, initially, my features arrays shapes were not corresponding to the shapes of the labels arrays before the features arrays were transposed.

- Finally, in line 20, I return the split data.

[1]NumPy is a Python library which adds support for arrays and matrices.

7.3 Grammar Produced

The grammar I used for this project is a classification grammar that uses various mathematical functions to calculate a prediction using the features in the dataset. It contains only two non-terminal symbols, ⟨e⟩ and ⟨c⟩.

Non-terminal	Production Rules and Terminals
⟨e⟩	add(⟨e⟩,⟨e⟩)|sub(⟨e⟩,⟨e⟩)|pdiv(⟨e⟩,⟨e⟩)|mul(⟨e⟩,⟨e⟩)|neg(⟨e⟩)|.. This line is repeated 21 times x[0] |x[1] |x[2] |x[3] |x[4] |x[5] |.. Continues to list all features up to x[104] ⟨c⟩⟨c⟩.⟨c⟩⟨c⟩|.. This line is repeated 104 times
⟨c⟩	0 |1 |2 |3 |4 |5 |6 |7 |8 |9

Table 7.3: Breakdown of grammar production rules.

The mathematical functions used are: "add", which returns the addition of values. "sub", which returns the subtraction of two values. "pdiv", which returns the division of two values while also protecting against errors which would cause the system to stop executing such as a division by zero error. "mul" which returns the multiplication of two numbers. And "neg", which returns the negative of the value. All these functions were made available by the GRAPE library.

As mentioned in table 7.3 each function is repeated 21 times as a production rule for the non-terminal ⟨e⟩. This repetition is because the dataset contains many features; hence, I had to balance the probability of a function being chosen as the production rule with the probability of a feature being chosen.

The grammar also incorporates decimal numbers to be used along with the features. ⟨c⟩⟨c⟩.⟨c⟩⟨c⟩ allows for the non-terminal ⟨c⟩ to be broken down to a single decimal as shown in table 7.3. This production rule is repeated 104 times also to balance out the probability it is chosen.

```
from ponyge2_adapted_files import Grammar

BNF_GRAMMAR = Grammar("../GE_GRAPE_Mammogram_Classification/
    grammar.bnf")
```

Listing 7.2: Loading in the grammar file using GRAPE's Grammar class

In listing 7.2 the variable BNF_GRAMMAR, which is used later in the implementation, is set to an object of the GRAPE Grammar class, which accepts the grammar file as a parameter. This class acts as a parser for the grammar that has been passed to the constructor.

The benefit of having a recursive aspect to the grammar (having the non-terminal ⟨e⟩ call itself in its production rules e.g., add(⟨e⟩,⟨e⟩)) is that we can have very long individuals that contain many decimals, mathematical functions, and features, as well as short ones. This recursion can allow for a higher degree of exploration and better results. The effects of recursion can be seen in the experiments in chapter 9.

7.4 Fitness Function

The fitness measure I decided to use in my fitness function was accuracy. I chose accuracy as it incorporates both true positives and true negatives, and it is a clear and straightforward measure to communicate results.

```
def fitness_eval(individual, points):
  x= points[0]
  Y= points[1]

  if individual.invalid==True:
    return np.NaN,

  try:
    pred= eval(individual.phenotype)
  except (FloatingPointError, ZeroDivisionError, OverflowError,
   MemoryError):
    return np.NaN,
  assert np.isrealobj(pred)
```

```
13
14  try:
15    Y_class= [1 if pred[i] > 0 else 0 for i in range(len(Y))]
16  except (IndexError, TypeError):
17    return np.NaN,
18
19  # TP -> True Positive, FP -> False Posititve, TN -> True
     Negative, FN -> False Negtaive
20  # 0 -> negative, 1 -> positive
21
22  TP, FN, TN, FP = 0, 0, 0, 0
23
24  for x, y in zip(Y, Y_class):
25    if x == 1:
26      if y == 1:
27        TP = TP + 1
28      else:
29        FN = FN + 1
30    elif x == 0:
31      if y == 0:
32        TN = TN + 1
33      else:
34        FP = FP + 1
35
36  if FN == 0 and TN == 0:
37    return np.NaN,
38  elif FP == 0 and TP == 0:
39    return np.NaN,
40  else:
41    # Calculating Accuracy
42    numerator = (TP+TN)
43    denominator = (TP+TN+FP+FN)
44    fitness = numerator / denominator
45
46  return fitness,
```

Listing 7.3: Fitness Function

The code for my fitness function can be seen in listing 7.3; it accepts two parameters: the individual that is going to be evaluated and the data points, which are either X_train and Y_train or X_test and Y_test. A further breakdown

of the function's code is as follows:

- Lines 2 and 3 separate the features and the labels which are stored in the parameter "points" and store them in X and Y, respectively.

- Lines 5 and 6 checks if the individual is invalid, which is a variable set by the mapper inside the GRAPE code. If it is invalid, the fitness function returns np.NaN, which stands for Not a Number. This way, the individual is ignored and does not contribute to fitness scores. If it is valid, we continue.

- Lines 8 to 11 is where the predictions are calculated. Line 9 uses Pythons eval()[1] function to evaluate our individual and, in turn, generate predictions. The individual will be made up of the grammar shown in table 7.3, so eval will execute the mathematical functions and use the x array in line 2 for the terminals that refer to the features such as 'x[0]'. The 'pred' variable is enclosed in a try-except block so that errors such as a zero division can be caught, and np.NaN is returned as the fitness to avoid Python crashing.

- Line 12 uses the assert[2] keyword and np.isrealobj()[3] to ensure the predictions in pred are all real values.

- Lines 14 to 17 assign each prediction to a class using a boundary of 0. If the prediction is greater than zero, I place a 1 in the Y_class array. Else if it is less than or equal to zero, I place a 0 in the array.

- Lines 22 to 34 are concerned with calculating the Confusion Matrix for the individual. The for loop loops through both the true labels (Y) and the predicted labels (Y_class), taking the values two at a time (one from each array). If the true label is positive, i.e. equal to 1, we check the predicted label and if it is also positive we increment the true positive variable (TP); else, if the predicted label is negative (equal to 0), we increment the false

[1]eval() is used to evaluate arbitrary Python expressions from a string-based or compiled-code-based input.

[2]The assert keyword lets you test if a condition in your code returns True, if not, the program will raise an AssertionError.

[3]np.isrealobj(x) returns True if x is a not complex type or an array of complex numbers.

negative variable (FN). The true negatives (TN) and false positives (FP) are also incremented similarly.

- Lines 36 to 39 check to see if the individual has classed everything as positive or everything as negative. If it has, we return np.NaN so that individual does not do well and does not contribute to the next generation.
- Lines 42 to 46 implement the Accuracy formula and store the result in the fitness variable, which is then returned as the individuals' fitness score at the end of the function.

7.5 Grammatical Evolution

Once the data is prepared, grammar defined, and fitness function implemented, the next step is to set up the GE environment. This involves tuning the hyperparameters, initialising the DEAP toolbox, and finally running GE.

7.5.1 Hyperparameters

GE has 11 hyperparameters that need to be set before running GE. Tuning these hyperparameters to best suit your problem is essential to the GE process and getting better results.

The values I used for the hyperparameters are shown in table 7.4. I arrived at these hyperparameters through some experimentation *(discussed in chapter 9)* in which I found these were the best performing in terms of fitness scores. A further breakdown of the information in table 7.4 is as follows:

Hyperparameter	Value
POPULATION_SIZE	1000
MAX_GENERATIONS	500
P_CROSSOVER	0.95
P_MUTATION	0.01
ELITE_SIZE	1
HALL_OF_FAME_SIZE	1
MAX_INIT_TREE_DEPTH	16
MIN_INIT_TREE_DEPTH	8
MAX_TREE_DEPTH	24
MAX_WRAPS	0
CODON_SIZE	320

Table 7.4: Hyperparameters and their values.

- POPULATION_SIZE: The number of individuals each generation should have. In this case, it is 1000 individuals in each generation.
- MAX_GENERATIONS: The number of generations GE should run for before terminating. In this case, GE will run for 500 generations of 1000 individuals each. Hence GE will be generating approximately 500,000 individuals.
- P_CROSSOVER: is the probability of crossover, which determines whether or not crossover will be performed with an individual. In this case, that probability is relatively high and set to 0.95. A further explanation of why I chose to go with a high rate can be found in section 10.1.3.
- P_MUTATION: is the probability of mutation, which determines whether mutation will or will not be applied to an individual. In this case, that probability is set to 0.01, which is relatively standard for mutation to be low so that we do not keep mutating and, in turn, ruin fit individuals.
- ELITE_SIZE: determines the number of high performing individuals (as in having high fitness scores) that will be copied, as they are, to the next

generation. In this case, that is set to 1, so only 1 individual is copied to the next generation. This size is relatively small, and a further explanation of why I chose to go with 1 can be found in section 10.1.3.

- HALL_OF_FAME_SIZE: this is a DEAP parameter more than a GE one. It determines the number of individuals stored in Hall of Fame. Through each generation, Hall of Fame will store best performing individuals. In this case, it is set to 1 as I only want to know about the overall best individual of all 500 generations.

- MAX_INIT_TREE_DEPTH: limits the size of the syntax tree generated in initialisation to represent an individual so that its depth cannot be bigger than specified. In this case, it is set to 16, which is a relatively big tree.

- MIN_INIT_TREE_DEPTH: specifies the minimum acceptable size of syntax tree to represent an individual. As we can see, it is set to 8, so each individual at initialisation will be in the depth range 8 to 16.

- MAX_TREE_DEPTH: limits the size of the syntax tree throughout the GE process, not just initialisation. This parameter is set to 24, so no tree should be deeper than 24. The reason a limit has to be set is to control run times, as discussed in 10.1.4.

- MAX_WRAPS: If GE reaches the end of a genome and still has non-terminals in the individual, it goes back to the start of the individual and continues mapping as many times as specified by this parameter. Here it is set to 0, so GE will discard the individual in that case.

