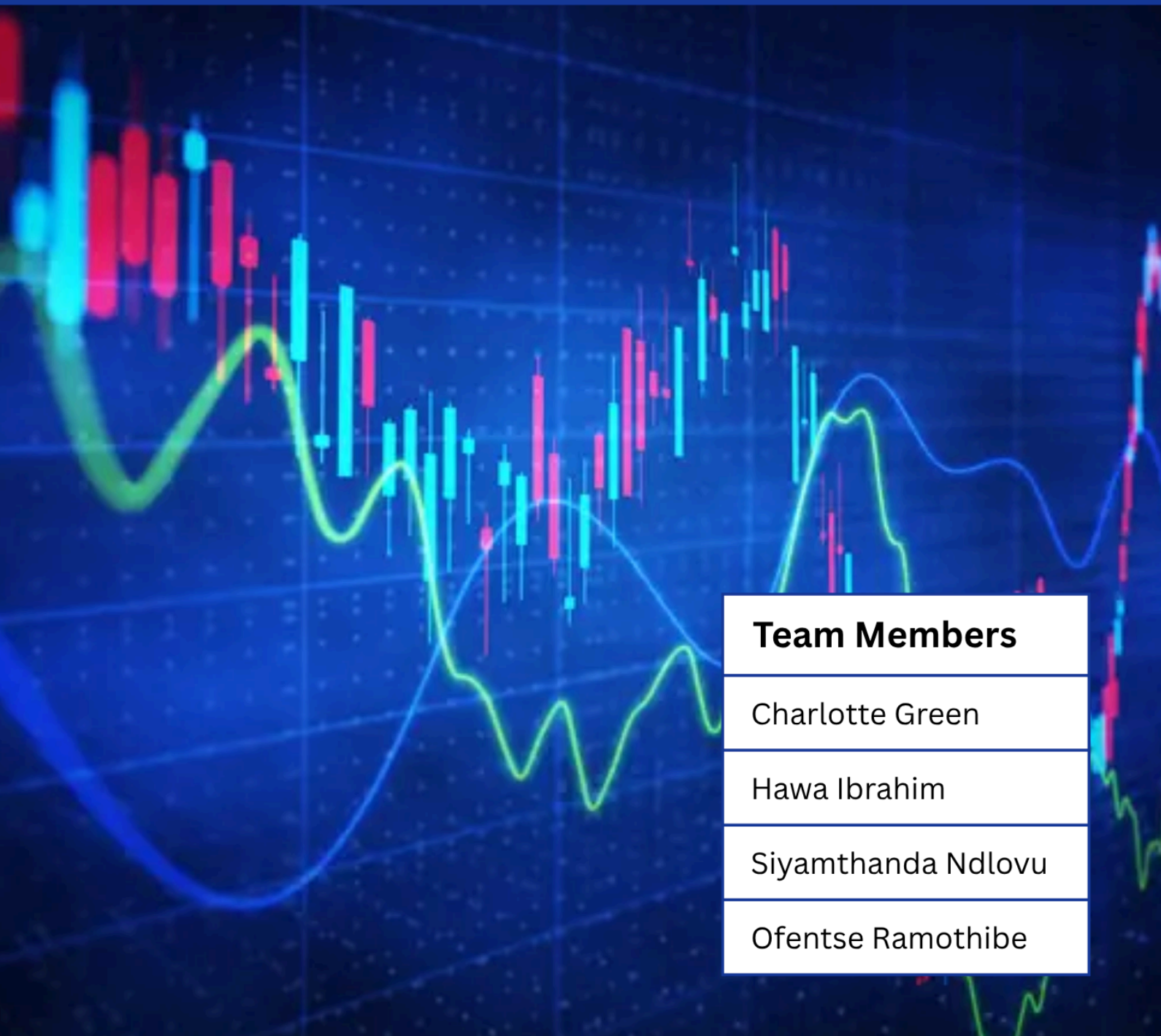
Forecasting Stock Purchases

Using Multi-Layer Perceptrons, Genetic Programming and Decision Trees



Team Members
Charlotte Green
Hawa Ibrahim
Siyamthanda Ndlovu
Ofentse Ramothibe

Contents Menu

1. Introduction

Problem Overview

Financial forecasting refers to the use of historical stock market information in order to predict the future financial performance of a company or asset. It explores previous trends and analytics in order to estimate revenues, expenses, profits, investment signals and stock price movements.

