To Create and Compare the Predictive Accuracy of a Genetic Program and an Artificial Neural Network to Predict Corporate Bankruptcy: Final Report

Carl Saptarshi
Student Number: 640032165

April 2017

Contents

1 Background and Introduction

Due to the dynamic and volatile economy that we live in, the number of companies filing for Corporate Bankruptcy (CB) are rising, especially in times of economic uncertainty, for example, during a period of recession. In turn, being able to predict the likelihood of a corporation going bankrupt and filing for bankruptcy is very important and has been a focal point of issue in accounting research and analysis over the past thirty years [**?**].

At present, copious amounts of historical financial data can be extracted from a range of small, medium and large companies. Through looking at some of this data, we can see whether a company has declared bankruptcy or not as yet. Unfortunately, there is not a straightforward way to identify whether a company is *currently* financially distressed and the likelihood of the company going bankrupt in the foreseeable future based on their raw data alone. By selecting appropriate key performance indicators (KPI's) - financial variables that are known to affect a company's performance the most - this data can be manipulated and combined in various ways to help find a way to predict if a company is likely to go bankrupt and if they are financially distressed. The combination of these KPI's should be able to classify any sized company.

A ***genetic program*** (GP) can be used to generate and evolve mathematical expressions to produce an intuitive function that could classify whether a company is financially distressed and likely to go bankrupt. Therefore, the aim of this dissertation is to create a GP which will produce a function that will predict if a company is likely to go bankrupt. This model will then be compared against other benchmark models already made to compare their predictive accuracies.

1.1 Background into Bankruptcy

1.1.1 What is Bankruptcy

When a company (the *debtor*) takes out a loan or borrows money from somewhere like a loan company(the *creditor*) , it is up to the debtor to ensure that the creditor is repaid the full amount that was borrowed subject to the creditors terms and conditions.

If the debtor starts to fall behind on their payments and are unable to repay their debts, the debtor may file for a Chapter 7 bankruptcy in which the court will appoint a trustee to shut down the company and liquidate their assets, for example, by selling machinery, land and company shares to recover some money which they the trustee can give back to the creditor to clear the company's debt. If the company is still unable to pay back the debt even after this, the company will be terminated. As economies have grown rapidly since 1960's, especially in the western part of the world, bankruptcy has been recognised on a much grander scale [**?**]

1.1.2 Who does it Affect?

Bankruptcy does not just affect those that are employed within that company; it also affects third party members such as shareholders, investors, and company clients. CB has been an area of interest, especially in the field of financial analysis and for stakeholders who are interested in the performance of the company [**?**]. Since the 1960's, empirical risk assessment models have been developed which have been used to predict CB.

Using financial data of a company, the likelihood of CB can be predicted for *'n'* number of years ahead. Loan companies can use this information to determine whether loans should be granted to a corporation, as they will know the likelihood of a company defaulting or not [**?**]. This has helped to

give banks competitive advantages, as they become aware of how likely a company will be to default, and is able to predict customer behaviour in times of difficulty [**?**].

Looking at reports from the American Bankruptcy Institute [**?**] showed that in the year 2000, 35,742 companies filed for bankruptcy, 43,546 companies in 2008 and 60,837 by 2009, at the peak of the recession. By 2012 this fell to 40,075 and 24,114 by 2016 . These statistics clearly indicate the volatility and uncertainty in the economy as it changes, which is part of what makes CB prediction incredibly important.

1.2 Algorithms for Corporate Bankruptcy Prediction

As the aim of the project to classify whether a company is likely to go bankrupt, this type of problem can be called a binary classification problem. This takes a series of inputs and returns a classification which determines whether or not a company is financially distressed. Companies likely to suffer from financial distress will have certain characteristics associated with them, similarly for companies not facing this problem. This means that the data being used should be linearly separable when classifying the data, making this a linear binary classification problem. As it is difficult to compare very small companies with very large corporations, to make them comparable, the data used tends to be standardised. This helps to scale the data into units where all companies of any size can be compared.

There have been several techniques that have been used to predict CB, some of which will be introduced here.

Individual Ratio Selection (IRS) -This process involved selecting 30 financial variables, and converting these to ratios. Based on a threshold for each variable, this would determine if a company is financially distressed or not.

Multivariate Discriminant Analysis (MDA) - Altman created this technique in the 1960's, which takes uses a discriminant function to score a company [**?**]. This function uses five financially weighted ratios. Based on the overall discriminant score, the company can be classified as financially distressed or not.

Supervised Learning (SL) - Some of the current methods to predict CB fall under the umbrella of Machine Learning (ML). ML is a form of Artificial Intelligence that allows a computer program to learn without the use of explicit programming [**?**]. This means whilst the program is running, it will start to determine patterns in the data, and adapt the program appropriately to try to produce the best predictions and classifications. In SL, the output after each iteration of each model is compared against the already known desired output. This can then be used for checking the model's classification accuracy. For every iteration, the model will start to learn, so the classification accuracy will start to improve over time. Eventually, when an unknown set of data (hold out data) is inputted into the model, it will be able to correctly produce an output to declare if the company in question will go bankrupt or to a certain degree of accuracy.

Genetic Program (GP) - GPs fall under the umbrella of Evolutionary Computing. Algorithms that belong to the EC family are inspired by biological evolution and used for optimisation problems. A Genetic Algorithm is one such member, inspired by Darwins Theory of Evolution. Genetic Programs are a type of GA which have been used for prediction and classification. A population of functions is created where fitter individuals in the population are more likely to survive and produce offspring that are (in theory) more suited to their environment. The aim of this is to create an optimal function which can give the most accurate classification prediction on unknown data to see if a company is likely to go bankrupt within one year.

Artificial Neural Networks (ANN) - Also known as the Multi Layer Perceptron (MLP), this technique is inspired by the interconnectivity of the brain and applies this to prediction and classification [**?**]. ANN's are made of an input layer, hidden layers and an output layer which produces a

classification based on the input. The network uses the weights on its neural that connect each node to the next layer, which are tweaked, based on the accuracy error, to allow the network to learn and give an accurate classification for unknown data.

In this report, I will be discussing how I developed and compared an ANN and GP to predict CB. I start by introducing some of the research that was conducted to gain a deeper understanding of what this project entailed. Using this research, I then discuss the requirements that were needed to complete the project. When talking about ANN's, assume that the ANN structure follows the format *(input, Number of nodes per layer, output)* format. Using this research, I then formulate my design specification which was used to form the structure of the models that were implemented to test the predictive accuracy of the two models. After this, I will go on to talk about how the models were tested individually and compared against each other, before giving an evaluation of the project that has been completed. After this, I will then mention work that could be completed in the future to potentially improve this project, before coming to final conclusions.

2 Summary of literature review and specification

2.1 Literature Review

All the techniques mentioned in section 1.2 have been used extensively in the area classification and prediction. Altman and Ohlson , pioneers of CB prediction since the 1960's selected financial variables from multiple company's bank statements to predict CB [**?**]. These variables (*key performance indicators* (KPI's)) were used as they believed these were important factors that indicated a company's performance. To make companies more comparable, both techniques involved converting the KPI's into ratios as a method of standardising the data. Newer models proposed are based around Altman's financial ratios and use their prediction accuracy for MDA and IRS as benchmarks to compare their new proposed work against techniques already in use.

Beaver introduced the idea of using financial ratios which could be used for CF prediction. Financial ratios were selected one at a time. Using the results of each of the outputs of the ratio values, they would be used to give an overall prediction as to whether or not a company would fail.These ratios would be used in order to get a binary classification as to whether a company is likely to fail or not. For each ratio, a threshold value was set. If the ratio was below the threshold, it failed, otherwise it would not fail.

Altman used MDA in order to approach the task of CB prediction. He took an empirical set of financial variables and created financial ratios which were KPI's that would be associated with failure prediction. This was essentially a linear model which was used in order to classify between non-failed companies and companies that are likely to fail.MDA allows for multiple ratios to be used as inputs and to be associated with weightings, to provide a classification of a selection of ratios simultaneously, making this very accurate and efficient, producing 95% predictive accuracy.

The MDA technique was favoured as used multivariate data at once to get an overall prediction rather than taking each ratio and scoring that to give predictions [**?**], making this superior to IRS, but only if the KPI's were jointly distributed according to a multivariate normal distribution [**?**].
Wilson [**?**]and Lensburg [**?**] used newer approaches by implementing ANNs and GP's respectively to this classification problem. Both of these techniques can handle noisy data that is unevenly distributed better than Altman's MDA, showing that both ANNs and GP's have potential to be more accurate than IRS and MDA.

ANNs tend to perform very well in terms of performance and accuracy. Wilson et al used an ANN

approach with a (5, 10, 2) structure [**?**]. To improve predictive accuracy, the Monte-Carlo technique was used to give a better representation of classifications. Overall, they achieved a 97.5% accuracy on their testing dataset, making this much more accurate than MDA and IRS.
Lee used a decision tree (DT) method to predict CB [**?**]. Lee used 8 different KPI's when approaching this problem. Using this GP model, the testing accuracy of 92.91%. Rostamy [**?**] used a similar approach to Lee, using five different KPI's. After training the GP, it could correctly predict if a company would go bankrupt or not 90% of the time, which was like MDA, however more flexible in terms of the type of data that could be used.