- CODON_SIZE: A codon is the 8-bit long segment extracted from the genome. This parameter determines how many codons are in each individual. Here it is set to 320, further explanation on why I chose this number can be found in section 10.1.1.

7.5.2 DEAP toolbox

The next part of the setup is to initialise and register parameters with the DEAP toolbox and creator. So the toolbox contains our evolutionary operators, and the creator helps us create classes and types for our EA. The code for setting up toolbox and creator can be seen in listing 7.4.

```
1  toolbox = base.Toolbox()
2
3  creator.create("FitnessMax", base.Fitness, weights=(1.0,))
4
5  creator.create('Individual', ge.Individual, fitness=creator.
       FitnessMax)
6
7  toolbox.register("populationCreator", ge.initialisation_PI_Grow,
       creator.Individual)
8
9  toolbox.register("evaluate", fitness_eval)
10
11 toolbox.register("select", ge.selTournament, tournsize=5)
12
13 toolbox.register("mate", ge.crossover_onepoint)
14
15 toolbox.register("mutate", ge.mutation_int_flip_per_codon)
```

Listing 7.4: DEAP toolbox initialisation

Breakdown of code:

- In line 2, the Creator module is used to create a "FitnessMax" class with Creators abstract Fitness class as its base. Creators Fitness class accepts a tuple of weights; a "FitnessMax" fitness has positive weights and works to maximise the fitness value. A "FitnessMin" fitness would use negative weights and work to minimise the fitness value. As I used accuracy as my fitness function, the appropriate choice is "FitnessMax" to maximise the accuracy.

- In line 3, an individual class is created using GRAPE's individual class as its base class. I also set the individuals fitness class to the one we created in line 2.

- In line 7, I register the initialisation method I would like to use with the toolbox. I chose Position Independent Grow 3.3 as it results in a better balance of trees, eliminating a left-handed or right-handed bias.
- In line 9, I register the fitness function that I created in 7.4 so that it will be used as the evaluation measure.
- In line 11, I register the selection method I would like to be used, which is GRAPEs tournament class 'selTournament' and set the tournament size to 5. Tournament selection works by randomly selecting individuals up to the size specified and choosing the fittest of them for crossover.
- In line 13, I register the crossover method I would like to be used. I chose one-point crossover, which is previously discussed in section 3.3.
- In line 15, I register the mutation method I would like to be used. I chose int flip per codon in which, according to the mutation probability, each codon is replaced independently by choosing a random integer from a range of possible codon values.

7.5.3 Main function

Once the toolbox was set up, I moved on to creating my initial population and starting the GE process.

```
1  population = toolbox.populationCreator(
2                       size=POPULATION_SIZE,
3                       bnf_grammar=BNF_GRAMMAR,
4                       min_init_tree_depth=MIN_INIT_TREE_DEPTH,
5                       max_init_tree_depth=MAX_INIT_TREE_DEPTH,
6                       max_tree_depth=MAX_TREE_DEPTH,
7                       max_wraps=MAX_WRAPS,
8                       codon_size=CODON_SIZE
9                       )
10
11 hof = tools.HallOfFame(HALL_OF_FAME_SIZE)
12
13 stats = tools.Statistics(key=lambda ind: ind.fitness.values)
14 stats.register("avg", np.nanmean)
```

```
15 stats.register("std", np.nanstd)
16 stats.register("min", np.nanmin)
17 stats.register("max", np.nanmax)
18
19 population, logbook = algorithms.ge_eaSimpleWithElitism(
20                         population,
21                         toolbox,
22                         cxpb=P_CROSSOVER,
23                         mutpb=P_MUTATION,
24                         ngen=MAX_GENERATIONS,
25                         elite_size=ELITE_SIZE,
26                         bnf_grammar=BNF_GRAMMAR,
27                         codon_size=CODON_SIZE,
28                         max_tree_depth=MAX_TREE_DEPTH,
29                         max_wraps=MAX_WRAPS,
30                         points_train=[X_train, Y_train],
31                         points_test=[X_test, Y_test],
32                         stats=stats,
33                         halloffame=hof,
34                         verbose=True
35                         )
```

Listing 7.5: Main function

The code in listing 7.5 can be divided in to three parts:

1. Initialising the population.
2. Setting up statistics to gather information on the generations.
3. Start the evolution process and production of each generation.

Initialising the population

Lines 1 to 9 of listing 7.5 handle the initialisation of the population using DEAPs toolbox populationCreator. populationCreator accepts various parameters that I have set, such as the POPULATION_SIZE, MIN_INIT_TREE_DEPTH, BNF_GRAMMAR etc. Using those parameters, it will create 1000 individuals and store them in the population variable.

Statistics

Lines 11 to 17 of listing 7.5 focus on setting up the statistics that I want to be kept track of. First Hall of Fame5.4 is initialised using the size I set early in the hyperparameters 7.5.1. Then I move on to the Statistics5.4, initialising the statistics tool and registering the stats I would like to keep track of, which are:

- "avg" is the average training fitness of all individuals in the generation.
- "std" is the standard deviation of the training fitnesses of the individuals, i.e. how dispersed they are from the mean.
- "min" will keep track of the lowest fitness in the generation.
- "max" will keep track of the highest fitness in the generation.

Evolution

Finally, lines 19 to 35 of listing 7.5 is where GE starts evolving from the initial generation up to the MAX_GENERATIONS limit I set. There were a few EAs made available by GRAPE to choose from; I chose to use ge_eaSimpleWithElitism(), which reproduces the most straightforward evolutionary algorithm and includes elitism[1]. The algorithm accepts many parameters, among them the hyperparameters P_CROSSOVER, ELITE_SIZE, etc. and also the data which I earlier split into test and train sets. The algorithms process goes as follows:

1. It evaluates the invalid fitness and keeps track of the number of invalid individuals.
2. It enters the generational loop. Here the selection method gets applied to replace the previous population. The replacement ratio used by this algorithm is a 1:1 ratio. Hence we need to use a selection procedure that is stochastic and will select the same individual multiple times. This requirement is met with my choice of tournament selection.

[1]Elitism is a selection method which always copies the best n individuals to the next generation where n is set by the user.

3. It produces the next generation's population with a mix of crossover and Elitism.

4. It evaluates the new individuals and calculates the statistics I specified earlier.

5. Lastly, once MAX_GENERATIONS is reached, it returns a tuple with the final population stored in the population variable, and a summary of the results and stats stored in the logbook variable.

```
max_fitness_values, mean_fitness_values = logbook.select("max", "
    avg")
min_fitness_values, std_fitness_values = logbook.select("min", "
    std")
fitness_test = logbook.select("fitness_test")
best_ind_length = logbook.select("best_ind_length")
avg_length = logbook.select("avg_length")
max_length = logbook.select("max_length")
selection_time = logbook.select("selection_time")
generation_time = logbook.select("generation_time")
gen, invalid = logbook.select("gen", "invalid")
```

Listing 7.6: Storing Statistics

```
max_fitness_values, mean_fitness_values = logbook.select("max", "
    avg")
min_fitness_values, std_fitness_values = logbook.select("min", "
    std")
fitness_test = logbook.select("fitness_test")
best_ind_length = logbook.select("best_ind_length")
avg_length = logbook.select("avg_length")
max_length = logbook.select("max_length")
selection_time = logbook.select("selection_time")
generation_time = logbook.select("generation_time")
gen, invalid = logbook.select("gen", "invalid")
```

Listing 7.7: Extracting Best Individual from Hall of Fame

Once the GE process is complete, I extract some statistics from the logbook, as can be seen in listing 7.6, and I extract the best individual from all the generations which has been stored by Hall of Fame, which can be seen in listing 7.7. Using

this information, I determine and visualise my results which are discussed in the next chapter 8.

Chapter 8

Results and Results Confidence

In this chapter the results obtained from the implementation in Chapter 7 are presented along with 30 re-runs of the implementation to demonstrate result confidence.

8.1 Results

8.1.1 Best Individual

The best individual the implementation in Chapter 7 produced is as follows:

```
1    sub(mul(mul(sub(add(x[85],x[70]),93.63),mul(x[34],add(mul(x
   [74],x[4]),x[65]))),sub(neg(add(add(pdiv(mul(88.40,66.32)
   ,98.40),mul(x[85],sub(mul(x[0],mul(mul(43.93,x[98]),mul(00.35,
   add(mul(pdiv(83.36,mul(x[34],08.46)),add(x[85],x[70])),93.63))
   )),add(mul(mul(90.30,sub(mul(83.08,mul(x[65],38.59)),x[64])),x
   [59]),x[70])))),43.33)),x[85])),x[70])
```

Listing 8.1: Best Individual produced - shown in Prefix Notation

As we can see from the listing 8.1, the best individual is long and complex, consisting of many different mathematical functions, features, and decimals.

Below in figure 8.1 is a syntax tree representation that I drew of the best individual in listing 8.1. We can see that there is a left-handed bias in the tree, and it is not balanced as the trees of the initial population would have been.