Machine Learning as a Tool in Financial Forecasting

Machine learning is a form of artificial intelligence that allows computers to make predictions using learned patterns from data without explicit programming. These systems start with a training set of data and are improved over time as more data is input and the systems can adjust accordingly.

Machine learning models are frameworks of machine learning that help guide the solving of a given problem in a particular way. This assignment looks at the use of three of these models (Genetic Programming, Multi-Layer Perceptron and Decision Tree) in order to tackle the financial forecasting problem of predicting whether or not a financial stock should be purchased based on historical data.

By making use of the aforementioned models, the technical indicators of opening and closing prices, daily high and low and adjusted close can be analysed to form profitable buying or selling decisions. These predictions generate a binary prediction of whether or not to buy in the form of 1 or 0, where 1 refers to a ‘buy’ signal and 0 to a ‘do not buy’ signal.

Statistical Analysis

To analyse the results of the given problem, the findings from the algorithms mentioned will be drawn up in table sets and summaries will be made from each table respectively. The Wilcoxon signed-rank test will be carried out between the Genetic Programming and Multi-Layer Perceptron algorithms to evaluate the difference in their performances.

Data Description

The dataset contains historical stock data with the following attributes:

Attribute	Description
`Open`	Stock price at the start of the day
`High`	Highest stock price during the day
`Low`	Lowest stock price during the day
`Close`	Price at market close
`Adj Close`	Price adjusted for dividends/splits
`Output`	Binary label (1 = Buy, 0 = Don’t Buy)

2. Machine Learning Models

2.1 Genetic Programming (GP)

Overview

Genetic Programming forms an extension of the Genetic Algorithm in that it is an Evolutionary Algorithm made up of individuals with the aim of evolving by use of a fitness function and through population evolution until a near-optimal solution is reached. In the context of financial forecasting, Genetic Programming is used to discover mathematical expressions that can accurately predict whether a stock should be purchased or not, based on historical market indicators.

Each individual in the population represents a candidate decision-making program, structured as a syntax tree that is composed of mathematical operators and financial input features (such as Open, High, Low and Close prices). These individuals are initially generated at random and are refined over time using genetic operators such as crossover and mutation.

To build these individuals, Genetic Programming relies on two key components:

- A function set: defines the mathematical operations available (+, -, *, /).
- A terminal set: the input variables (stock price features) and random constants.

The above two sets must satisfy two principles:

- Closure: all operations produce valid outputs for future operations
- Sufficiency: they are capable of representing a solution to the problem

The resulting syntax trees are converted into executable expressions used to classify data points as either Buy (1) or Don’t Buy (0).

Through multiple generations, Genetic Programming selects, evaluates, and evolves these symbolic models using a fitness function based on classification accuracy. The goal is to produce an optimal or near-optimal expression that reliably signals buying opportunities in the financial market.

Initial Solution Generation

The initial generation of individuals in Genetic Programming is generated at random. Each program, using the training or historical data, should classify whether or not a stock should be bought using values of 1 and 0 as aforementioned.

The maximum depth of a syntax tree created is to be user-defined along with the size of the population. To initialise the generation of the tree, one of three methods (namely full, grow and half and half) can be used.

In order to create a program, a root node is selected at random from the function set and the nodes thereafter are constructed by recursively building subtrees until the maximum depth is reached, where the leaf nodes are determined depending on the type of tree generation method used. Nodes at the maximum tree depth will always reflect as values from the terminal set.

Buying vs Selling

The purpose of the algorithm is to evolve the mathematical functions that are represented by the syntax trees that take input in the form of historical stock features such as Open, High, Low, Close and Adjacent Close prices. These input features will help the algorithm to determine whether or not to buy a stock as follows:

1. Each program is evaluated using a row of the input features
2. An individual processes the input values using the operators specified within the function set to produce a single numerical output
3. The numerical result is then interpreted using a classification rule as follows:

if (value > 0)
then
Buy (label = 1)
else
Don't Buy (label = 0)

4. The classification result is compared to the label in the dataset in order to determine the fitness of the program.

Fitness Function

The fitness returns as follows:

$$\frac{AmountOfCorrectNumericalClassificationPredictions}{TotalNumberOfClassificationResults}$$

Genetic Operators

In order to produce genetically diverse offspring to allow for more optimal solutions to be created, genetic operators such as mutation and crossover are applied.