GP's may work slower as they explore a large search space and may be restricted to certain limitations e.g. a maximum tree depth. For each crossover and mutation, the depth of tree may increase. This can increase the computation time rapidly. When designing and implementing the GP, these factors will typically be accounted for as seen in Etemadi's et al paper [**?**].

Through the research completed, many papers used Altman's KPI's as their inputs. However, as Altman suggested, these ratios may not necessarily be the most optimal, but these still provided the best alternative discriminant function to work with at the time [**?**]. Since then, economies have changed significantly, these ratios may not necessarily be the best to use to predict CB, but may still be significant enough to give an accurate enough prediction. As seen by Back, Rostamy and Lee[**?**], other ratios have been used to predict CB, and achieved similar results to MDA, which could potentially be more significant now. Wilson used Altman's KPI's and achieved the 97.5% accuracy, with far fewer ratios relative to Back and Lee, which must be taken into consideration [**?**].

2.2 Project Specification

After careful consideration of the researched techniques, I will be predicting CB using ANNs and GPs due to their strong predictive accuracy rates and ability to handle noisy data. Though they do have drawbacks, I will aim to minimise these through the project specification and implementation.

As the models used in the research depended on various datasets, the first thing that needed to be acquired was a dataset with an enough data, and containing enough variables that could give an indication as to whether a company went bankrupt or not. The decision to use only small to medium enterprises (SME) was to have more consistent, comparable data, as shown by Altman. Using this dataset, I would then able to create financial ratios which can then be used in the models to be developed.

Two programs were intended to be made for this project; a feed forward ANN with back propagation and secondly an expression tree GP. I have chosen to take forward these two models is that ANNs are known to produce very accurate results efficiently. The reason an expression tree GP has been put forward is because this a valid technique that can be used as they are able to produce a function that will directly map the input KPI's to give a classification.

Both techniques should give an indication as to which KPI's affect the classification prediction, and give an indication as to which KPI's affect the companies and could be a cause of their failure.
As this is a binary classification problem, the output of each model determines whether a company can be classified as likely to go bankrupt or not. To represent the classification, 0 will represent a company that is not likely to go bankrupt and 1 will represent a financially distressed company.

Since this representation is a number, the actual floating point value of the output (which will be between 0 and 1) for the ANN, this number will be used to represent the likelihood of failure as a probability. For example, if the value was 0.618, it could be said that the company has a probability of 0.312 of staying afloat for the forthcoming year. Whereas if the value was 0.111 then the company

has a 0.899 probability of staying afloat for the forthcoming year. Here, a clear differentiation can be made between two companies, one which is more likely to fail than the other. Once the development part of the project is complete, I planned to test the results of both models, individually against each other to compare their predictive accuracies, and against Altman's benchmarks results.

To allow the ANN to learn, I will be using the training dataset. When testing this, I will use a testing dataset to validate the results to see the true accuracy. I will also use K-Fold Cross-Validation techniques to help improve the comparability and the accuracy as well.. To increase comparability, I will also use the same datasets for both ANN and GP.

I will also consider the number of iterations that have been used to complete the task in both ANN and GP. The time it takes to process the inputs will also be compared. To determine what KPI's have a greater contribution than others, for the variables that are being used for ANNs, I will remove one ratio, rerun the programs under the same training and testing conditions to then check the outputted value to see how significantly different the output is with and without that ratio. This will help to determine what ratios have a greater weighting and could be a significant factor in the future success or demise of a company.

Overall, to make my project successful, I will take these factors into account before starting to program, to prevent any long-term errors that could occur. I will also use a reliable GUI to help prevent programming errors, which in turn will make my project more successful.

3 Model Design

This project follows the Waterfall Development methodology, a subset of the Software Development Life Cycle(SDLC). The reason this was chosen is to give a clear structure to the way that I planned on completing the project.

The first stage completed was collecting the data to use. After this, I use the research to structure the design for the ANN, and the GP, before mentioning how I planned to test the models that were designed. Finally, I briefly introduce the programming language that I chose to use before the development process began.

3.1 Data Collection

Firstly, I had to collect all the data that I planned to use. The data that collected had been given to me by the University of Exeter Business School as they had access to multiple years' worth of data for several companies. As each economy is different, companies may perform better or worse in different environments, therefore to make the data more comparable, the data that collected was solely from American companies from the US economy. The reason data had to be collected in the first place is because for a machine to learn, they need to have data to work with, to understand the data and start forming patterns accordingly.

The dataset contained 671 rows of data, with 31 different variables from small, medium and large companies. Out of these, only 8 were financial variables KPIs. For example, *net income* and *sales* were KPIs that affected the company performance, however *company name* and *ticket*, would be much less likely to affect a company's performance. Therefore, variables not considered to be KPIs were removed from the dataset. Any rows with incomplete data were also removed. These were rows of data in which a cell contained a question mark (?) in any of the cells for a company as this could have changed the way the models learn. To make smaller companies more comparable to larger companies, the KPIs were standardised into five ratios, similar to the methods that Altman used to standardise his dataset. This

new standardised dataset contained 5 KPI ratios and their associated classification, labelled **X1,..., X5**, each representing a specific ratio, and would be used as inputs for the machine to learn:

X1 - working capital / Total Assets

X2 - Retained Earnings / Total Assets

X3 - Earnings Before Interest and Tax / total Assets

X4 - Market Value of Equity / Total Debt

X5 - Sales / Total Assets

Failed - 0 or 1

For an explanation about these KPI's and why these ratios were chosen, please refer to Appendix [**?**].

3.2 Artificial Neural Network Design

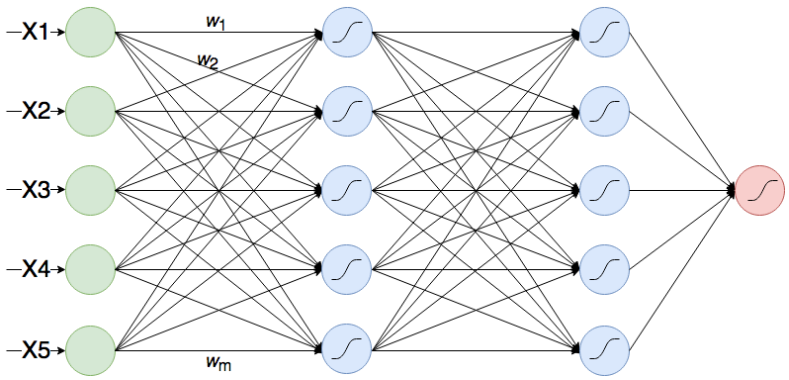

Figure 1: Structure of a Feed Forward Artificial Neural Network. Green nodes: input features. Blue nodes: hidden nodes and hidden layers, and a logistic activation function. Red nodes: response vector (the output) and a logistic function.

I chose to implement a feed forward artificial neural network (FFANN). Based on the research conducted in section 2.1, their results [**?**], proved to be more than adequate for this problem.
A FFANN usually follows the same layout which makes them applicable to an array of complex problem. The basic structure of the ANN can be seen in Figure 1, consisting of an input layer, hidden layer(s) and an output layer. The input layer consists of *feature vectors* representing the data inputs as a single layer of nodes. The hidden layer(s) consist of hidden nodes, used to help alter the representation of the data by transforming it, reducing any non-linearities in the data. The output is a *response vector* that consisting of a single node, which will output the predicted classification. Connecting the nodes in each layer is accomplished using weighted neural synapses, that connect each node in the current layer to each node in the next layer.

Since ANNs have been used numerous times for this problem, the FFANN was designed as a benchmark for predictive accuracy on this particular dataset, which could then be compared against the GP that was developed.

3.2.1 Input Layer and Synaptic Weights

Since ANNs can cope with high dimensional, non-linear data, I decided to use all 5KPI ratios as the input feature vectors from the dataset as the ANN would be more than capable to handle all this data at once.

Due to the stochastic nature of an ANN, when initialising the network before the data from the input features are read into the network, the synaptic weights needed to be initialised. As part of the design, I chose to randomise the weights on each of the synapses. If the weights were not initialised randomly, when the network learns, it would learn in the exact same way every time the program is run because the network will always start at the same position in the search space. This means it will always follow the same routine to get to a solution which would always be the same since is it predictable. By initialising the weights randomly, the symmetry of the network is broken, which allows the network to be initialised differently, so the weighted input signal that moves from one layer to the next would always be different, allowing more of the search space to be explored, and more (possibly optimal) solutions to be found.

3.2.2 Hidden Layers and Activation Function

Figure 1 shows an example of a FFANN with a (2,5,5,1) structure. Through research, it was found that when using one hidden layer with multiple nodes, the network is more likely to start memorising the data being passed to it, which can cause the model overfitting[**?**]. This means that when testing the predictive accuracy of the hold out data, the results would be poor as it has been unable to generalise the model. Therefore, to avoid this, I decided to use multiple hidden layers as this would allow the network to generalise better without network memorisation. To make it easier for the user to input their own ANN structure, I decided to provide a configuration file for custom FFANN configuration file to allow a user to input their own structure.

Each neurone in the hidden layer transforms the values from the previous layer with a weighted linear summation.The cumulative sum of these products (z) is used as input to the next node in the hidden layer, which then is passed through a nonlinear activation function. For the ANN that is being designed, I initially used a *logistic activation function*, as indicated by the logistic curves in each of the hidden nodes in Figure 1 and equation 1.

$$activationL(z) = 1/1 + e^{-z} \tag{1}$$

The logistic function adds an element of non-linearity to the to the model, which allows the computation of nontrivial problems using only a small number of nodes, making this function much more popular relative to other activation functions like a *step function*.