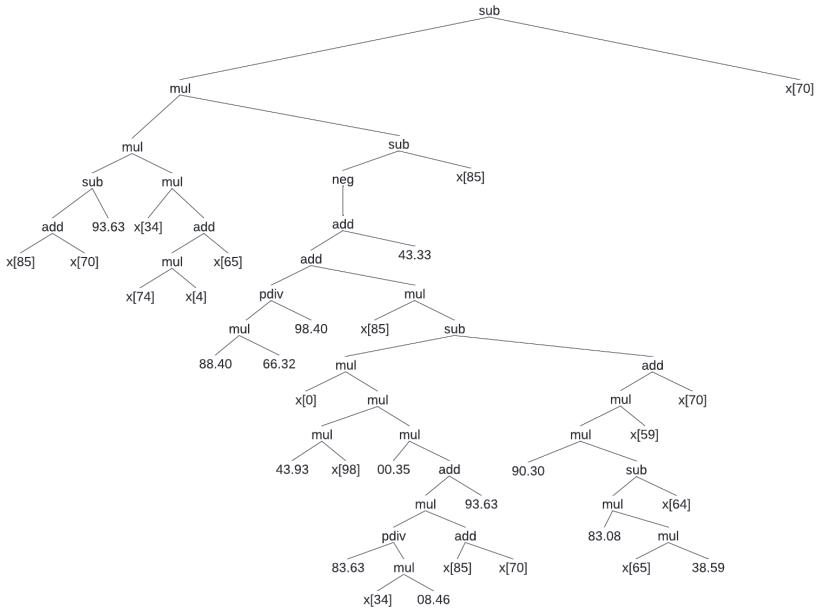

Figure 8.1: Best individual represented as a syntax tree.

From table 7.4 we can see some facts about the individual. The training fitness of the individual is 66.6%, meaning that it is 66.6% likely to classify a mammogram segment correctly; this is the individuals' performance on the train data. The remaining three rows are concerned with the size of the individual and how much of it was used. We can see that only 9% of the genome was used to produce this phenotype through mapping.

Metric	Value
Training Fitness	0.6661412639405204 (accuracy)
Depth	18
Length of the genome	1221
Used portion of the genome	0.09

Table 8.1: Best Individual metrics.

8.1.2 Testing Best Individual

The fitness measure used to evaluate the individuals through the evolutionary process has been Accuracy. However, I wanted to find out some more about the performance of the best individual. Therefore, using the test set and the best individual, I calculated the individuals' Confusion Matrix, test Accuracy, True Positive Rate, True Negative Rate, False Positive Rate, False Negative Rate, and AUC.

/	Actual Positive	Actual Negative
Predicted Positive	5722	3370
Predicted Negative	1483	3836

Table 8.2: Best Individual Confusion Matrix.

Metric	Score
Test Accuracy	66.3%
True Positive Rate	79.4%
True Negative Rate	53.2%
False Positive Rate	46.8%
False Negative Rate	20.6%
AUC	0.66

Table 8.3: Best Individual Fitness metrics scores.

8.1.3 Plots of Generations

To visualise the evolution process, I created three plots:

1. The best training fitness and test fitness throughout the generations.
2. The average fitness across the generations.
3. The max, average, and best individual genome length across the generations.

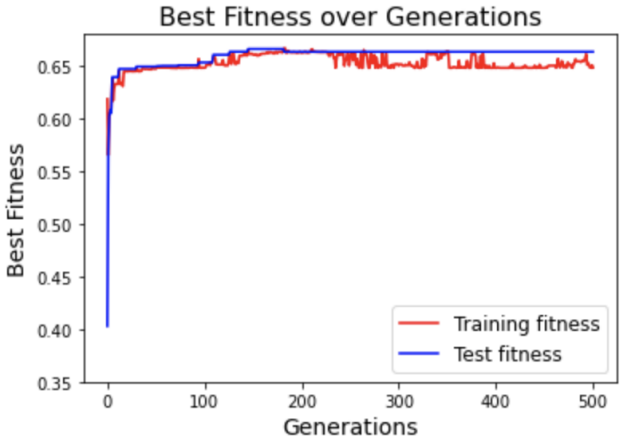

Figure 8.2: Best training and test fitness over 500 generations.

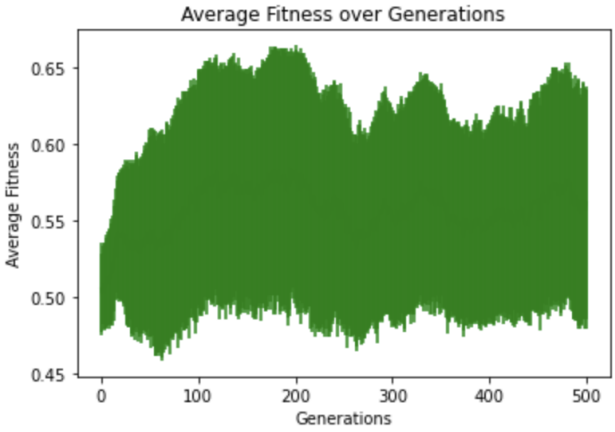

Figure 8.3: Average fitness over 500 generations.

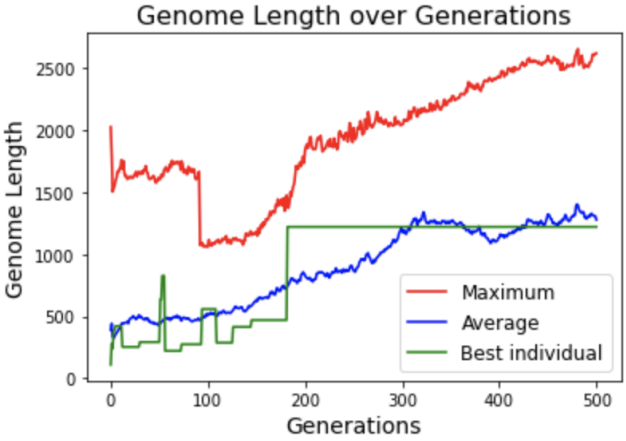

Figure 8.4: Max, average, and best individual length over 500 generations.

As we can see in figure 8.2, the test fitness had a good amount of growth at the start but then started to level off near the end from about 200 generations. The train fitness varies throughout, dropping and rising in approximately 0.03 intervals which can be an indicator that GE is doing a high degree of exploration. I was glad to see the train and test fitness remain close, showing that no over-fitting or under-fitting occurred.

In figure 8.3 we can see the average fitness of the generations. We can see that it is growing at the start and then begins to level out and drop a little. This indicates that GE is doing more exploration of new individuals than exploiting existing strong individuals. Crossover rate and elite size need to be tweaked to allow both exploration and exploitation and not favour one over the other.

In figure 8.4 we can see the max, average and best individual genome lengths of each generation. We can see that as the generations go on, the genomes do tend to get longer, but the best individual genome levels out from approximately generation 200 like the test fitness. This can mean that the elite individual (as ELITE_SIZE is set to 1) is being copied into each generation from 200 on, and no better individual is being produced.

8.2 Results Confidence

In order to establish confidence in the results I achieved above, I executed the GE process repeatedly for 30 runs and kept track of the results. I re-split the data for each run of the 30 before running GE.

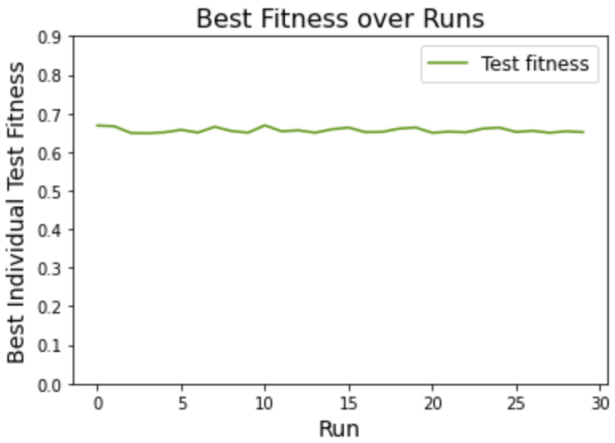

Figure 8.5: Best Individual test fitness over 30 runs of 500 generations.

As we can see from the figure 8.5 above, the test fitness of each best individual produced by the 30 runs does vary slightly throughout. This can be attributed to the stochastic[1] nature of EAs. Table 8.4 below gives a more detailed breakdown of the scores of the best individuals across various fitness metrics. We can see that the Accuracy ranges from a low of 65% to a high of 67% achieved on the test set.

[1]Stochastic means having a random pattern that, even though it can be analysed statistically, it cannot be predicted precisely.

Run	Accuracy	TPR	TNR	FPR	FNR	AUC
1	66.8%	81.1%	52.6%	47.4%	18.9%	0.67
2	66.7%	82.4%	51%	49.1%	17.6%	0.67
3	65%	88%	42%	58.1%	12%	0.65
4	65%	87.4%	42.3%	58%	12.6%	0.65
5	65.1%	86%	44.4%	56%	14.2%	0.65
6	66%	81%	51%	49.3%	19.2%	0.66
7	65%	88.4%	42%	58.3%	12%	0.65
8	67%	83.4%	50%	50.4%	17%	0.67
9	65.4%	83%	48.2%	52%	17.4%	0.65
10	65%	88.2%	42%	58.2%	12%	0.65
11	67%	84%	50%	50%	16.2%	0.67
12	65.4%	82.1%	49%	51.4%	18%	0.65
13	66%	83%	48.2%	52%	17%	0.66
14	65%	88.2%	42%	58.2%	12%	0.65
15	66%	84.1%	48%	52.3%	16%	0.66
16	66.3%	78.4%	54.2%	46%	22%	0.66
17	65.1%	88.1%	42.1%	58%	12%	0.65
18	65.2%	87%	44%	56.5%	13.2%	0.65
19	66%	83.4%	49%	51.4%	17%	0.66
20	66.3%	87%	46%	54.3%	13%	0.66
21	65%	86%	44%	56%	14.1%	0.65
22	65.3%	76.1%	54.4%	46%	24%	0.65
23	65.1%	88.4%	42%	58.2%	12%	0.65
24	66%	84%	48.2%	52%	16.2%	0.66
25	66.3%	82.2%	50.4%	50%	17.8%	0.66
26	65.2%	87%	44%	56.2%	13.4%	0.65
27	65.5%	88.3%	43%	57.4%	12%	0.65
28	65%	86%	44.1%	56%	14.1%	0.65
29	65.3%	90%	41.1%	59%	10.5%	0.65
30	65.1%	89%	41.2%	59%	11%	0.65

Table 8.4: Fitness metrics of Best Individual from each run.