In subtree crossover, two parent trees exchange genetic material by trading subtrees that are marked at a randomly selected crossover point. The offspring are pruned in order to conform to a maximum depth limit.

Within the mutation, a terminal node is randomly selected and replaced with a newly generated subtree, using grow mutation to increase the size of the tree and possibly diversify the decision patterns. The mutation depth parameter limits how large the newly inserted subtree can be.

The number of individuals produced by crossover and mutation is controlled by application rate parameters:

- Crossover Rate: Percentage of the population generated via crossover
- Mutation Rate: Percentage generated via mutation

These rates ensure that the newly evolved population maintains the same size as the original, while balancing exploration (mutation) and exploitation (crossover).

Stopping Criteria

- Maximum number of generations
- Perfect accuracy in buy/sell decisions

2.2 Multi-Layer Perceptron (MLP)

Overview

This is an implementation of a Multi-Layer Perceptron (MLP) with Backpropagation as the learning algorithm to predict whether to buy or sell a stock based on its previous price values. Our experimental design is guided by Jaiswal & Das (2017).

We designed a 3-layer architecture (input → hidden → output) and experimented with various hidden layer sizes (2,3,4,10). Two neurons for the hidden layer provided optimal results.

Seed and Random Value Initialisation

Each run of the model was tested with a different seed and number of neurons for the hidden layer. The seeds were used to randomly initialise the weights and bias. Initialisation of the weights and bias was limited to a range of small values (x) between

$$x \in [-\ 0.05,\ 0.05],\ where\ x\ \neq\ 0$$

Our implementation retries initialisation until a valid x value is assigned.

Justifications:

- Our data is normalised to a range of approximately [0.9, 1.7]
- Allowing the weights and bias to equal 0, results in gradient values, i.e. changes of 0 and the network essentially gets stuck, never starting and never learning anything.
- Large starting values result in network congestion because of our use of the sigmoid function. The sigmoid function responds to large values by outputting 1 or 0 more often, causing a saturation of 1 and 0 weight and bias values. Saturation of 1s and 0s causes the network to get stuck, unable to change while learning
- Small values, which sigmoid responds well to, return varying values like 0.3, 0.7, 0.2, etc., allowing the network to learn and change easily

(Popescu et al. (2009)).

Architecture

Input Layer and Features

Our input layer consists of 5 financial features extracted from Euro/USD stock data: Open, High, Low, Close, and Adj Close prices. This follows the established OHLC methodology used in financial prediction literature (Jaiswal & Das, 2017). The model predicts a binary output (1 = buy, 0 = do not buy) based on these normalized price indicators.

Hidden Layer

It has been shown that networks with one hidden layer can approximate any continuous function (Popescu et al., 2009) - this is the universal approximation theorem. Given this, we settled with one hidden layer, leaving us with the task of finding the right number of neurons for this hidden layer to configure the network.

The number of neurons used for the hidden layer varied; we used sizes {2, 3, 4, 10}. This mirrors Jaiswal & Das (2017) implementation and provides a range that includes sizes less than the number of inputs, equal to and greater than the number of inputs.

Output Layer and Logistic Sigmoid Function

Our output layer consists of one neuron, which yields 1 or 0. Our implementation uses the logistic sigmoid function, which converts each of the OHLC combinations into a final output value in the range [0, 1]. This range is interpreted as a final value of 1 or 0, which determines if the stock should be purchased or not.

$$f(x) = \frac{1}{1+e^{-x}}$$

$$if\ f(x) > 0.5,\ output = 1\ (buy\ EUR/USD)$$

$$if\ f(x) < 0.5,\ output = 0\ (do\ not\ buy\ EUR/USD)$$

Learning Algorithm: Backpropagation

The backpropagation learning algorithm learns by conducting a series of forward and backwards passes. Forward passes use the current weight and bias values and pass the values through the layers to produce an output value. Backwards passes involve calculating the errors for the output layer nodes, weights and biases as well as the hidden layer nodes, weights and biases.

The weights for the network are then updated, followed by the mean square error (MSE) calculation to assess the performance of the current weights. This completes one epoch. If the termination conditions are not met, another epoch is carried out till the termination conditions are met.