3.2.3 Output Layer and Learning Mechanism

The output layer receives values from the final hidden layer and transforms them into output values. I planned to classify the outputs as either a 0 or 1 by passing the raw output through another logistic activation function. If the new value was below 0.5, then it was classified as 0, otherwise 1. Using this, the error percentage could be measured against the true data classifications, and the weights on the synapses were be altered accordingly using a learning mechanism. For this network, the a *back-propagation* learning mechanism was needed to allow the network to learn, and try to minimise the error rate by manipulating the weights on the synapses, to give the best possible accuracy, showing that the network has trained.

3.3 Genetic Program Design

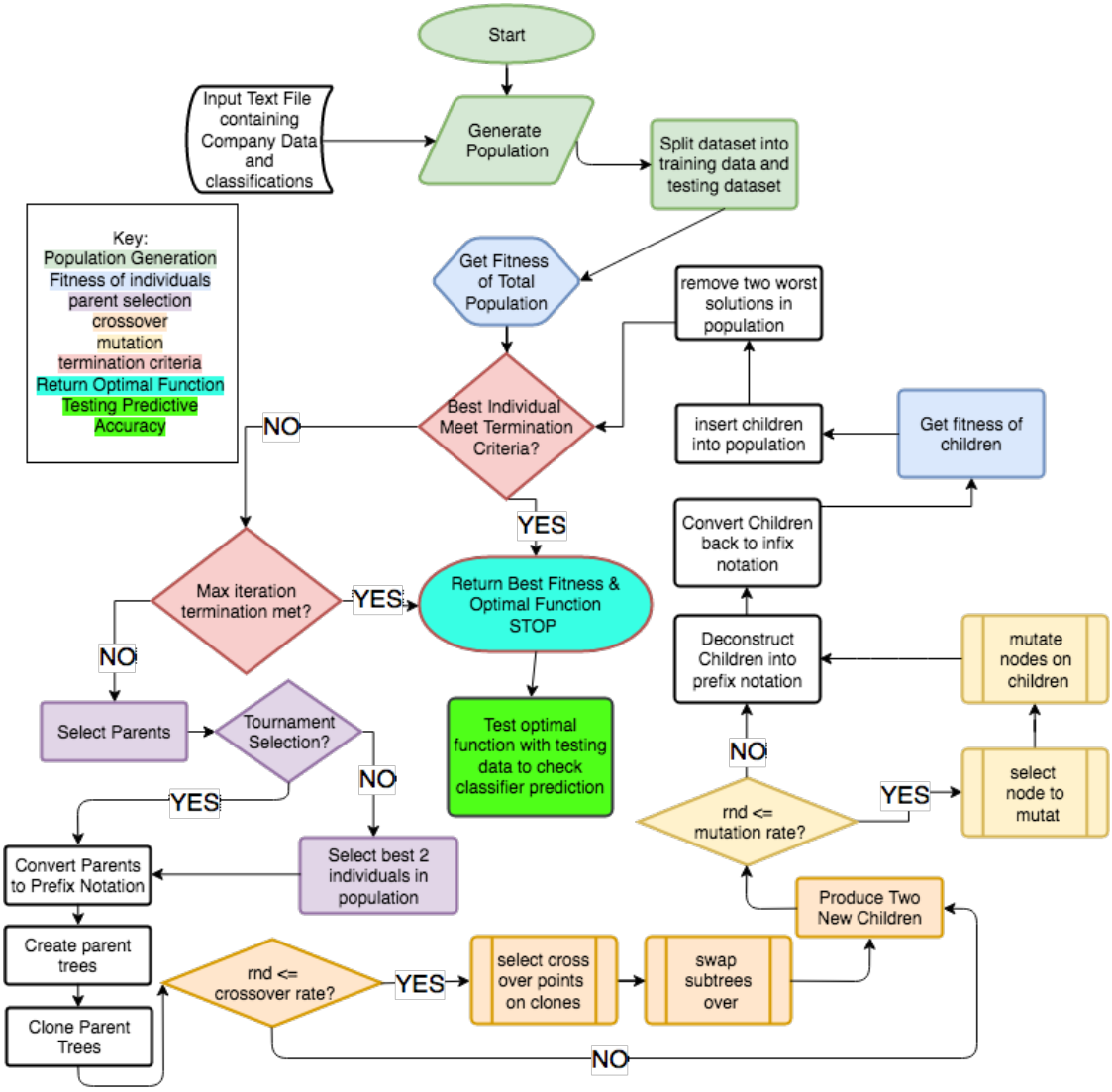

Figure 2: Flow chart to represent the Genetic Program that was designed. Dark green boxes: population generation. Red boxes: termination criteria. Purple boxes: selection methods. Orange Boxes: genetic crossover. Yellow boxes: genetic Mutation. Blue boxes: fitness function. Turquoise box: optimal function returned. Light green box: use of optimal function on testing data set.

The aim of the GP model was to produce an optimal function can predict to a degree of accuracy, whether a company is likely to go bankrupt or not within one year, based on unknown input data. As this type of model is know to be able to handle noisy and non-linear data well, I decided to use all the 5 KPI ratios for my inputs, the same inputs for the ANN. This allowed for data consistency and would make the models more comparable in the testing phase.

3.3.1 Data Representation

Through out this project, each member of the population had to have some form of representation. Here, I mention some of the ways that the data could be represented which I planned to implement in section 4.

An example of a random population generated containing 4 members can be seen below. Each member is represented as a string, using standard infix notation; the entire population is stored within a list. Infix notation is a format expressions tend to be written in, where 2 operands surround an operator. The reason for this is because when an optimal function is found, it will be returned as a mathematical function, which are typically represented in this particular format.

$$pop_n = [\text{“}X1 + 1.3 - X3 + X2/X4 * X5\text{”}, \text{“}X2/X5 + (6.433 - X1) * X3 * X4\text{”},$$
$$\text{“}X1 * 9.56 - (X2 * (X5 * X4)) - X3\text{”}, \text{“}4 + 8.22/X1 * X2 - 7 + X3/X4 * X5\text{”}]$$

To make the crossover and mutation process simpler, I decided to represent the parents using lazy instantiation of a *binary expression tree* data structure. Lazy instantiation is the process of performing an action only when required. Binary trees are make the process of genetic crossover and mutation significantly easier as the model learns as they can perform insertions and deletions of elements efficiently. The reason I chose lazy instantiation was because it would save computation time from having to generate binary trees for every single member of the population, even though only the two selected parents will be going through the genetic operators to create children. Figure 3 is an example of a two expressions converted into binary trees. After the processes of genetic crossover and mutation are completed, the children need to be put back into the population if they are fitter than the worst members in the population. Therefore, since the children have to be in the infix format, the binary trees have to be parsed back into an infix notation which the population will accept.

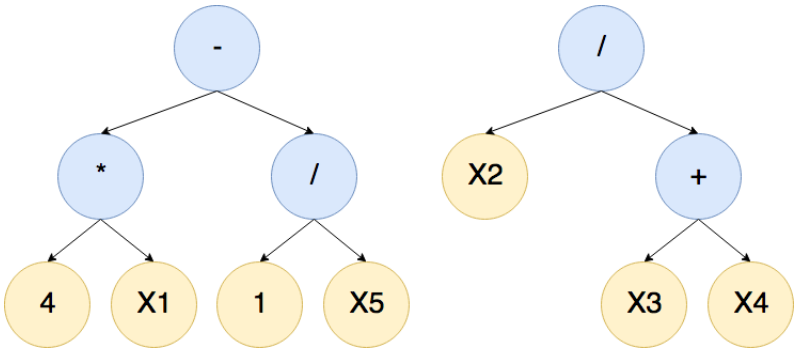

Figure 3: Infix expressions: “4*X1-1/X5” and “X2/(X3+X4)” converted into binary trees. Blue nodes: operators. Yellow nodes: operands.

3.3.2 Generating a Population

It is common practise to initialise a population randomly in GPs. The reason for this is to allow different populations to begin in different parts of the search space, such that they can explore different areas to try to find the global optimum solution, which in this case would be the optimal function. This population would consist of randomly created mathematical functions, eachrepresenting one member of the population. A mathematical function typicaly consists of two parts; a functional set of operators (*+,-,*,/*) and terminal set of operands (numerical constants and variables *X1,...,X5*). To create an random member, I decided to design a mathematical expression generator. This would allow random,

valid functions to be created using randomly selected members of the functional and terminal sets and would represent an individual's *genotype* - the elements that make up the function. To begin with, a population would consist of 500 randomly generated individuals. Based on the arities of the functional operators, the randomly generated population was based on the *full* method of population generation [**?**] rather than *grow* or *ramped half and half* [**?**]. This provided sufficient variety in expression generation.

3.3.3 Fitness Function

The fitness function would represent the *phenotype* of an individual, i.e. how good the individual is suited to its environment, representing how close it is to an optimal solution. This was done is through calculating the error between the true classifications of the data set and the predicted output as given by the function produced. The aim of this is to minimise the error between the actual classifications from the dataset and the number of incorrectly predicted classifications produced by the function where the optimal function which correctly classified every company would have a fitness value of 0. This can be called a minimisation problem where the objective is to minimise the fitness function.
For this, I used the *Number of Hits fitness function* [**?**] . This fitness function is calculated by firstly passing each row of data through the function. For each row of data, the output will either be a positive or negative number given by this function. This output was then passed through a logistic function. If the value is greater than 0.5, it was classified as 1, otherwise 0. Using these classifications for each predicted row of data, these can now be compared against each of the true classifications from the data. For every row of data that is classified correctly, the fitness value does not increase. However for every misclassification, the fitness value for that function increases by 1. This shows that a good function which has more number of hits would have a lower fitness value, and a function which has a high misclassification rate has a high fitness value.