Chapter 9

Experiments and Results

To improve my results, I conducted experiments in which I tweaked the hyperparameters and grammar I used until I arrived at the implementation and results discussed in 7 and 8 respectively. Through the following experiments, I increased my accuracy from 54.9% to 66.6%. The first three experiments use the initial grammar I produced, while the remaining use the final grammar discussed in 7.3.

The initial grammar, the set up, and the results of each experiment are presented and discussed in the following chapter.

9.1 Initial Grammar

Initially, I had produced a different grammar to that discussed in 7.3. What differentiates this grammar is that it used five non-terminals and separated the dataset's features so that one non-terminal (segment_a) referenced features of the breast segment in question and another non-terminal (segment_b) referenced the features of the same breast segment on the opposite breast.

As we can see from table 9.1 below, the mathematical functions are not repeated as many times as they were in 7.3. Instead, there are two of each, with one occurrence accepting segment_a as the first argument and segment_b as the second and vice versa for the second occurrence. The decimal non-terminal is also repeated far fewer times with just one occurrence.

Overall, on reflection, I believe this grammar was too restricting to allow GE to create varying length individuals. It only allows GE to compare two features at a time, but as we can see from 8.1.1, GE used 18 different features in the best individual, making just two features far too low.

Non-terminal	Production Rules and Terminals
⟨e⟩	add(⟨segment_a⟩,⟨segment_b⟩)| sub(⟨segment_a⟩,⟨segment_b⟩)| pdiv(⟨segment_a⟩,⟨segment_b⟩)| mul(⟨segment_a⟩,⟨segment_b⟩)| ⟨decimal⟩ | ⟨segment_a⟩ | add(⟨segment_b⟩,⟨segment_a⟩)| sub(⟨segment_b⟩,⟨segment_a⟩)| pdiv(⟨segment_b⟩,⟨segment_a⟩)| mul(⟨segment_b⟩,⟨segment_a⟩)| ⟨segment_b⟩
⟨decimal⟩	⟨decimal⟩⟨decimal⟩.⟨decimal⟩⟨decimal⟩
⟨segment_a⟩	x[0] .. x[51]
⟨segment_b⟩	x[52] .. x[103]
⟨n⟩	0 |1 |2 |3 |4 |5 |6 |7 |8 |9

Table 9.1: Breakdown of grammar production rules.

9.2 Experiment one

9.2.1 Set Up

The grammar is that described in 9.1 and fitness function is Accuracy as described in 7.4.

Hyperparameter	Value
POPULATION_SIZE	200
MAX_GENERATIONS	100
P_CROSSOVER	0.8
P_MUTATION	0.2
ELITE_SIZE	round(0.1*POPULATION_SIZE)
HALL_OF_FAME_SIZE	10
MAX_INIT_TREE_DEPTH	8
MIN_INIT_TREE_DEPTH	2
MAX_TREE_DEPTH	10
MAX_WRAPS	0
CODON_SIZE	255
TOURN_SIZE	10

Table 9.2: Experiment 1 Hyperparameters and their values.

9.2.2 Results

The best individual produced in this experiment is as follows:

```
1     sub(x[0],x[52])
```

Listing 9.1: Best Individual produced in experiment 1

This individual produced the following results:

Metric	Score
Train Accuracy	54.9%
Test Accuracy	53.6%
True Positive Rate	57.9%
True Negative Rate	49.3%
False Positive Rate	50.6%
False Negative Rate	42%
AUC	0.53

Table 9.3: Best Individual from experiment 1 fitness metrics scores.

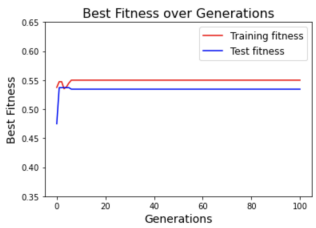

Figure 9.1: Best training and test fitness of experiment 1.

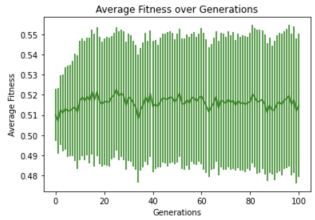

Figure 9.2: Average fitness of experiment 1.

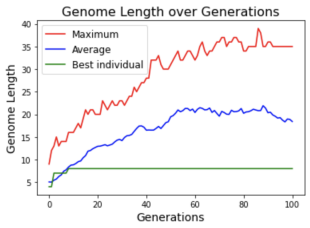

Figure 9.3: Genome lengths for experiment 1.

9.2.3 Conclusion

At the time, I had two issues with these results:

- The Accuracy was very low, only getting about half of the classifications correct.
- The average, test, and train fitnesses, as can be seen in figures 9.1 and 9.2, level off very early on in the generations and are very stagnant.

As far as I could tell, GE was doing too much exploitation, settling on an individual it deemed good enough early on and not exploring new solutions. Therefore,

I knew I would have to keep tweaking my hyperparameters to achieve better results.

9.3 Experiment two

9.3.1 Set Up

The grammar is that described in 9.1 and fitness function is Accuracy as described in 7.4.

Hyperparameter	Value
POPULATION_SIZE	200
MAX_GENERATIONS	100
P_CROSSOVER	0.8
P_MUTATION	0.02
ELITE_SIZE	round(0.01*POPULATION_SIZE)
HALL_OF_FAME_SIZE	10
MAX_INIT_TREE_DEPTH	11
MIN_INIT_TREE_DEPTH	6
MAX_TREE_DEPTH	17
MAX_WRAPS	0
CODON_SIZE	255
TOURN_SIZE	10

Table 9.4: Experiment 2 Hyperparameters and their values.

9.3.2 Results

The best individual produced in this experiment is as follows:

```
1     sub(x[0],x[52])
```

Listing 9.2: Best Individual produced in experiment 2

This individual produced the following results:

Metric	Score
Train Accuracy	55%
Test Accuracy	53.9%
True Positive Rate	58.5%
True Negative Rate	49.3%
False Positive Rate	50.6%
False Negative Rate	41.4%
AUC	0.54

Table 9.5: Best Individual from experiment 2 fitness metrics scores.

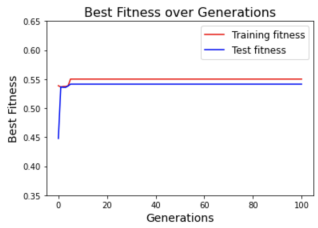

Figure 9.4: Best training and test fitness of experiment 2.

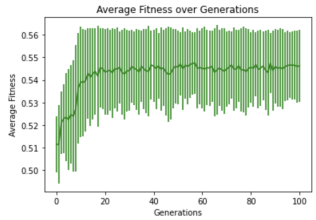

Figure 9.5: Average fitness of experiment 2.

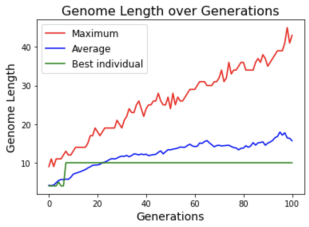

Figure 9.6: Genome lengths for experiment 2.

9.3.3 Conclusion

In this experiment, I focused on tweaking the mutation rate and elite size. I thought that if I increased the number of strong individuals being copied over to the next generation and protected them against mutation, that would increase the best individual fitness. However, we can see the effect that the increase in elite size had was a stronger and increasing average fitness compared to experiment one's average fitness, which was weaker and stagnant (see figures 9.2 and 9.5). This is expected as now each generation will contain strong performing individuals from the previous one. The decrease in mutation rate does not seem to have a noticeable effect here; my thinking behind the decrease was to protect strong

individuals who may have been lost in experiment one due to a high mutation rate. I cannot see that that was achieved here, as the increase in the fitness metrics is very slight. Also, with the exact same best individual being produced, this increase is most likely attributed to a new split of the test and train data, with the test data being more favourable for the best individual this time around.

9.4 Experiment three

9.4.1 Set Up

The grammar is that described in 9.1 and fitness function is Accuracy as described in 7.4.

Hyperparameter	Value
POPULATION_SIZE	200
MAX_GENERATIONS	100
P_CROSSOVER	0.6
P_MUTATION	0.2
ELITE_SIZE	round(0.1*POPULATION_SIZE)
HALL_OF_FAME_SIZE	10
MAX_INIT_TREE_DEPTH	8
MIN_INIT_TREE_DEPTH	2
MAX_TREE_DEPTH	10
MAX_WRAPS	0
CODON_SIZE	255
TOURN_SIZE	10

Table 9.6: Experiment 3 Hyperparameters and their values.

9.4.2 Results

The best individual produced in this experiment is as follows:

```
1     sub(x[79],x[1])
```

Listing 9.3: Best Individual produced in experiment 3

This individual produced the following results:

Metric	Score
Train Accuracy	56.1%
Test Accuracy	55.5%
True Positive Rate	64.6%
True Negative Rate	46.5%
False Positive Rate	53.4%
False Negative Rate	35.3%
AUC	0.55

Table 9.7: Best Individual from experiment 3 fitness metrics scores.

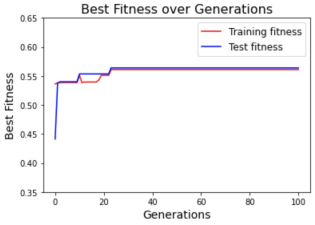

Figure 9.7: Best training and test fitness of experiment 3.

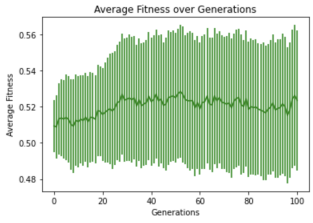

Figure 9.8: Average fitness of experiment 3.

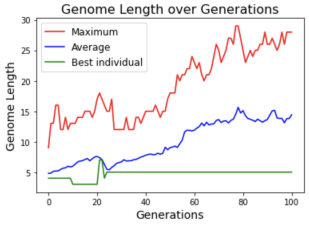

Figure 9.9: Genome lengths for experiment 3.