Forward Propagation

Our implementation starts with reading the training dataset **`(ReadFile.java)`** to capture the number of features, which sets the number of neurons in the input layer, the number of output columns to set the number of neurons for the output layer, followed by capturing each row of the dataset **`(Dataset.java)`**.

The network layers are created by the functions **`MLP.MLP()` → `Layer.Layer()`**. Each row of the training dataset is fed into the network, setting the input layer neuron values, i.e forward pass each row **(`MLP.forward()` → `Layer.forward()` → `Neuron.calculateOutput()`)**.
Each neuron of the current layer has the Neuron.calculateOutput() applied to it, which applies the following functions:

$$weighted\ sum\ for\ hidden\ layer\ neurons:\ n_1j\ =\ b_1j\ +\ \sum_{i=1}^{n}(w_{1_j}i\ *\ x_i)$$

$$weighted\ sum\ for\ output\ layer\ neurons:\ n_2k\ =\ b_2k\ +\ \sum_{i=1}^{m}(w_{2k}i\ *\ z_i)$$

$$f(x)\ =\ \frac{1}{1+e^{-weightedSum}}$$

$b_1k\ =\ hidden\ layer\ bias$

$b_2k\ =\ output\ layer\ bias$

$w_{1_j}i\ =\ weight\ from\ input\ layer\ neuron$

$w_{2k}i\ =\ weight\ from\ hidden\ layer\ neuron$

$x_i\ =\ input\ neuron\ feature\ value$

$z_i\ =\ hidden\ neuron\ output\ value$

Each layer serves as an 'input layer' to the next layer.

- Input layer: Raw features (Open, High, Low, Close, Adj Close)
- Hidden Layer: Layer.forward() applies weighted sums and sigmoid activation
- Output layer: Final prediction [0,1] to buy or sell

Backwards Propagation

The error for the current row is processed. The error assesses how well the current weight and bias values predicted the target value (**`Layer.calculateOutputLayerError()`** and **`Layer.calculateHiddenLayerError()`**).

Errors for the output layer neurons are calculated first, then backpropagated to the hidden layer

Output Layer Error Calculation:

$$\delta_k = (t_k - f(n_2k))\ *\ f'(n_2k) = (t_k - f(n_2k))\ *\ (f\ (n_2k)(1\ -\ f\ (n_2k)))$$

$t_k = target\ value$

$f(n_2k) = output\ neuron's\ sigmoid\ value$

Hidden Layer Error Calculation:
Backpropagate output layer errors to the hidden layer using the chain rule:

$$\delta_j = [\sum_{i=1}^{m} (\delta_k * w_{2k}j)] * f'(n_1j)$$

$\delta_k = output\ neuron's\ error$

$w_{2i}k = weight\ from\ hidden\ neuron\ j\ to\ output\ neuron\ k$

Loss Function, Weight, Bias, and Error Update

After backpropagation, each neuron's weights and biases are updated. The error for the dataset is updated using the loss function, Mean Squared Error (MSE). Updates are implemented in `Neuron.updateWeights()`.

Bias function update:

$$b_1k = b_1k + \alpha * \delta_k$$
$$b_2j = b_2j + \alpha * \delta_j$$

$\delta_{k/j} = output/hidden\ layer\ error$

$\alpha = learning\ rate$

Hidden layer weight update:

$$w_{1j}i = w_{1j}i + \eta * \delta_j * x_i$$

$\eta = learning\ rate$

$\delta_j = error$

$x_i = neuron\ value$

Output layer weight update:

$$w_{2k}j = w_{2k}j + \eta * \delta_k * z_j$$

$\eta = learning\ rate$

$\delta_k = error$

$z_i = neuron\ value$

Mean square error:

$$MSE = \frac{1}{n}\sum_{i=1}^{n} (f_i - y_i)^2$$

$n = number\ of\ data\ points$

$f_i = value\ return\ from\ network\ output\ for\ data\ point\ i$

$y_i = actual\ value\ i.e\ target\ value\ for\ data\ point\ i$

This completes one training example. The process continues for all rows in the training dataset to complete one epoch. After each epoch, the MSE is calculated across all training examples. Training continues until either:

- MSE < 0.001 (convergence criterion), or
- Maximum epochs reached (10,000 in our implementation)

Once training completes, the learned weights and biases are used to evaluate the test dataset.