3.3.4 Selection

For most GPs, genetic operators like crossover and mutation are applied to certain individuals in the population based on their fitness value. The better the individual is in the population, the more likely they are to produce children functions.to select these parents, I chose to use *tournament selection* [**?**]. This works by selecting a random subset of the population, t, and then comparing the fitnesses of each of members of t. The individual with the best fitness value i.e. closest fitness to 0, will be selected as the first parent. This same process will occur to select the second parent. An individual can only be selected once per generation, as the same member of the population cannot be selected to be both parents. This will keep the selection pressure relatively constant as it does not favour the best individuals in the population, rather it selects the best individuals in the subset of the population selected as candidate parents. The reason this method was chosen was to help reduce selection pressure, which favours the very best individuals. By reducing this, it enables more members to be selected as candidates, and be chosen as parents, even if they were not the best solutions in the population.

3.3.5 Genetic Operations

Using the parents, the children can be created using the genetic operators - *crossover* and *mutation*. Typically, in a GP, crossover occurs first. I decided to use a *single subtree* crossover between the parents, to create two children, each containing characteristics of both parents. After the two new children have been created, the genetic operator, *mutation* can be used to mutate nodes on the children. This was used to help maintain genetic diversity within the population from current generation to the next. Here, I chose to use *node replacement* mutation, where a random node on each of the children will be

selected and be altered based on its arity. For example, if the node selected is a numerical value, with an arity of 1, then this will be replaced by another value with an arity of 1, and similarly if the node selected has an arity of 2, then it will be replaced by another functional node.

3.3.6 Termination Criteria

The termination criteria was checked in two places. The first termination criteria was if an optimal function has been found in the population. Since there was a possibility of this occurring when the initial population is evaluated, this was checked before the program GP continues. If an optimal solution existed, then return the best individual. After n generations of running the program, if the best individual in the population still had not been able to meet the optimal fitness criteria, then the GP should stop and return the fittest individual in the current population as this would have a fitness value closer to 0 compared to all others in the population.

Using the optimal function, it should be tested on the holdout dataset to determine its predictive accuracy of the function to see if it has predicted whether a company is likely go bankrupt correctly. This output will then be passed through the same logistic function (eq. 1). Using a 0.5 threshold. If the value outputted from this function is greater than 0.5, it would be classified as 1, otherwise 0. This can then be used to give the classification predictive accuracy of the function. The reason this was done is because when training the model, to ensure consistency in the predicted classifications, the thresholds were kept the same.

3.4 Testing Design

After the development is completed, various tests could be conducted on both the ANN and GP to see if these tests would alter the learning process and predictive accuracy which will be mentioned now and in the testing phase of the dissertation (section 5). The design of the tests has been based on the project specification and on the way the models have been developed. The first type of testing I chose to design was manual testing for the ANN and GP. This will be running the models' multiple times with smaller datasets with already accepted solutions. This will be able to check whether the models are working and performing as they should be. During the development of the program, I decided to use software testing to determine whether the functions that are being made are functioning in the correct way. After this, I could then perform validation testing by running the models multiple times to ensure that they are both learning the way they should be, to get to an optimal result .To do this, *unit* testing , and *black-box* testing could be used to determine whether each function performs the way it should by comparing the desired output of the function to the actual output of the function, and check whether they match. After this, performance testing could have completed on both models, to ensure that they are learning and predicting at an acceptable rate. This will be completed by testing the performance of each function and the overall model to identify the areas of the model which may perform unusually slow. Finally, after the development and testing has completed, I *user acceptance* testing was designed to ensure that the program works and could be used as a product that could be deployed in the future.

3.5 Programming Language and Environment

The final part of the design stage was to determine what programming language and programming environment that could be used for this project. Through careful consideration, I decided to use Python as my main programming language. The main reasons for this were that since the ANN was going to be used as a benchmark, rather than trying to implement an ANN from scratch, I chose to

find a library - *sklearn*, as this provided all the functionality to allow a FFANN with back propagation methods to be created, along with many other functions which could be used to test the network e.g. using other transfer functions, methods of splitting the data and how to train and test the network. For consistency, I also decided to use Python as the choice of language for the expression tree GP. This was because of the fact that Python offers object orientation, speed, high level abstraction for memory allocation and offered others optimised mathematical libraries such as Numpy which can make data handing easier, and Matplotlib which offers easy to make graphs which can then be used for data analysis.

4 Development

To begin with, I developed the models by attempting to solve smaller problems first, with static data. This way, debugging and understanding the errors would be easier. To make the testing of the models more reliable, I decided to use the same dataset for the ANN and the GP.

Since the dataset was the same for both model, I used the same method to read and split the data sets from their associated classifications. As this was a SL problem, the decision to split the data was because the classifications from the dataset could be used to form the fitness functions for both techniques that were implemented.

4.1 Development of the Artificial Neural Network

As part of the design stage, I chose to use a library in Python - Sklearn, a well-known machine learning library that has the facilities to develop a feed forward ANN quickly, the ability to train, test, and present the outputs of the data in a variety of ways.

4.1.1 Multi Layer Perceptron Classifier

To initially read the data, a function *read_data* was used. This took the dataset, shuffled it, and then split the dataset such that the data was stored in a variable *data_CBD* and the classification labels associated with each row of the data stored in class_labels_CBD. To maintain type consistency, since the data was read and stored using the Numpy module, the classification array was casted to a numpy array.

Next, since the network had to be trained, I made a function called *split_data*. This took the data, and their associated labels and allows the user to determine how much of the data should be used for training the model, and how much should be used for testing the model. To do this, the *train_test_split* class was imported, which would randomly split the data and their labels into a training and testing set, based on the train and test size parameters.

To train and test the model, rather than using two separate functions, I decided to use a function called *run_classifer*. This function used the training data and training data classification labels, the user defined FFANN structure as well as the type of activation and learning methods to us, along with the maximum number of iterations for the model to run once. Using the *sklearn.neural_network* module, I imported the *MLPClassifer* class. Using the MLPClassifier constructor, I chose to use the *logistic* function as the activation function. This is the most population activation to use on feed forward ANNs and part of the project design. On top of this, I also chose to use the *lbfgs* back propagation technique to make the model learn. For a full explanation of this learning method, please see Appendix (?).

As part of this constructor, the weights on the synapses of the network are initialised randomly when the model is created by default, therefore this did not have to be accounted for explicitly. To ensure that complex models which would have a high training accuracy were not favoured, I chose to

also use a regularisation parameter *L2*. A regularisation parameter is used to penalise networks that are more complex, as they can train networks accurately, but have tendency to over-fit the data, which means they will have poor regularisation, and in turn poor classification predictive accuracy.

The final parameter used for the MLPClassifier constructor was the *hidden_layer_sizes*. Based on the research conducted and as part of the design stage, I chose to make a configuration file, to allow a user to input their own ANN structure. This allows more poeple to easily use the model, without having to directly edit the code. For instructions on how to use the configuration file, please refer to Appendix(?) or to Configuration_file_README.txt.

As many of the crucial functions that were required to make the ANN were prebuilt as part of the Sklearn library, it was computationally efficient in terms of time to develop this model, to allow more time for testing and evaluating the model using the main dataset.

4.2 Developing the Genetic Program

To being with, a small subset of the dataset was used. Whilst developing this model, I worked on smaller problems with inputs and outputs I knew would work, as well as an optimal function which the smaller model could find. The reason this was done is so that whilst developing the smaller program, the structure would be developed correctly, and debugging functions through the development phase would be much easier. It also meant if the structure of the model was appropriate, then scaling the model to work with the main dataset would not cause any issues.

4.2.1 Generating and Handling the Population

The first stage in the development process was to create a population of functions based on the functional and terminal sets, and population size as set out in the design specification. To do this, I created a class called *GenMember*. The purpose of this class was to handle a population by generating all the members, finding their fitnesses, selecting the parents and updating the population. By encapsulating the functions within a class, this helped to improve the structure of the model.

The first function, *generate_expression* was used to generate random mathematical functions in infix notation using recursion. Recursion allowed multiple sub expressions to be created which could be concatenated together to build full, valid functions. As the size of the function increased, the longer it took to evaluate and manipulate the functions, therefore to ensure that that the function stayed a reasonable length, a *max_depth* parameter was imposed. Using the project design for the population generation, I used the *full method*[**?**] of function generation to provide sufficient variety in the size of each of the functions produced, based on the maximum depth limit. This allowed the functions made to be more organic. Unfortunately, since not functions contained all 5 KPI ratios, I created function - *get_valid_expressions* which filtered out and replaced any invalid functions with valid ones.
The next step was to create a fitness function to evaluate each of the members of the population. The *get_fitness* function is called in two scenarios - when the population is initially created and when a new child is produced. When the population is initially generated, all the individuals need to be evaluated to find the fitness's of all the solutions as an optimal expression may exist within the original population. To evaluate each member, each row of the sample data is fed into the population of expressions, and the total was found. A negative value was classified as 0, otherwise 1. Using this, the actual classification labels from the dataset were then compared to the predicted classification labels. For every incorrect prediction, the fitness error value would increase by 1. Very poor solutions would have a very high fitness value, whereas very good individuals with strong predictive accuracy had a very low fitness value as the aim is to minimise the error, therefore they had a high predictive accuracy.