9.4.3 Conclusion

For this experiment, I only tweaked the crossover rate. Since altering elite size and mutation rate in experiment two did not help GE find a better individual than in experiment one, I left those hyperparameters as they were and focused

on crossover rate. The decrease in crossover has resulted in an increase in fitness metrics, perhaps as this individual was subject to crossover in the previous experiment and lost. At this point, I could tell that the grammar was far too restricting; to see any significant increase in fitness, I had to use a grammar that allowed for a broader range of individuals.

9.5 Experiment four

9.5.1 Set Up

The grammar is that described in 7.3 and fitness function is Accuracy as described in 7.4.

Hyperparameter	Value
POPULATION_SIZE	200
MAX_GENERATIONS	100
P_CROSSOVER	0.9
P_MUTATION	0.6
ELITE_SIZE	round(0.1*POPULATION_SIZE)
HALL_OF_FAME_SIZE	10
MAX_INIT_TREE_DEPTH	8
MIN_INIT_TREE_DEPTH	2
MAX_TREE_DEPTH	10
MAX_WRAPS	0
CODON_SIZE	320
TOURN_SIZE	10

Table 9.8: Experiment 4 Hyperparameters and their values.

9.5.2 Results

The best individual produced in this experiment is as follows:

```
1    add(mul(sub(add(x[101],x[10]),sub(44.15,pdiv(x[57],x[103]))),
   x[47]),neg(32.55))
```

Listing 9.4: Best Individual produced in experiment 4

This individual produced the following results:

Metric	Score
Train Accuracy	60.9%
Test Accuracy	60.9%
True Positive Rate	91%
True Negative Rate	30.7%
False Positive Rate	69.2%
False Negative Rate	8.9%
AUC	0.6

Table 9.9: Best Individual from experiment 4 fitness metrics scores.

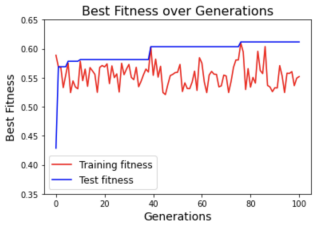

Figure 9.10: Best training and test fitness of experiment 4.

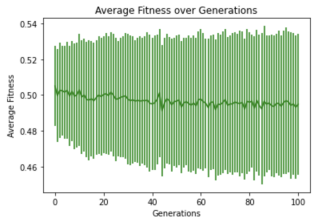

Figure 9.11: Average fitness of experiment 4.

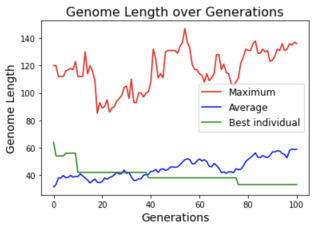

Figure 9.12: Genome lengths for experiment 4.

9.5.3 Conclusion

In this experiment, I increased the crossover and mutation rates in an attempt to allow for a higher degree of exploration. We can see there has been a good increase in the fitness metrics from the previous experiment, $\approx 4\%$. However, I

believe this can be attributed to the larger grammar as we can see the individual is much longer than that of the previous experiments. Looking at the graphs, I was glad to see movement in the test and train fitness in 9.10. However, looking at 9.11, it appears as though the average fitness of the generations was getting worse. I believe this can be attributed to the high mutation rate. Initially, I believed high mutation would allow further exploration, but here it seems to be causing strong individuals to become weaker.

9.6 Experiment five

9.6.1 Set Up

The grammar is that described in 7.3 and fitness function is Accuracy as described in 7.4.

Hyperparameter	Value
POPULATION_SIZE	200
MAX_GENERATIONS	100
P_CROSSOVER	0.9
P_MUTATION	0.5
ELITE_SIZE	round(0.2*POPULATION_SIZE)
HALL_OF_FAME_SIZE	10
MAX_INIT_TREE_DEPTH	8
MIN_INIT_TREE_DEPTH	2
MAX_TREE_DEPTH	10
MAX_WRAPS	0
CODON_SIZE	320
TOURN_SIZE	5

Table 9.10: Experiment 5 Hyperparameters and their values.

9.6.2 Results

The best individual produced in this experiment is as follows:

```
add(neg(add(81.31,add(neg(x[31]),pdiv(20.74,mul(25.22,46.85))
)))),x[11])
```

Listing 9.5: Best Individual produced in experiment 5

This individual produced the following results:

Metric	Score
Train Accuracy	62.2%
Test Accuracy	62.7%
True Positive Rate	76.7%
True Negative Rate	48.6%
False Positive Rate	51.3%
False Negative Rate	23.2%
AUC	0.62

Table 9.11: Best Individual from experiment 5 fitness metrics scores.

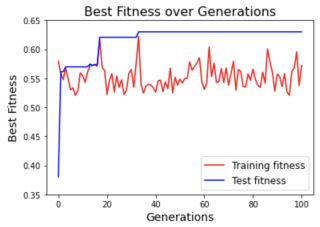

Figure 9.13: Best training and test fitness of experiment 5.

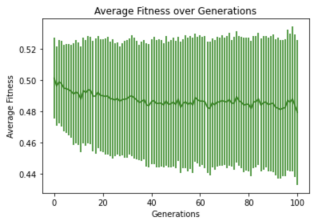

Figure 9.14: Average fitness of experiment 5.

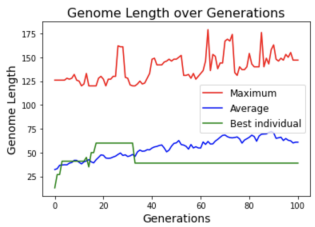

Figure 9.15: Genome lengths for experiment 5.

9.6.3 Conclusion

In an attempt to protect the strong individuals and exploit them, I lowered the mutation rate and increased the elite size in this experiment. We can see that it has caused an increase in the fitness metrics; however, the average fitness is still decreasing, indicating too much exploration is still going on.

9.7 Experiment six

9.7.1 Set Up

The grammar is that described in 7.3 and fitness function is Accuracy as described in 7.4.

Hyperparameter	Value
POPULATION_SIZE	200
MAX_GENERATIONS	100
P_CROSSOVER	0.6
P_MUTATION	0.2
ELITE_SIZE	round(0.01*POPULATION_SIZE)
HALL_OF_FAME_SIZE	1
MAX_INIT_TREE_DEPTH	10
MIN_INIT_TREE_DEPTH	1
MAX_TREE_DEPTH	17
MAX_WRAPS	0
CODON_SIZE	320
TOURN_SIZE	10

Table 9.12: Experiment 6 Hyperparameters and their values.

9.7.2 Results

The best individual produced in this experiment is as follows:

```
1     add(x[63],sub(x[5],68.79))
```

Listing 9.6: Best Individual produced in experiment 6

This individual produced the following results:

Metric	Score
Train Accuracy	63.7%
Test Accuracy	64.5%
True Positive Rate	86.9%
True Negative Rate	42%
False Positive Rate	57.9%
False Negative Rate	13%
AUC	0.64

Table 9.13: Best Individual from experiment 6 fitness metrics scores.

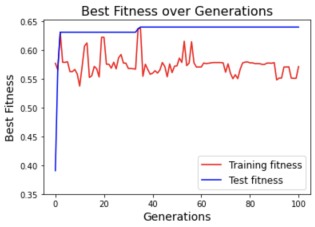

Figure 9.16: Best training and test fitness of experiment 6.

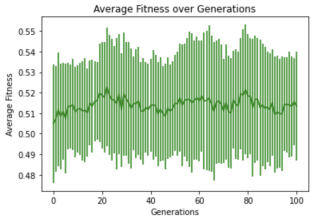

Figure 9.17: Average fitness of experiment 6.

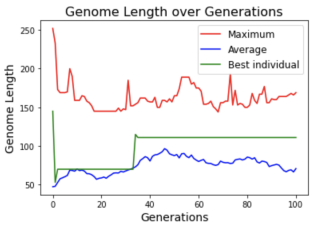

Figure 9.18: Genomes lengths for experiment 6.

9.7.3 Conclusion

I made a big decrease in the mutation rate in this experiment in an attempt to stop losing strong individuals and improve the average fitness of the generations. As we can see this decrease has resulted in a stronger best individual and the average fitness has improved with some growth taking place throughout the generations but the growth still isn't gradual and continuous as I would like it to be.

9.8 Experiment seven

9.8.1 Set Up

The grammar is that described in 7.3 and fitness function is Accuracy as described in 7.4.

Hyperparameter	Value
POPULATION_SIZE	200
MAX_GENERATIONS	100
P_CROSSOVER	0.8
P_MUTATION	0.01
ELITE_SIZE	round(0.01*POPULATION_SIZE)
HALL_OF_FAME_SIZE	1
MAX_INIT_TREE_DEPTH	13
MIN_INIT_TREE_DEPTH	3
MAX_TREE_DEPTH	23
MAX_WRAPS	0
CODON_SIZE	320
TOURN_SIZE	5

Table 9.14: Experiment 7 Hyperparameters and their values.

9.8.2 Results

The best individual produced in this experiment is as follows:

```
1    add(sub(x[53],80.87),add(x[7],pdiv(add(19.48,x[55]),sub(add(x
   [44],x[11]),79.47))))
```

Listing 9.7: Best Individual produced in experiment 7

This individual produced the following results:

Metric	Score
Train Accuracy	64.8%
Test Accuracy	64.3%
True Positive Rate	86.2%
True Negative Rate	42.4%
False Positive Rate	57.5%
False Negative Rate	13.7%
AUC	0.64

Table 9.15: Best Individual from experiment 7 fitness metrics scores.

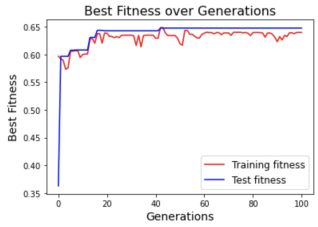

Figure 9.19: Best training and test fitness of experiment 7.