Training Parameters and Configuration

The learning rate was set to 0.1 following the recommendation of Popescu et al. (2009), who recommended that general problems only require learning rates around 0.2. The termination criteria are that the MSE is <0.001 or the number of training epochs has reached 10000. 10 Random seeds were chosen for reproducibility {12345, 23456, 34567, 45678, 56789,67890, 78901, 89012, 90123, 21234}.

2.3 Decision Tree

Overview

Decision trees are employed to represent structured decision-making processes. They function as classifiers and resemble flowcharts in structure:

- Each internal node evaluates a specific feature (e.g. *"Is price growth greater than 5%?"*).
- Each branch denotes the outcome of the feature test.
- Each leaf node assigns a class label (e.g. *"Buy"* or *"Don't Buy"*).

Instances are classified by traversing the tree from the root node to a leaf node, following the path determined by the outcomes of feature tests along the way.

Impurity

In the context of decision trees, *impurity* refers to the degree of heterogeneity within a group of data points based on their class labels. A group is considered *pure* if all instances belong to the same class (e.g. all 0s or all 1s). Conversely, a group is *impure* if it contains a mix of classes - for example, a 50:50 split between class 0 and class 1 represents maximum impurity.

The greater the impurity, the more difficult it becomes to make a clear decision based on the data. *Entropy* is a metric used to quantify this uncertainty. It provides a numerical value that reflects how mixed the class labels are within a subset of the data.

Entropy

Entropy is a metric used to measure the level of impurity or disorder within a dataset. A dataset is considered *pure* when all instances belong to the same class, resulting in an entropy value of 0. As the diversity of class labels increases, so does the entropy, reflecting greater uncertainty in classification.

For binary classification problems, entropy is calculated using the following formula:

$$Entropy \ = \ - p_0 \ * \ log_2(p_0) \ - p_1 \ * log_2(p_0)$$

$Where$:
$p_0 \ = \ is\ the\ proportion\ of\ instances\ belonging\ to\ class\ 0$
$p_1 \ = \ is\ the\ proportion\ of\ instances\ belonging\ to\ class\ 1$

This formula quantifies the unpredictability in the dataset, guiding the construction of decision trees by favouring splits that reduce entropy.

Splits

A split in a decision tree occurs when the dataset is divided based on the value of a particular feature. The objective of a split is to partition the data into smaller subsets that are more homogeneous in terms of their class labels - that is, subsets with lower entropy or impurity. Effective splits are critical to improving the tree's ability to make accurate classifications.

Splitting Criteria: Information Gain

Information Gain (IG) measures the effectiveness of a feature in classifying the data by quantifying the reduction in entropy achieved through a split. It represents the amount of "knowledge" gained by dividing the dataset based on a specific attribute. The goal is to select the split that results in the greatest separation of classes - that is, the split that produces the most homogeneous child subsets.
Information Gain is calculated as:

$$Information\ Gain\ =\ Entropy_{parent}\ -\ \sum_{i=1}^{k}(\frac{n_i}{n})\ *\ Entropy_i$$

$Where$:

$Entropy_{parent}\ =\ the\ entropy\ of\ the\ original\ dataset\ before\ the\ split$

$Entropy_i\ =\ the\ entropy\ of\ subset\ i$

$n_i\ =\ the\ number\ of\ instances\ in\ subset\ i$

$n\ \ =\ the\ total\ number\ of\ instances\ in\ the\ parent\ set$

$k\ =\ the\ number\ of\ resulting\ subsets\ after\ the\ split$

A higher Information Gain indicates a more effective split for separating the classes.

Tree Construction

Decision trees are constructed recursively by identifying the optimal feature and threshold to split the data at each step. The primary objective during this process is to create branches that lead to increasingly pure - or homogeneous - subsets.
At each node, the algorithm selects the split that yields the highest Information Gain (or the greatest reduction in impurity), and this process continues until a stopping criterion is met, such as maximum depth, minimum node size, or complete purity.