Using the evaluated population, the parents needed to be selected. To accomplish this, I made the *tournament_selection* function. The output of this function would be the two individuals selected to produce children through crossover and mutation. The population and their associated fitnesses were inputted, as well as a user defined tournament selection size. The size refers to the number of candidates that would be selected from the population, as potential parents. The two individuals that were the best from each tournament would then be the parents. The population and their associated fitnesses were zipped together to prevent a member obtaining the wrong fitness value associated with it. Based on the *selection_size* selected, *'c'* candidates were selected randomly from within the population. As two parents are needed for the process of cross over to occur, two tournaments were completed to select both parents.

4.2.2 Infix to Prefix Conversion

$$parents = [(\text{“}X1 + 1.3 - X3 + X2/X4 * X5\text{”}, 170), (\text{“}X2/X5 + 6.433 - X1 * X3 * X4\text{”}, 243)] \quad (2)$$

$$parents = [([\text{“}+\text{”}, \text{“}X1\text{”}, \text{“}-\text{”}, \text{“}1.3\text{”}, \text{“}+\text{”}, \text{“}X3\text{”}, \text{“}/\text{”}, \text{“}X2\text{”}, \text{“}*\text{”}, \text{“}X4\text{”}, \text{“}X5\text{”}], 170), ([...], 243)] \quad (3)$$

The parents could now be put into a binary tree structure. To do this, I first converted the infix expressions into prefix expressions (also known as *Polish* notation). This format is easier for the machine to parse into a tree structure. For this, I used a class called *ToPrefixParser*. Each parent was split into their individual elements along with the string 'stop'. This allowed the conversion from infix to prefix to never go beyond the number of elements in each of the parent lists. To ensure that each component of the parent was split correctly, I used regular expressions to split the strings.

Since the parents were split correctly, they could now be converted into prefix notation (eq. 3). Using [insert website name here], I manipulated a subset of their functions to convert the infix notation into prefix notation as this website was not enough by itself. The *get_operation* function is used to compare the *0th* element of the expression that is being passed in to the *expected character* that the parser is looking for. If they matched, then the value was popped out of the original expression. The *expected character*was checked throughout the other parsing functions. The *is_number* function is used to return a number which is a leaf value. However, if the character being checked is a "(" and the expected character is also a "(", then this will call the *get_expression* function which in turn would be used to get the subexpression that exists within the pair of parentheses. When *get_expression* is called, a chain of other functions is called sequentially, which will put the infix notation into prefix notation. *get_expression* calls the *get_product* function, which then calls the *is_number* function which checks to see if the current character being checked is a "(", in which case the sub-expression is then evaluated, otherwise a number is added to the prefix notation output. As each character in the infix notation is checked, these functions are called to correctly convert the infix notation into prefix notation.
After both parents have been converted from infix to prefix notation, the newly converted parents are returned by the *get_prefix_notation* function.

4.2.3 The Node Object

Now the binary tree was ready to be built. A binary tree consists of two types of nodes; branch nodes and leaf nodes. Branch nodes can hold a value, and have "left" and "right" child references. Leaf nodes on the other hand can hold a value but have no child references. Both branch and leaf nodes have parent nodes which can also be references. The top of a binary tree is called the root node, which is a branch node, however it is unable to hold a reference to a parent as it the eldest generation. Multiple nodes that are connected form a tree. This can be seen in Figure 4.
Based on the structure of what a node can hold, I made a class - *Node*, which has four functions.

The first function is the constructor which holds the possible features that a node object could hold. These include the current node value, the possibility of holding references to a left and right child, the possibility of holding a reference to a parent node, and finally a nodeID field which assigns a unique number to each node as they are created such that two nodes with the same value in it are still unique and will be referenced to by their nodeID, rather than the value they possess. Each node is null until a value is assigned to it, which will be done when the tree is being constructed. The second function was the ability to add a child node. This function took a value and either inserted a child into the parents left branch or right branch depending on the current state of the "left" branch. Every time a new child is added, the child also holds a reference to its direct parent to indicate that the child is connected to that one specific parent.
The final two functions used gave a visual representation of exactly how a tree would look. For each level that the tree goes down, the child of a parent is indented accordingly. The root node will always be the top node to be printed out first, followed by the its children.

4.2.4 The Tree Object

The tree representation of each parent could now be built using the prefix expression representation of each parent. The Tree constructor contained a root node, as this is how all trees need to begin. I created a function called *make_tree*. To initialise the tree with a root node, the first index of the prefix expression was removed and assigned to be the root node. To track the nodes, I used a *current_node* variable which was initially set to the root node. Since this node has now been created, the value was removed from the parent prefix list as it is no longer needed. Based on the current node selected, a check to see whether the value was within the functional set of values. If the current node was a functional operator, it would need to have exactly two children, a left and a right child. Therefore, I perform to check if the left child for that node has been created. If it is empty, then the next value in the prefix list is inserted into the left child and the left child becomes the new *current_node*. If the current node is not a functional operator, i.e. it is a leaf node from the terminal set of values, then this cannot have any more children. In turn, go back up to the leaf node parent and check to see if the right node can be created. If it can, then continue construction of the tree via the right branch. Otherwise keep going back up the tree to the root node and then start building the tree to root node right child to build the rest of the tree. After each value in the prefix list has been assigned to a node and the tree is completely built, the length of the prefix will now be 0. As a result, the tree has now been built using a depth first creation method, so the root node can now be returned.

The reason only the root node must be returned is because the root node has access to all its children which is the full tree, so every node can be accessed from the root node. A list of nodes is also returned to return all the subtrees of the tree as this will allow accessing the nodes for crossover and mutation to be easier. Finally, another variable *nodenums* is returned which is used to track the nodeID of each node in each tree. As each node is created, it is assigned an ID value. I made the nodeID variable static such that regardless of what tree is made, the value of the ID will always increment by 1. The reason for this was because when performing crossover and mutations on the tree, I would not have to reassign the node ID's to the new trees, they could keep their original nodeID's as each node ID is unique.

4.2.5 Crossover and Mutation

Through the process of crossover and mutation, two new children would be created, containing certain characteristics of both parents. Since I wanted to keep the parents in the population, and not directly replace them, before starting this process, I used a *deep clone* function to clone both the parents. The

reason for this is that the parent clones would be completely independent entities from the original parents. Using these parent clones, the children could then be produced through crossover and mutation.

I chose to complete a subtree crossover based on the design stage. Here, three functions were created; firstly, *select_random_val*.Using the list of nodes that were created when a parent tree was made, I selected a random subtree from the list of possible nodes . This would mean that the node selected and its entire subtree would be encapsulated within this. When selecting the node at random, I removed the root from this selection, as every node is a subtree of the root node, which means if crossing over occurred, then the full tree would be crossed over and this was prone to errors. This function returned the node that had been selected as well as the node's unique ID.

After the node had been selected, to ensure that the subtree did in fact did exist in the tree - *find_subtree* was used to locate the subtree within the parent. This function took three parameters. The first one being the tree itself as this is what would be searched and secondly the list of nodes as this is where the random value had been selected from and the random subtree would be selected from this. Finally, the last parameter to this function was the random node ID that had been selected. To search for the subtree within the parent, I chose to implement a recursive depth first search. If the depth first function located the node value, it would then also check the node ID to ensure that this was the correct node to be selected. Checking the nodeID was necessary to ensure that the correct node was selected as there was a possibility of two different nodes having the same value. However, as nodeID's were unique, there could only be one of each. If the subtree was found , it was returned. If the node chosen was not in the list, then an error is thrown to indicate that the depth first search was unable to find such a node with that specific node ID. This should never happen as for crossover to work, the node selected from the list of nodes needs to be within the parent tree this subtree of the parent will be crossed over with the other parent.

Next, another function called *swap_nodes* was made. The purpose of this function was to simulate genetic crossover between the two parents as this would create the children. This function took 4 parameters, including both the parent (clone) trees - which would be updated during the process of the crossover, and the nodes selected for crossover. This function first gets the nodes selected parent node and performs a check to see whether the parent's child is its left or right child, which is the node to be crossed over. By doing this, it is ensured that the correct subtrees are being crossed over with each other. Once the nodes have swapped over, the first parent now contains the second parent subtree and vice versa.

Once the crossover was performed, the 'parent clones' were now considered to be the children of the original parents as each new tree contained characteristic from both parents. As crossovers are more likely to cause bigger changes when searching the search space for an optimal function, it was recommended that crossover does not occur during every single epoch. Therefore, I used a crossover rate which can be defined by the user as a value between 0 and 1, where the user can select how frequently they want crossover to occur. After the new children have been made, they move onto the next stage which is mutation. When the cross over rate is low, there is a greater likelihood of crossover not occurring. When crossover does not occur, the parent clones remain exactly the same and progress to the next stage of mutation.

Although the children trees were built, the list of nodes that they each possess had not been created. Before children could be used for mutation, which uses the list of child nodes, the children had to use

a list of nodes that they could each have as well, similar to their parents. A function - *make_list_nodes* was created, to use the child tree and build a list of nodes which could then be used for mutation.