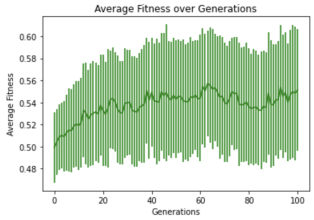

Figure 9.20: Average fitness of experiment 7.

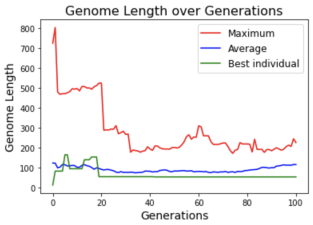

Figure 9.21: Genome lengths for experiment 7.

9.8.3 Conclusion

In this experiment, I decreased the mutation rate as it proved favourable in the previous experiment, I increased the crossover rate in an attempt to improve the exploration of new individuals, and I made the tree sizes bigger to produce larger individuals. We can see from the results that there has been an increase in the fitness metrics, indicating a better individual has been produced. We can also see a better gradual increase in average fitness, indicating that the generations are getting fitter with these hyperparameters. I was interested to see the continuous decrease in the genome lengths (9.21) when I had expected the increase in the max length of a tree to allow for individuals to get gradually larger.

9.9 Experiment eight

9.9.1 Set Up

The grammar is that described in 7.3 and fitness function is Accuracy as described in 7.4.

Hyperparameter	Value
POPULATION_SIZE	200
MAX_GENERATIONS	100
P_CROSSOVER	0.8
P_MUTATION	0.01
ELITE_SIZE	round(0.01*POPULATION_SIZE)
HALL_OF_FAME_SIZE	1
MAX_INIT_TREE_DEPTH	10
MIN_INIT_TREE_DEPTH	1
MAX_TREE_DEPTH	17
MAX_WRAPS	0
CODON_SIZE	320
TOURN_SIZE	5

Table 9.16: Experiment 8 Hyperparameters and their values.

9.9.2 Results

The best individual produced in this experiment is as follows:

```
1     sub(x[33],sub(93.48,x[83]))
```

Listing 9.8: Best Individual produced in experiment 8

This individual produced the following results:

Metric	Score
Train Accuracy	64.7%
Test Accuracy	64.8%
True Positive Rate	86.8%
True Negative Rate	42.9%
False Positive Rate	57%
False Negative Rate	13.1%
AUC	0.64

Table 9.17: Best Individual from experiment 8 fitness metrics scores.

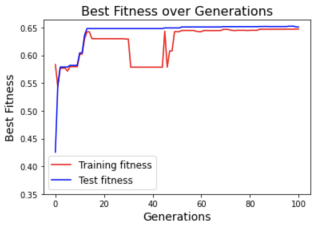

Figure 9.22: Best training and test fitness of experiment 8.

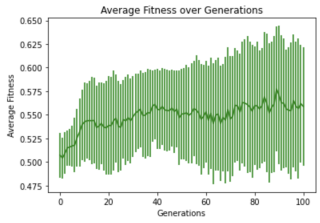

Figure 9.23: Average fitness of experiment 8.

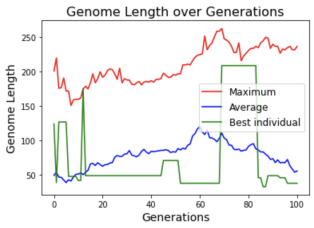

Figure 9.24: Genome lengths for experiment 8.

9.9.3 Conclusion

Given the genome lengths decreasing in the previous experiment, in this experiment, I set the tree sizes back to what they were initially with the same mutation rate, crossover rate, and elite size. As we can see, the best individual in this experiment is much smaller than that of the previous one. However, it received very similar results making the impression that the difference in size between this experiment and the previous one has not made a difference to GE's search.

9.10 Experiment nine

9.10.1 Set Up

The grammar is that described in 7.3 and fitness function is Accuracy as described in 7.4.

Hyperparameter	Value
POPULATION_SIZE	200
MAX_GENERATIONS	100
P_CROSSOVER	0.8
P_MUTATION	0.01
ELITE_SIZE	round(0.01*POPULATION_SIZE)
HALL_OF_FAME_SIZE	1
MAX_INIT_TREE_DEPTH	17
MIN_INIT_TREE_DEPTH	7
MAX_TREE_DEPTH	27
MAX_WRAPS	0
CODON_SIZE	320
TOURN_SIZE	5

Table 9.18: Experiment 9 Hyperparameters and their values.

9.10.2 Results

The best individual produced in this experiment is as follows:

```
1    neg(add(sub(neg(sub(neg(x[60]),sub(63.06,x[70]))),sub(x
   [46],79.57)),sub(pdiv(22.78,sub(x[85],43.59)),50.13)))
```

Listing 9.9: Best Individual produced in experiment 9

This individual produced the following results:

Metric	Score
Train Accuracy	64.2%
Test Accuracy	64.8%
True Positive Rate	84.4%
True Negative Rate	45.2%
False Positive Rate	54.7%
False Negative Rate	15.5%
AUC	0.64

Table 9.19: Best Individual from experiment 9 fitness metrics scores.

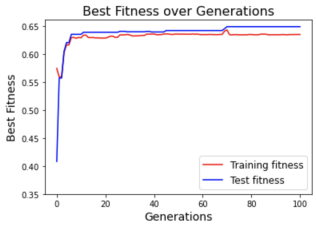

Figure 9.25: Best training and test fitness of experiment 9.

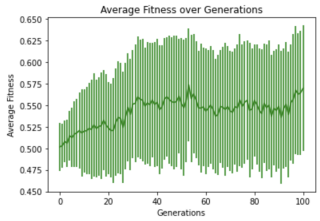

Figure 9.26: Average fitness of experiment 9.

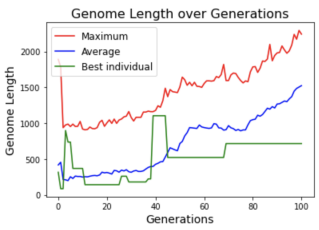

Figure 9.27: Genome lengths for experiment 9.

9.10.3 Conclusion

I still wanted to explore the effect of tree sizes, so in this experiment, I made them much larger than in previous experiments 9.8 and 9.9. Unfortunately, we can see there has not been much of a development regarding the accuracy of the individual. Further indicating tuning tree size alone is not enough to get a stronger individual.

9.11 Experiment ten

9.11.1 Set Up

The grammar is that described in 7.3 and fitness function is Accuracy as described in 7.4.

Hyperparameter	Value
POPULATION_SIZE	500
MAX_GENERATIONS	100
P_CROSSOVER	0.821
P_MUTATION	0.03
ELITE_SIZE	round(0.02*POPULATION_SIZE)
HALL_OF_FAME_SIZE	1
MAX_INIT_TREE_DEPTH	10
MIN_INIT_TREE_DEPTH	3
MAX_TREE_DEPTH	17
MAX_WRAPS	0
CODON_SIZE	392
TOURN_SIZE	8

Table 9.20: Experiment 10 Hyperparameters and their values.

9.11.2 Results

The best individual produced in this experiment is as follows:

```
1    sub(add(x[57],29.15),add(65.19,add(sub(23.56,x[98]),37.11)))
```

Listing 9.10: Best Individual produced in experiment 10

This individual produced the following results:

Metric	Score
Train Accuracy	64.6%
Test Accuracy	65%
True Positive Rate	85.6%
True Negative Rate	44.4%
False Positive Rate	55.5%
False Negative Rate	14.3%
AUC	0.65

Table 9.21: Best Individual from experiment 10 fitness metrics scores.

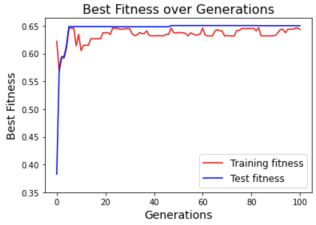

Figure 9.28: Best training and test fitness of experiment 10.

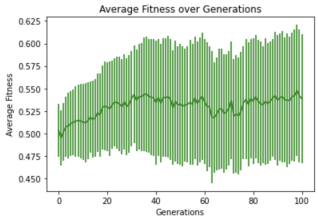

Figure 9.29: Average fitness of experiment 10.

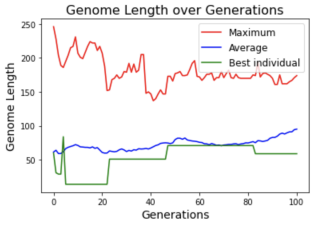

Figure 9.30: Genome lengths for experiment 10.

9.11.3 Conclusion

In this experiment, I decided to use the hyperparameters suggested for classification problems by Wang et al. in their "Hyper-Parameter Optimization for Improving the Performance of Grammatical Evolution"[1] paper which is discussed in section 4.4. The results achieved through those parameters slightly improve those of previous experiments. However, the test fitness, which can be seen in figure 9.28 is very stagnant and does not grow from $\approx$ generation 10. Therefore, I was not happy with these results and continued on to get a bit more of an improvement which resulted in the implementation and results discussed in chapters 7 and 8 respectively.

9.12 Conclusion of all experiments results

To conclude the results of all the experiments I made the following graphs:

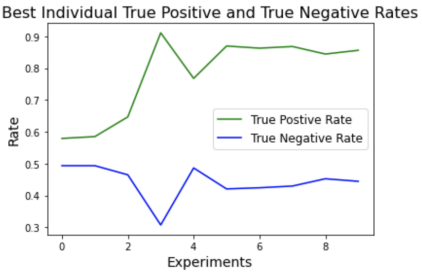

Figure 9.31: TPR and TNR across all experiments.

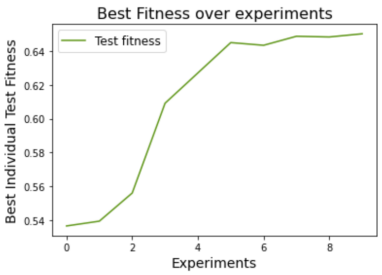

Figure 9.32: Test fitness across all experiments.