Pseudocode

```
Function BuildTree(dataset):

    If all examples are in the same class:
        Return a leaf with that class

    If dataset is empty or no attributes left:
        Return a leaf with the majority class

    For each attribute:
        Calculate Information Gain for splitting on this attribute

    Select the attribute with the highest Information Gain

    For each value (or range) of that attribute:
        Split the dataset into a subset
        Recursively call BuildTree(subset) to create child node

    Return a decision node with branches for each split
```

3. Experimental Setup

Genetic Programming (GP) Parameters

Parameter	Value
Population Size	200
Initial tree generation	Ramped half-and-half
Tree depth	12
Selection method	Tournament
Tournament size	5
Crossover Rate	0.85
Mutation Rate	0.1
Maximum generations	50

Multi-Layer Perceptron (MLP) Parameters

Parameter	Value
Hidden Layers	1
Hidden Neurons	2
Activation	Sigmoid activation
Learning Rate	0.1
Error Threshold	0.001
Maximum Epochs	10000

Decision Tree Parameters

Parameter	Values
Minimum number of samples needed for splitting	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 18, 20, 25
Maximum tree depth	1, 2, 4, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 20, 25, 30

4. Results and Analysis

4.1 MLP Results and Analysis

Our MLP was evaluated across 10 runs using seeds {12345,23456,34567,45678,56789,67890,78901,89012,90123,21234}, an architecture that consisted of 2 neurons in the hidden layer. 2 was determined to be the best number of neurons for the hidden layer based on initial experiments. Initial experiments tested the network with 2, 3, 4, and 10 neurons, with 2 and 3 providing the best results (99.62% accuracy).

Training Performance

All runs reached the maximum limit of 10,000 epochs. The training accuracy ranged from 91.48% to 98.10%. The F1 score range was from 0.9254 to 0.9817. MSE values ranged from 0.0389 - 0.0453, which is well above the convergence threshold of 0.001.

Failure to reach the MSE threshold suggests that the error criterion is too strict, more training epochs may reach the desired MSE or that the learning rate and number of hidden neurons are too low to allow faster convergence and need fine-tuning. But, despite this, high accuracy was still achieved, indicating that the model is still learning quite well and this is shown in the test results data.

Testing Performance

Best performance came from the seed 21234, which had a test accuracy of 100%, F1 score of 1 and MSE value of 0.00116, which is quite close to the convergence threshold of 0.001. The worst performance came from the seed 45678, with a test accuracy of 99.24%, F1 score of 0.9924, and MSE of 0.003597. Overall, the errors were minimal, with the worst-performing seeds having up to 2 misclassifications.

Generalisation Performance

The training dataset consisted of 998 training samples and 263 test samples. Despite the model's failure to converge to an MSE value of < 0.001 during training, the model still achieved high accuracy (91.48% - 98.10%) on the large dataset. Once again, the failure to reach the MSE criterion of < 0.001 for our shallow network may be too strict, though the MSE range (0.0389 - 0.0453) achieved during training, given the large dataset. The model's performance on the test data outperformed the training data with minimal errors (0-2 misclassifications).

Architecture

During initial experimentation, we experimented with hidden neuron values 2, 3, 4, and 10 to find an optimal configuration for our network. 2 and 3 gave the best accuracy with and the decision was made to settle with 2 hidden neurons to keep the network simple.

Hidden Neurons	Average accuracy (%)	Performance Rating
2	99.62	Optimal
3	99.62	Optimal
4	98.86	Good
10	99.24	Very Good

4.2 Decision Tree Results and Analysis

Explanation of key parameters

- **maxDepth**: This parameter defines the maximum allowable depth of the decision tree. It controls how many levels the tree can grow. A higher maxDepth allows the tree to capture more complex patterns in the training data, but it also increases the risk of overfitting. Conversely, a lower maxDepth limits model complexity and may lead to underfitting.
- **minSamplesSplit**: This parameter specifies the minimum number of samples required to split an internal node. It acts as a form of regularisation by preventing the tree from growing branches based on very small data subsets.

Testing Performance

This evaluation involved training decision trees using various combinations of minSamplesSplit and maxDepth parameters. Trees were constructed using a training dataset and then evaluated on a separate testing file to assess model generalisability.