The process of mutation could now begin. Like the process of crossover, I used the *select_random_value* function to select a random node that would be selected for the mutation. After this node was selected, the node was then found using the *find_subtree* function. Next, the *mutate_node* function was created. This function considered the tree to be mutated, the list of nodes it held , the node to be mutated and the new fitness of the child. This function checks to see what type of value has been selected to be mutated, i.e. whether it is a functional node, a variable node or a numerical value. If the value is from the functional set, then it will randomly select another value from the functional set as they all the same arity. However, if the value is from the terminal set, then another check is performed to see if the value is a variable X1,..., X5 or a numerical value. If the value is a numerical value, based on the fitness of the child, the mutation will cause the node to be altered by a certain amount. However, if the selected node is a variable, then it will randomly select another variable instead. Whichever value is mutated automatically updates the tree and updates the list of nodes accordingly to ensure that the newly created child is consistent with all the changes that have occurred. Similar to the crossover rate, mutations are not meant to necessarily occur during every epoch, therefore a mutation rate was used, which again, will be defined by the user, however by default is set to 0.1 such that mutations would only occur 10% of the time.

4.2.6 Prefix to Infix Conversion

Once the children have been made, they need to be evaluated again and revert to a state that the population would accept, i.e. the child function in infix notation. This meant making a class *ToInfixParser*.

ToInfixParser contained two methods, which would convert the trees into infix notation. *deconstruct_tree* took the nodes from the child list of nodes, and extracted the values from the list of nodes, to return it back into the prefix notation state.

conv_infix took the prefix list, and goes through each of the values in the prefix list backwards, starting at index -1. For every value that is not an operator, it adds the value to the stack. If an operator is encountered, it pops the values currently into the stack and puts the current sub- expression into a bracketed form - '(⟨operand1⟩⟨operator⟩⟨operand2⟩)'. This starts to build the full expression back the infix notation, which is now in a state that the original population will accept.

4.2.7 Updating The population

As the new children that have been created have been converted back into their infix notation, the next step was to find the children fitness's, using the same *get_fitness* function. However, since this is a child expression, when *get_fitness* is called, the default *child* Boolean variable changes to true, such that the child expression is parsed into an acceptable format that will allow *get_fitness* to return the fitness in the same state as if the population was originally evaluated. The reason that only the children had to be evaluated is that the rest of the population had already been evaluated and did not to be re-evaluated as this would waste computation time.

Once the child fitness's have been found for the children, the population can be updated accordingly. To do this, the function *update_population* was called. This function takes the original population and the new child, and compares the new child against the worst member in the current population. If the child is better than the current worst member of the population, then the child will replace this member in the population. This process is repeated for the second child as well, to ensure that the

worst members of the population are filtered out, and to enable the GP to start learning, but accepting better fitted individuals and rejected those that are worse. By replacing one individual at a time, this also help to make sure that the population size remained the same size as when it started.

4.2.8 Termination and Running

After implementing the classes and subroutines based on the design specification and initial project specification, the had now been built. The final step of development was to create the main function to put all the classes and functions together into the main run function where the model could be executed at once. For this, a *train_gp* function was created, which is where the model would learn, to produce the optimal function to predict CB. Since the *train_gp* function is where the model would learn, this function contained multiple parameters. The reason for this is that when testing the code, rather than having to search for every parameter that can be changed, I would only have to change the value in the definition of the function.

Within the *train_gp* function, the initial population is generated exactly once, and enters a while loop. The significance of this loop realtes to one of the termination criteria - *max_iteration*. The model starts by evaluating the population exactly once, and then checks to see if an optimal solution already exists in the original population. If the original fitness is smaller than or equal to 100, then this has met the termination criteria as the this function would have at least 80% training accuracy. The second termination criteria is met if the user defined maximum number of iterations are met.

Once the termination has been met, the optimal function is then returned and this function is used to test the predictive accuracy of hold out data to see if it is able to classify the hold out sample data correctly.

After executing this model multiple times, *optFunc* and *optFunc2* show examples of optimal functions that were produced, which gave 75% and % predictive accuracies on the hold out sets respectively.

$$optFunc1 = 35.400318494530815 + (X3 - X2 - (673.1926 * X2 * X4^4 * X5^2 + 8.7779947854) - 25.99600428294065 * X4^3 * X5) = 75\% accuracy$$

$$optFunc2 = X3 * (X3 + (((X4 * (X2/((X4 * (3X4 - X1 - X3)) - X2))) - X2) - X1 + X4)) - X2 - X5 - X1 + X4 = 76\% accuracy$$

Now that this was complete, the GP had been successfully designed and implemented, following the design specification. Now that both models were created, the next stage of the waterfall development was to test the code, which will be discussed in section 5.

5 Testing, Results and Comparisons

5.1 Testing and Results

I used various methods to test whether the code was performing the way it should, and to see if different techniques affected the training and testing predictive accuracy of both the models. For both classification models, I created a testing plan which was split into two sections for both models. The first section of the plan related to *software testing*, to check if each function was performing as it should, which would lead to the correct outputs. The second section of both plans related to testing whether or not the models made were classifying the data correctly and to change various parameters to see

if this affected the training and testing predictive accuracies. Within this section, I describe some of the methods I used to perform different types of testing such as black box, unit, performance, and user acceptance testing. I then describe some of the experiments that were completed, and parameter tuning used to get optimal predictive accuracies.

5.1.1 Code functionality & Parameter Tuning: Artificial Neural Network

Since the feedforward ANN was created using *Sklearn* - a well known library for machine learning - it did not seem necessary to perform software testing to determine if the *MLPClassifer* was functioning correctly. The reason for this that through reading the Sklearn documentation for the MLPClassifier, the class and its functions that are provided have already been thoroughly testing before being deployed to the public and made to be user friendly. That being said, it was possible to test the functions that I created to make sure that the data was being read in, the data being passed through functions like the *run_classifier* would ensure that the model was learning on the correct data. Although code testing did not have to be done for the *MLPClassifer* or *train_test_constructors* , there were still different ways that the parameters could be changed which would lead to different predictive accuracies.

After reading in the data, it was shuffled and split into two sets; a training set and a hold-out set. The data was shuffled to give all the rows an equal chance of being selected, regardless of their classification, as this could affect how the model learns. The training data was used to train the ANN model by altering the parameters of the synapses. The testing data was used to predict how accurate the model was on unknown hold out sample data. By doing this, it means that the network could train on one set of data, and the performance of the model can then be evaluated based on the hold out sample. This method is called *train_test_split* (TTS). The benefit of splitting the datasets is that the testing accuracy rewards simpler models, rather than overly complex ones since simpler models tend to generalise better. Not just this but splitting the data into two sets also helps to prevent the overfitting of the model. This tends to occur when the whole dataset is learnt perfectly when training the model due to memorisation, however when unknown data is fed into the model, it will perform very poorly as it has generalised very badly. Within the TTS constructor, I chose to use a (5, 5, 5, 1) structure to compute train the model on. This was based on the design and project specification that was originally set out. Within the TTS constructor, I chose to vary the training and testing set sizes, to determine how much data the network needed to train on to produce a solution that produce the best classification accuracy. This can be seen in Figure(?). The train test split method is known as a high variance estimator. What this means is that every time a new set of data is being used to train and test the data, since this is random, the network will train differently every time it is run as the observed values in the training set will differ each time, a major drawback to this technique. This issue can be seen through out the Figure 4 as the variance of using 10% and 90% of the data to train the model, the predictions had a range between 51 and 95.96 respectively. Another reason that a large variance may occur is because this is a *non-deterministic* model to begin with. Every time a new instance of the model is created, the weights on the synapses are initialised differently. As a result, different areas of the search space will be explored using a learning algorithm to try to find an optimal solution. Out of the different sized data sets used, using 80% of the dataset to train the model and 20% to test the model was the best one to use. The reason for this is that on average, it produced a higher classification accuracy on the testing datasets, but also had a smaller variance and standard deviation than when using 90% of the data for training, although this was marginal (90.24 and 95.96 variances respectively).

One way that the problem of high variance can be overcome is by using another method - *K-Fold Cross-Validation*. K-Fold Cross validation is the idea that the train test split method is run several