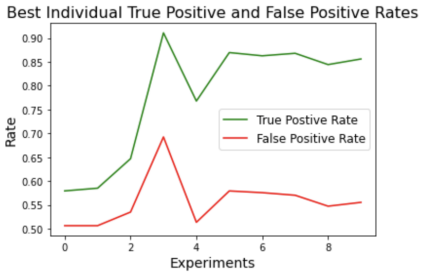

Figure 9.33: TPR and FPR across all experiments.

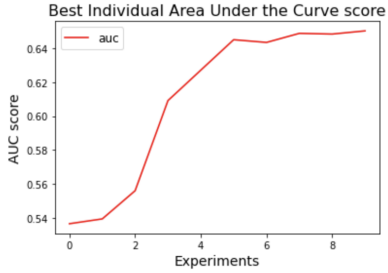

Figure 9.34: AUC score across all experiments.

The graphs in figures 9.32 and 9.34 show the test fitness and AUC scores across the experiments. As we can see, they have had a significant increase from start to finish which is attributed to hyperparameter tuning and updating the grammar.

The graph in figure 9.31 shows us the true positive rate and true negative rate achieved by the best individual of each experiment. The ideal to see here would be to have the two plots very close to each other to indicate we are classifying both positive and negative cases correctly. However, throughout all the experiments, we can see that the best individuals are far stronger at classifying the positive cases than the negative ones. We can also see that the plots are the inverse of each other; When an individual is better at classifying positives than the previous one, they are also worse at classifying negatives and vice versa.

The graph in figure 9.33 relays the true positive rate also but this time along with the false positive rate of the best individual of each experiment. The ideal in this situation would be to have the plots with as much space between them as possible to indicate that each positive case is being classified correctly intelligently. However, what we can see is that as the individual is better at classifying positives, they are also more likely to classify negatives as positives incorrectly.

In summary of both graphs 9.31 and 9.33, the most prominent information they relay to us is that an individual is performing in a state where its thinking is: if unsure, classify it as positive.

Chapter 10

Discussion and Conclusion

To conclude the report the following elements are discussed in this chapter: the problems encountered in the development process, comparison of the result achieved to that of previous solutions, future work that can be completed on the project, and finally an overall conclusion.

10.1 Problems Encountered

During this project, I encountered four problems that I found interesting, which I will discuss further below.

10.1.1 Codon Size

Early on in my implementation, when I switched grammars, running the GE function was causing a crash with a value error. I had everything initialised and set correctly to my knowledge, so I was not sure why it was crashing. It turned out to be due to the CODON_SIZE parameter; I had it set to 255 as is the usual CODON_SIZE, but my grammar had 313 production rules for the start symbol, making it was impossible for the mapping to occur correctly. Therefore my CODON_SIZE needed to be greater than or equal to 313; hence from experiment 4 onwards, CODON_SIZE is set to 320 or greater.

10.1.2 TPR as Fitness Function

Another interesting problem I faced was when I used the True Positive Rate as my fitness function. As can be seen in figure 10.1, the individuals' test fitness started at 0 in the first generation, then quickly reached 1 and stayed there throughout. While 1 is the ideal score, I quickly realised that GE was just a greedy algorithm. GE quickly found a way to exploit the fitness function by deciding early on in the evolution process that the intelligent thing to do would be to classify everything as positive, which maximises the possible fitness achieved. Therefore, while TPR is a fair fitness function to use for classification problems, it was not suitable in my case. Hence, I focused on evolving the generations with a fitness function that cannot be exploited by classifying everything as either positive or negative, which was accuracy, and then using the predictions the best individual produced to calculate TPR, FPR, TNR, and FNR.

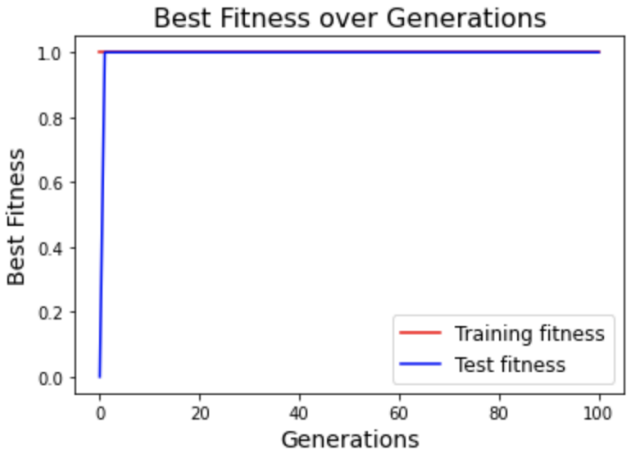

Figure 10.1: Test and train fitness of individuals while using TPR as Fitness Function.

10.1.3 Balancing Exploitation vs Exploration

While tuning hyperparameters, an issue encountered is the balancing of exploration and exploitation. Exploration is when the EA is creating brand new random individuals. Exploitation is when the EA relies on strong individuals of previous generations to create new individuals.

I discovered in some instances that GE was focusing on just exploitation, which resulted in stagnant results. It would find an individual it deemed fit enough early on in the run and stick with it. An example of this happening can be seen in graphs 9.28, 9.22, 9.19, 9.16, 9.7, 9.4, and 9.1. To solve this issue, I had to increase the crossover rate and reduce the elite size so that a larger variety of individuals are being produced in the generations. An example of the improvement this brought about can be seen in figures 9.10, 9.13, and 9.25.

However, as the results improved with the higher crossover rate and small elite size, it became evident that GE was performing too much exploration. Hence, the average fitness suffered as can be seen in figures 9.11, 9.14, and 8.3. The ideal would be to have the average fitness increasing continuously, as would happen through a proper balance of exploration and exploitation; each generation would be getting fitter. However, in these cases, we can see that the average fitness behaves as though each generation is essentially made up of completely random individuals, increasing and decreasing frequently.

10.1.4 Run Time

An issue I faced throughout my implementation was extensive run times which were caused due to my hardware limitations. Because I was using my laptop, which only has two cores, for the implementation, I was faced with extensive run times when executing GE. As a result, I was limited on the number of individuals and generations I could use, the sizes of the trees I could use, and also on the number of experiments and confidence runs I could run. Analysing the average fitness graphs, I would argue that with a stronger computer, to allow for larger populations and generations, better results could be achieved.

10.2 Results Comparison

In this section, I compare my results to results achieved in past projects that also use GE to classify mammograms. These projects are discussed in 2.4.1 and 2.4.2.

Last year's FYP by Sean Morrissey 2.4.1 scored a TPR of 82% compared to my TPR of 79.4% and an AUC of 0.63 compared to my AUC of 0.66 using a single objective fitness function that uses accuracy. Though Sean's implementation performed better with regards to TPR, mine performed better with regards to AUC, indicating a better distinction between negative and positive cases. Unfortunately, Sean did not report any results regarding FPR, TNR, or FNR for comparison.

With regards to Fornells-Herrera et al. 2.4.2, they reported a misclassification percentage of 28.57%. Given that accuracy is the percentage of correct classifications, we can determine that Fornells-Herrera et al. received an accuracy of approximately 71.43% compared to my score of 66.3%. We can see they outperformed my implementation by $\approx 5\%$. However, the approach used by Fornells-Herrera et al. is very different to mine, as the classification of a mammogram is based on the results of two models. Model 0, which is implemented using GE and model 1, which is implemented independently of GE. In comparison, my implementation relies solely on GE.

10.3 Future Work

Some future work and aspects of this project that can be further explored are:

Hyperparameter Configuration

The results I achieved leave room for much improvement, and as can be seen from the experiments in 9, tweaking the hyperparameters can make big changes in the scores received. Therefore, further combinations of values for the hyperparameters can be explored and their effects on fitness. The approach used by Wang et al. in 4.4 would be an interesting approach to help further tune these parameters.

Boundary Determination

Throughout my implementation, I settled on a boundary of 0, i.e. any predictions greater than 0 result in a positive class classification and any predictions

less than or equal to 0 result in a negative class classification. This leaves room to explore whether a higher or lower boundary results in stronger individuals.

Data Sampling

For my implementation, I settled on SMOTE to oversample the data. However, there are many different sampling techniques available, as discussed in 2.2.2. Therefore the effects of another sampling technique could be explored.

10.4 Conclusion

This report has presented the complete implementation of utilising GE to develop a classifier for mammograms. The GRAPE library, which is made available through the Python programming language, was used in order to access GE and its capabilities in this project. The classifier developed operates on Haralick features extracted from mammogram segments to compare textural asymmetry between breasts as instructed by the BNF grammar.

The results achieved in this project are an accuracy of 66.3%, a TPR of 79.4%, and an FPR of 46.8%. These results leave a lot of room for improvement, especially due to the FPR being so high. Though a stage 1 CAD system was produced which are are expected to have a high FPR, I would have liked to achieve a low FPR as with the current classifier classifying a large number of mammograms incorrectly as positive; it's no longer useful for reducing the workload of radiologists, which was among the motivations for the project. Confidence in these results was presented through running the GE process 30 times and documenting the results; the results proved to be as expected with only slight changes in accuracy, which is due to GE's stochastic property.

Multiple experiments were conducted in the project with the aim of achieving stronger results through the tuning of hyperparameters. The experiments presented show the growth of the accuracy of the classifier from 54.9% to 66.6%. The best accuracy is achieved through the combination of a high crossover rate and low mutation rate, and elite size.

Limitations of this project are the use of a boundary of 0 with no exploration of other boundaries and the use of SMOTE to oversample the data without exploring other sampling techniques.

Finally, we can conclude that the research and implementation presented in this report achieve the objective of the project to use GE to develop classifiers for mammograms.

References

[1] H. Wang, Y. Lou, and T. Bäck, "Hyper-Parameter Optimization for Improving the Performance of Grammatical Evolution, https://ieeexplore.ieee.org/document/8790026 , year = 2019," vi, 23, 24, 86

[2] M. Sampat, M. Markey, and A. Bovik, "Computer-Aided Detection and Diagnosis in Mammography, *[online], available: https://www.researchgate.net/publication/249838209_Computer-Aided_Detection_and_Diagnosis_in_Mammography* ," 2005. viii, 1, 2, 3, 4, 6, 23

[3] A. Murphy, "Mammography views, *[online], available: https://radiopaedia.org/articles/mammography-views?lang=us* ," 2021. viii, 7

[4] M. Clinic, *Breast Calcifications* [online], available: https://www.mayoclinic.org/symptoms/breast-calcifications/multimedia/breast-calcifications/img-20007062 . viii, 8

[5] R. Levenson, E. Krupinski, V. Navarro, and E. Wasserman, "Pigeons (Columba livia) as Trainable Observers of Pathology and Radiology Breast Cancer Images, *[online], available: https://www.researchgate.net/publication/284165798_Pigeons_Columba_livia_as_Trainable_Observers_of_Pathology_and_Radiology_Breast_Cancer_Images* ," 2015. viii, 8

[6] S. Narkhede, *Understanding Confusion Matrix* [online], available: https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62 . viii, 13

[7] F. Padró, J. Ozón, and D. Aplicada, "Graph Coloring Algorithms for Assignment Problems in Radio Networks, https://www.researchgate.net/publication/2263212_Graph_Coloring_Algorithms_for_Assignment_Problems_in_Radio_Networks , year = 1995," viii, 14

[8] F. Lerch, A. Ultsch, and J. Lötsch, "Distribution Optimization: An evolutionary algorithm to separate Gaussian mixtures., https://www.nature.com/articles/s41598-020-57432-w#citeas , year = 2020," viii, 15

[9] D. Gavrilis, G. Gerogoulas, I. Tsoulos, and E. Glavas, "Classification of fetal heart rate using grammatical evolution., https://link.springer.com/chapter/10.1007%2F978-3-662-44303-3_14 , year = 2005," viii, 22

[10] NBSS, *NBSS User Guide* [online], available: hhttps://www.dropbox.com/s/dl/158kdzsbfdyd9pp/1630%20DS0026%20NBSS%20Crystal%20Reports%20Tables_2015.pdf . viii, 33

[11] Q. Do, M. Lewis, A. Madhuranthakam, Y. Xi, A. Bailey, R. Lenkinski, and D. Twickler, "Texture analysis of magnetic resonance images of the human placenta throughout gestation: A feasibility study, https://www.researchgate.net/publication/330550795_Texture_analysis_of_magnetic_resonance_images_of_the_human_placenta_throughout_gestation_A_feasibility_study , year = 2019," viii, 34

[12] CDC, *What is a Mammogram?* [online], available: https://www.cdc.gov/cancer/breast/basic_info/mammograms.htm . 1

[13] F. Noorian, A. M. de Silva, and P. H.W. Leong, "Grammatical Evolution: A Tutorial using gramEvol, *[online], available: https://cran.r-project.org/web/packages/gramEvol/vignettes/ge-intro.pdf* ," 2020. 3, 17

[14] WHO, *Breast Cancer* [online], available: https://www.who.int/news-room/fact-sheets/detail/breast-cancer . 3, 4

[15] Cancer.Net, *Breast Cancer: Statistics* [online], available: https://www.cancer.net/cancer-types/breast-cancer/statistics . 3

[16] RadiologyInfo.org, *Mammography* [online], available: https://www.radiologyinfo.org/en/info/mammo . 6

[17] C. Ryan, K. Krawiec, U.-M. O'Reilly, J. Fitzgerald, and D. Medernach, "Building a Stage 1 Computer Aided Detector for Breast Cancer Using Genetic Programming, *[online], available: https://link.springer.com/chapter/10.1007%2F978-3-662-44303-3_14* ," 2014. 6, 7, 8, 21, 33

[18] M. Clinic, *Dense breast tissue: What it means to have dense breasts* [online], available: https://www.mayoclinic.org/tests-procedures/mammogram/in-depth/dense-breast-tissue/art-20123968 . 7

[19] M. Wallace, "Emerging Artificial Intelligence Applications in Computer Engineering., *[online], available: https://books.google.ie/books?hl=en&lr=&id=vLiTXDHr_sYC&oi=fnd&pg=PR1&dq=Maglogiannis+I.,+Karpouzis+K.,+Wallace+M.,+Soldatos,+J.+(2007).+Emerging+Artificial+Intelligence+Applications+in+Computer+Engineering+-+Real+Word+AI+Systems+with+Applications+in+eHealth,+HCI,+Information+Retrieval+and+Pervasive+Technologies.+Emerging+Artif&ots=CZosyv3Jil&sig=lpnenUEzyYNI68e89YRcAoeE9is&redir_esc=y#v=onepage&q&f=false* ," 2007. 8

[20] S. Asiri, "Machine Learning Classifiers., *[online], available: https://towardsdatascience.com/machine-learning-classifiers-a5cc4e1b0623* ," 2018. 8, 9, 10

[21] J. Korstanje, *SMOTE* [online], available: https://towardsdatascience.com/smote-fdce2f605729 . 11

[22] Y. Genome, *What is Evolution?* [online], available: https://www.yourgenome.org/facts/what-is-evolution . 11

[23] V. Mallawaarachchi, "How to define a Fitness Function in a Genetic Algorithm?, https://towardsdatascience.com/how-to-define-a-fitness-function-in-a-genetic-algorithm-be572b9ea3b4 , year = 2017," 11, 12

[24] A. F. Gad, *Evaluating Deep Learning Models: The Confusion Matrix, Accuracy, Precision, and Recall* [online], available: https://blog.paperspace.com/deep-learning-metrics-precision-recall-accuracy/ . 12

[25] S. Narkhede, *Understanding AUC - ROC Curve* [online], available: https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5 . 12

[26] A. Fornells-Herrera, E. G.-R. , E. B. -Mansilla, and J. M. -B. ı, "DECISION SUPPORT SYSTEM FOR THE BREAST CANCER DIAGNOSIS BY A META-LEARNING APPROACH BASED ON GRAMMAR EVOLUTION., *[online],*

available: https://sci2s.ugr.es/keel/pdf/keel/congreso/fornells06MGE.pdf ," 16

[27] D. H. Willacy, *Cardiotocography* [online], available: https://patient.info/pregnancy/cardiotocography . 22

[28] Python.org, *What is Python? Executive summary.* [online], available: https://www.python.org/doc/essays/blurb/ . 26

[29] Statisticstimes.com, *Top Computer Languages 2021.* [online], available: https://statisticstimes.com/tech/top-computer-languages.php . 27

[30] A. BROTHERS, *Why is Python so popular in machine learning and AI?.* [online], available: https://asperbrothers.com/blog/why-python-for-machine-learning/ . 27

[31] Jupyter.org, *Project Jupyter.* [online], available: https://jupyter.org . 27

[32] O. D. Science, *Why You Should be Using Jupyter Notebooks.* [online], available: https://odsc.medium.com/why-you-should-be-using-jupyter-notebooks-ea2e568c59f2 . 27

[33] PyPI, *deap.* [online], available: https://pypi.org/project/deap/ . 28

[34] Pandas.pydata.org, *Package overview — pandas 1.3.5 documentation.* [online], available: https://pandas.pydata.org/pandas-docs/stable/getting_started/overview.html . 28

[35] ActiveState., *What Is Pandas in Python? Everything You Need to Know.* [online], available: https://www.activestate.com/resources/quick-reads/what-is-pandas-in-python-everything-you-need-to-know/ . 28, 29

[36] Matplotlib.org, *Matplotlib — Visualization with Python.* [online], available: https://matplotlib.org . 29

[37] P. Barrett, J. Hunter, J. Miller, J. Hsu, and P. Greenfield, "Matplotlib - A Portable Python Plotting Package., https://adsabs.harvard.edu/full/2005ASPC..347...91B , year = 2005," 29

[38] G. Docs., *Hello World.* [online], available: https://docs.github.com/en/get-started/quickstart/hello-world . 29

[39] P. Brynolfsson, D. Nilsson, T. Torheim, T. Asklund, C. Thellenberg Karlsson, J. Trygg, T. Nyholm, and A. Garpebring, "Haralick texture features from apparent diffusion coefficient (ADC) MRI images depend on imaging and pre-processing parameters., https://www.nature.com/articles/s41598-017-04151-4.pdf , year = 2017," 33, 34

[40] B. Sebastian V, A. Unnikrishnan, and K. Balakrishnan, "GREY LEVEL CO-OCCURRENCE MATRICES: GENERALISATION AND SOME NEW FEATURES., https://arxiv.org/pdf/1205.4831.pdf , year = 2012," 34

[41] N. Dey, J. Chaki, and R. Kumar, "Sensors for Health Monitoring, https://www.sciencedirect.com/book/9780128193617/sensors-for-health-monitoring , year = 2019," 34

[42] Omnisci, *Data Exploration - A Complete Introduction.* [online], available: https://www.omnisci.com/learn/data-exploration . 35

[43] A. Murphy, *XCCL view* [online], available: https://radiopaedia.org/articles/xccl-view?lang=gb . 38

[44] A. Murphy, *Mediolateral view* [online], available: https://radiopaedia.org/articles/mediolateral-view?lang=gb . 38

[45] A. Murphy, *Lateromedial view* [online], available: https://radiopaedia.org/articles/lateromedial-view?lang=gb . 38

[46] S. Anunaya, *Data Preprocessing in Data Mining -A Hands On Guide - Analytics Vidhya.* [online], available: https://www.analyticsvidhya.com/blog/2021/08/data-preprocessing-in-data-mining-a-hands-on-guide/ . 39, 40