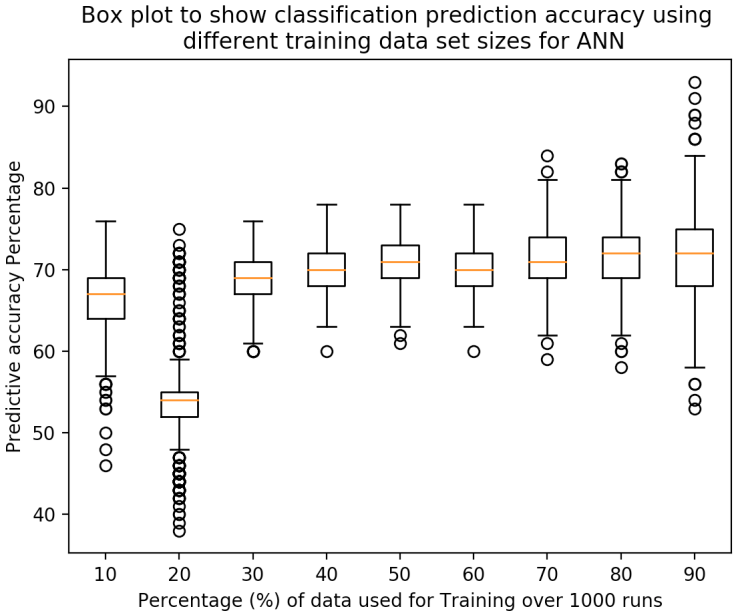

Figure 4: Left graph shows the use of train test split to get the classification accuracy of the ANN. Right graph shows the use of K-Fold cross validation to get the classification accuracy of the ANN

times, using different train test splits each time, calculating the testing accuracy each time and then taking an average of all the results to get an average overall prediction, which in turn would also reduce the variance. For this to work, the data is divided into K partitions, where one set would be used to train the model , and the rest of the data is used to validate the model, calculating the predictive accuracy each time. This process is repeated several times, until an average of the validation accuracies of the model is obtained. This average is the cross validated out of sample accuracy. The benefit of this method is that here, every data point is used during the training phase and also during the testing phase. To implement this, I used the *cross_val_score* class, with 10 fold cross validation. To improve upon the original ANN, as seen in the development stage, I decided to use 2 hidden layers with 5 nodes, however this may not necessarily be most optimal number of layers and the number of nodes for each layer. Therefore to find an optimised number of nodes for a simple ANN, I put the MLP classifier within a for loop such that for each value of n, 10 fold cross validation was used, and the average taken after each fold for each value of 10. The range used for this was between 1 and 50. Looking at figure 3, are the outputs of running this process three times and plotting average accuracies over time, along with the optimal number of nodes for to produce the predictive accuracies. From these three iterations, it could be seen that when using one hidden layer, although more nodes were required, this still produced the best generalisation without memorising the data. The first graph shows that if more than 50 nodes were used for this layer, it may have in fact performed even better. On the other hand, it is difficult to say whether or not the hold out sample predictions would have improved if more nodes were inserted into each of the laters.

5.1.2 Software Testing: Genetic Program

I originally generated a very small set of static data with classifications, with an optimal function that could be used to verify whether the GP was learning or not when starting GP development. GenMember was used to create random valif mathematical functions, select parents and update the population. Unfortunately, as it was very difficult to test whether *generate_expression* was working, I

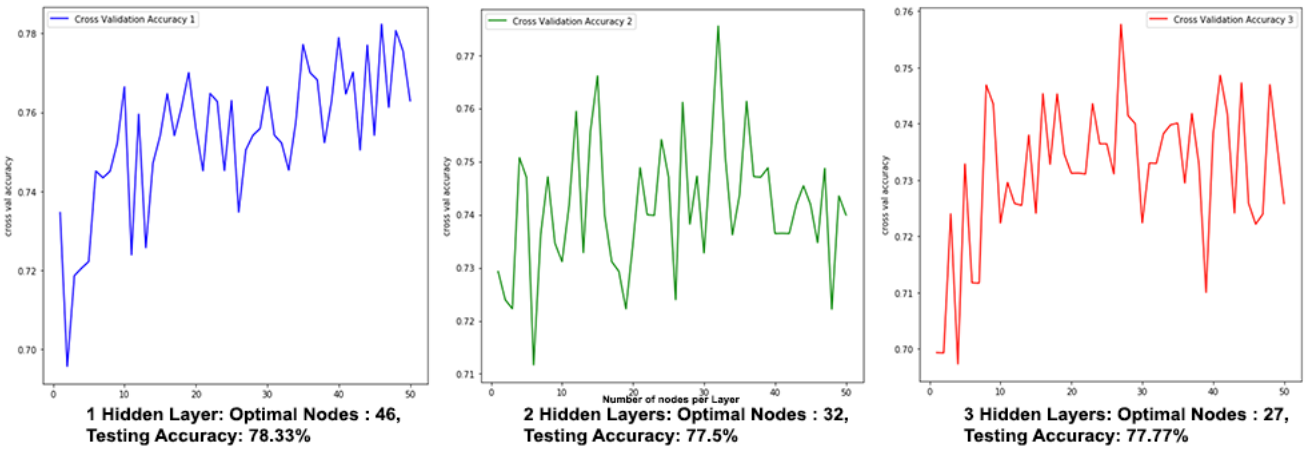

Figure 5: Output of predictive accuracy for 1,2 and 3 layer ANN's

had to manually verify that this function was producing valid mathematical functions, with complete bracketing in the correct places. To ensure that the *get_valid_expression* worked, since the generated expression needed to contain all five variables X1,...,X5, the only way that this condition could be checked was to call the function multiple times based on the generation created originally. Using these outputs, which I was able to acknowledge that they were correct, I created a small population of 6 individuals. Based on the fitness values given to each of the individuals in the population, I could perform Unit testing on functions such as *tournament_selection*, as I knew the sample population. For this function, I could use white box testing within the unit test, to ensure that out of these 6 individuals, 4 were candidates and the best individuals were assigned to be parents. This was tested multiple times to ensure that the best candidates were always being selected.

The next class to test was *ToPrefixParser*. To enable the infix to prefix conversion to be successful, the state of the infix function had to be changed into an acceptable format for *get_prefix_notation* to accept. As this code was used from another source, to initially test this on sample problems, I wrote out infix functions and manually converted them to prefix notation. Using this, I used black box testing to check whether the infix notation being inserted was correctly being converted to prefix notation, on small and larger expressions, to ensure that every expression was being correctly converted.

Using two expressions that were converted into prefix, I was then able to start testing the *Tree* class. For *make_tree*, I used a simple prefix expression and manually constructed what the tree should look like. If the expected tree followed the same structure as the manually drawn tree, i.e. nodes placed in the correct place, and correct associations with their respective parents, then the *make_tree* function was functioning correctly. This same process was repeated for larger, more complex prefix expressions, to ensure that this function could build any tree, based on any variety of prefix expression.

To ensure that *find_subtree* was performing as it should, rather than trying to find a random nodeID, I chose a specific node to find, such that I was able to perform the search manually and in a 'verbose' mode, such that as the function was running, I could confirm that at every stage, it was reaching the expected node it was meant to.

Another crucial function that needed testing was *swap_nodes*. To test this, I used two prefix expressions which simulated parents in their binary tree structures. By inserting and printing the parents that would be crossed over, it was obvious to see that the nodes had been changed before and after the crossover took place. To ensure that this was correct, white box testing was performed to determine that the new children produced were contained in the right variables and contained the correct

node values. This was relatively straightforward to test as I could print the new trees produced and compare the original parents to the children, based on the child nodes that I had selected to be crossed over. This same process was performed for the *mutate_node* function, as they both required the tree inputs, and the node(s) to be manipulated, therefore since I selected a specific node to be mutated, I was able to see exactly what it was mutated to, to ensure that this function was performing as expected.

The next class that needed testing was the *ToInfixParser* class. To test this class, using a simple problem to begin with, I used a prefix notation expression, and manually converted this to infix notation. Using this as the desired output, I could use black box testing to determine if that prefix expression was converted to the same infix expression in the*conv_inf* function . This was repeated multiple times on small and then bigger problems. To ensure that this was correct, I had to evaluate the newly created infix expressions to ensure that due to the encapsulation of the sub expressions, the numerical output remained the same. If they did, then this test was successful.

After the children were evaluated, I used white box testing to confirm that the children were in the correct string format to be accepted back into the population. Through this white box testing, I was able to see where the children were checked against the worst members of the population, and then see which members of the population were changed, and to see whether the population sized definitely stayed the same. If the worst members of the population were removed, then this confirmed that the model was starting to learn as it was filtering out worse members of the population.

Finally, the last section to test was the termination criteria. using a sample population, I was able to track the progress of the population through a few iterations, and kept the termination criteria very flexible, to check that if certain conditions were met, the program would terminate successfully with the correct function being returned as part of the output.

5.1.3 Running the Genetic Program - Training and Testing datasets

Similar to the ANN, in GPs, it is very common for the network to train using one dataset and to test the predictive classification accuracy on another dataset. During the development of the project, after simple problems were able to produce the correct results, I used a random selection of 10 rows of data from the original dataset which could be used to train the network as a sample, as a validation method to see if the accuracy of training prediction increased over time. If after n iterations, the optimal solution was found for example with 90% training accuracy, this confirmed that the model was learning the data over time, to produce an optimal function. Using this confirmation, I was then able to feed in the full dataset to see if the model would still learn.
My first method of testing was to train the GP using the full given data set. After this was done, since another dataset was unavailable, I used the same dataset to test the predictive accuracy of the genetic program optimal equations. Even thought it was thought that this would in theory achieve 100% predictive accuracy, the GP produced between 70-75% accuracy. The reason for this is every time a new population is generated, a different area of the search space is explored, which means the function, albeit have the same training and testing data, it will not produce 100% testing predictive accuracy.

To make the the ANN and GP more comparable, I also attempted to use the train test split function to split the data into two sets, using 80% of the data to train the model and 20% to test the model. The outputs of this can be seen in Figure below [insert graphs of GP and ANN when using TTS].

Although K-Fold Cross Validation can be used for the GP by being able to produce the average predictive accuracy of the model, the aim of the project was to find an optimal function that could be used to predict CB. However it is not possible to average multiple expressions which would produce

a solution that is equivalent to the predictive classification accuracy produced by K-Fold. Therefore I decided not to use this method as a form of training and testing the datasets.

5.1.4 Genetic Program Experiments

As an optimal function was being found for smaller problems, I used the actual dataset and altered the parameters to see if the predictive accuracy would increase on more complex problems. To make this process easier, the *train_gp* function contained all the parameters that could be optimised, which could affect the predictive accuracy of the model. For every test that was conducted, each test ran for a minimum of 50 times to ensure that each of the tests were giving consistent results, and to identify any patterns and anomalies that may exist when running various tests.

The first parameter that was changed when testing on the hold out data set was the threshold value to classify a company as 1 or 0. The values of this ranged between 0.1 - 0.9, to see the different classification accuracies when being used on the hold out set.Through the use of box and whisker plots, it was noted that the worst predictive accuracies came from threshold values that were closer to the the values or 0 or 1. When using threshold values of 0.1- 0.4, although the accuracies did improve, there was a significant overlap between all the accuracies. Due to their high variances and standard deviations, it was clear to see these thresholds gave very volatile predictive accuracies. Not just this but it could also be seen that the average predictive accuracy was increasing between threshold levels. As such, 0.5 and 0.6 threshold values were then tested. Using the data from both thresholds, the mean, variance and standard deviation were calculated from both to determine their consistency. The variance of 0.5 and 0.6 was 0.0021 and 0.0023 respectively. Their standard deviations were 0.045 and 0.048 respectively. Looking at the graphs and these numbers, both show that the predictive accuracies using both thresholds remained relatively consistent and that the spread of data remained relatively confined, around the mean, however 0.5 was marginally narrower than 0.6. The next test that was

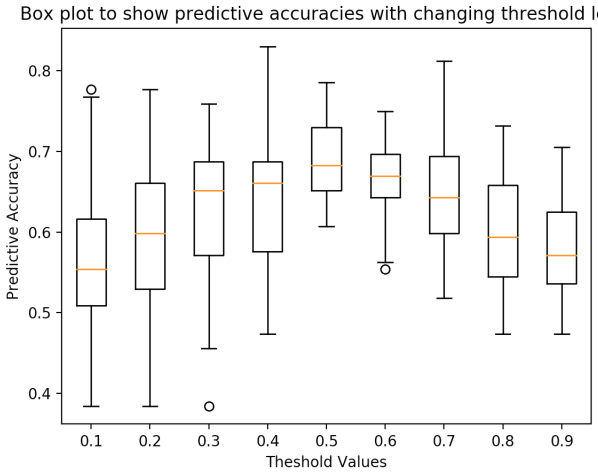

Figure 6: Box and Whisker Plot to show different testing threshold values and their classification accuracies over 50 runs

performed was using different selection methods. As the selection function is also a parameter, this increased the flexibility of the model, as I was able to create another selection mechanism, select best to train and test the model. This selection function selected the best two individuals in the population every single time. The expectation of this is that when training the model, it would plateau off as the

fitness values would start to converge to a fitness value similar to the parents, therefore converging at a local optimal solution. This did occur as can be seen in figure(?) which compares the fitness errors of using tournament selection against select best. [insert figure here].

Another parameter that could be tuned was the maximum number of iterations allowed before the model terminates. This involved using a variety of different max iteration values, to see if this made any difference to the predictive accuracies. In theory, if there were too few iterations, then the model would not be able to learn sufficiently. If there were too many iterations, then there is a possibility that learning would eventually stop, and that this would be very computationally expensive to try to find an optimal solution to this problem. The results of this can be seen in Figure (?) where when using 10 and 100 as the parameter values, the graph clearly indicated that learning was still occurring, however, after when the maximum iterations were 1000, 2000, 3000 , they showed signs of convergence, and eventually stopped learning as quickly. [insert figure here]. However, when using over 1000 members in the population, this showed signs of poor predictive accuracy.

5.2 Comparisons of Artificial Neural Networks and Genetic Programs

Using the two models, these could both be tested individually as above, against each other and against other bench mark tests such as the predictive accuracies that Altman was able to achieve. Through testing both models that were made, I tuned the parameters such that the optimal parameters that gave the best results from each model could be compared.

6 Critical assessment and Reflection

The aim of the project was to create and compare two models to determine if a company was likely to go bankrupt or not. Overall, both of the models were successful in being able to do this, however both gave different, but comparable results. There are multiple factors that may have affected the overall classification accuracy for both models, which will be discussed here.

Firstly, as part of the project specification, I chose to implement a feed-forward ANN. Since this model was used as a benchmark, being able to create the model was relatively uncomplicated and successful. The reason for this is because to execute the ANN model, there were only a few crucial components required to construct it, fit the data to the model, and use the trained model to test the predictive accuracy. In turn, this allowed me to exploit the library functionality to enable tests to be performed on this model. This included running the model once on a small static data set to determine if the model was in fact learning, before using the full data set to run the classification model. Once this was achieved, using the *cross_val_score* functionality, I was then able to perform split the data into training and testing datasets using 10-Fold cross validation find the optimal number of nodes that would produce the best predictive accuracies. Through the testing completed, I was able to successfully return the optimal number of layers, and the optimal number of nodes required for all the layers. Through the 10-Fold cross validation, the optimal average predictive accuracy was also able to determined.

Due to the variety of parameters that could be applied to the *MLPClassifer*, I was able to exceed the expectations of the ANN by being able to apply different activation functions and different learning functions that could be combined together to find the most appropriate methods used to make the ANN successful. This was not part of the project or the design specification, however, as these facilities were available, I was able to experiment with these. Using these enabled more experimentation to be performed than initially planned as these functions were already pre-built functions.

As part of the ANN model, even though it was not possible to produce an optimal function that a user could use, through the testing completed, I was able to compare the predictive accuracies of the ANN when a different number of ratios were inputted into the network. This was able to determine significance of each KPI, which was part of the original specification. Finally, by passing the outputs through a logistic function, it was able to produce a probability to represent the likelihood of failure.

Overall, after the ANN model was researched, designed and developed, I believe that this model was a good solution to this problem, and could be used to classify whether or not a company is likely to go bankrupt within one year.

As part of the project specification, I chose to implement a genetic program, which would be able to produce an optimal function to predict corporate bankruptcy. Since this was completed without the use of any machine learning libraries, the development of this was more complicated, however by the end of the project, I was able to successfully produce an optimal function which was the main aim of the GP to approach this problem, as can be seen in section (?)

As the two example functions produced show, the X1 KPI - *working capital / Total Assets* had been filtered out of *optFunc1*, which proved to show that this variable had no effect on the predictive accuracy of the model, and was not needed when random constants were used. However all the KPIs were required when random constant variables were not included. This met the project specification by showing which KPIs were crucial in all environments and which ones were not.

Although I intended on using K-Fold cross validation for the GP in the project specification, I was unable to meet this requirement. The reason for this is because this technique finds an average of multiple runs to give the best average prediction accuracy. Whilst this is possible, it is not possible to

find an *average* function based on the optimal functions produced. In turn, I decided to not use this technique to train and test the model.

7 Future Work

Due to time restrictions, for this project, there were several other features that each model could have used, which could have improved the development of each model but also in terms of comparisons.

Firstly, the ANN was built using a library, rather than from scratch, whereas the GP was built without the use of optimised libraries to help give better results. To make the the models more comparable, the ANN could be created without the use of a library such as *Sklearn*. Secondly an optimised library such as *PyEvolve* or *DEAP* could be used, to develop the genetic program. The reason that I have suggested this, is that more models could then be compared, against 2 ANNs and 2 GP, and can compare their accuracies, efficiencies, and the possible causes for this.

Secondly, when developing the genetic program, if there was more time, a maximum tree depth would be enforced more strictly. Through crossover and mutation, the trees that are built can start to grow very quickly, especially when the crossover node has been selected higher up in the tree. As Figure (?) shows, this also means that the larger the tree, the more computationally expensive it is to evaluate a larger tree. By imposing a maximum tree depth on any trees created, this would have several benefits. Firstly, it would reduce the computation time significantly as only children that fit within this maximum depth would be produced. In turn, the solutions produced would would also be much smaller, but just as effective and could produce an optimal result. This also means that when training a model, the maximum iteration counter could be significantly larger, as this would not affect the computation time at all, as all the solutions would be much smaller. Another factor that could be avoided in the future would be to not include a function that requires the building of a list of nodes to store the trees. To select a random node, rather that selecting from a list, this could be done directly via the tree itself. which would require less computation than building the tree, a list of nodes and selecting then selecting a node from the list of nodes. This in turn would help to improve the performance of the model.

Next, when the optimal function produced is returned, the function produced could be pruned. Currently, the expression may very large, which may not be straightforward to read. One way that this could be fixed in the future would be to prune the optimal expression, which evaluate any sub-expressions, and therefore cut down the total size of the function, whilst maintaining its original accuracy. This would make for a much better formulated function which would be much easier for a user to and use.

Whilst testing the GP, only one type of crossover and mutation was implemented. Although this was sufficient, to allow more testing to occur, other crossover methods could be implemented such as *size-fair crossover* could be used, which would help control the tree size by proportionally selecting a random subtree for parent one, but the same sized subtree for parent two. For mutations, a *hoist mutation* or a *shrink mutation* could be implemented. These could then be used with the crossover methods to help create more tests, which may be far better than just using the standard node replacement mutation that was used.

Finally, for the datasets used, they were only American companies that were considered. In the future, data from other economies could be used, for example, from the British economy. This would provide score for more testing to be completed, to see if the current models could predict as accurately for companies from other countries. By having data from other economies, it could also potentially be used to make GP that could be applied universally, such that any company from around the world would be able to use the function to get a prediction and a classification.

Overall, if there was more time, many features could be inserted or adapted to make the development and usability of the genetic program potentially better, and more accurate.

8 Conclusion