- Accuracy and F1 Score generally improved with increasing maxDepth, peaking around values of 10–11.
- Beyond maxDepth 11, further increases in depth yielded diminishing returns in Accuracy and F1 when minSamplesSplit was low.
- Increasing minSamplesSplit effectively prunes the tree by limiting growth. For a fixed maxDepth, this results in shallower trees with fewer nodes and leaves.
- The pruning effect of minSamplesSplit is more pronounced at higher maxDepth values (e.g., 8 and above), where it significantly influences both model performance and structure.
- For lower maxDepth values (1–7), tree structures and performance metrics remain mostly unaffected by changes in minSamplesSplit, indicating that the split threshold is rarely exceeded.
- The highest observed Accuracy (67.68%) and F1 Score (0.675) were achieved at maxDepth ≥ 10 with low minSamplesSplit values (0–2).
- Training time generally increased with tree size and lower minSamplesSplit values, although some variability was noted even with identical settings.
- At maxDepth = 1, the model defaulted to predicting the majority class for all instances, yielding baseline performance.

Training Performance

This experiment used the same training file for both constructing and evaluating the decision trees.

- Accuracy and Macro F1 Score increased sharply with greater maxDepth, especially for small minSamplesSplit values.
- Maximum performance (100% Accuracy and 1.0 Macro F1) was reached at maxDepth ≥ 10 and minSamplesSplit ≤ 2, indicating overfitting.
- Increasing minSamplesSplit for a fixed maxDepth reduced performance and model complexity (fewer nodes, leaves, and shallower depth).
- Model complexity grew with maxDepth but was constrained by the value of minSamplesSplit.
- Training time increased with maxDepth and decreased with minSamplesSplit, though there was variability for identical settings.

Results

The following parameter ranges yielded the best performance across both training and testing datasets:

Metric	**Testing File**	**Training File**
maxDepth	10–30	10–30
minSamplesSplit	0–2	0–2
Resulting Accuracy	67.68%	100%
Resulting F1 Score	0.675	1.0

Analysis

The analysis shows that increasing maxDepth and lowering minSamplesSplit generally improves performance, but also increases the risk of overfitting, as evidenced by perfect scores on the training data.

Pruning via minSamplesSplit plays a key role in managing model complexity and generalisation. The optimal balance between depth and split threshold is critical for achieving strong performance on unseen data.

4.3 Wilcoxon Signed-Rank Test for Comparing GP and MLP

The Wilcoxon signed-rank test is used to assess whether or not the population mean ranks of two related samples differ. The test is useful when comparing the data and performance between two models on the same dataset. In this case, we will be comparing the predictive performances of GP and MLP.

To perform the test, performance values from both the GP and MLP models need to be obtained using the same data gathered by making multiple runs through the models with different seeds. We will be using the following data values:

- Accuracy: how well the model predictions were for each run
- F1 Score: the harmony between precision and recall for each run

These values will be paired for GP and MLP per each run and used as the input for the test. The test will work as follows:

1. Data will be grouped into GP-MLP pairs for both the accuracy and F1 Score metrics where both data value types are within the range between 0 and 1.
2. The differences between the GP and MLP pairs will be calculated for both accuracy and F1 scores
3. Absolute values of the differences are ranked from lowest to the highest value.
4. Signs will be assigned to the ranks according to the value of the differences previously calculated
5. The sum of the positive ranks will be calculated
6. The sum of the negative ranks will be calculated
7. A test statistic *W* will be equated to the smaller of the two sums
8. Calculate a p-value:
    a. by finding the normal approximation using:

$$Z = \frac{W - \frac{n(n+1)}{4}}{\sqrt{\frac{n(n+1)(2n+1)}{24}}}$$

    b. If Z is a standard normal distribution and P(Z≥|z|) can be looked up:

$$p\text{-value} = 2 \times P(Z \geq |z|)$$

9. Compare *W* to a critical value or use a p-value to determine significance
    a. If the p-value is less than 0.05, there is statistically significant difference
    b. If greater than or equal to 0.05 shows no significant difference

Running the Test

Run Number	GP Accuracy	MLP Accuracy	GP F1 Score	MLP F1 Score
1	0.00	0.9962	0.00	0.9962
2	0.5133	1	0.6701	1
3	0.5133	1	0.5493	1
4	0.4943	0.9924	0.0148	0.9924
5	0.7452	1	0.7988	1
6	0.6388	1	0.6058	1
7	0.5133	1	0.6751	1
8	0.5057	1	0.6000	1
9	0.6312	0.9962	0.6255	0.9962
10	0.6236	1	0.7227	1

Calculating Differences (Absolute Values)

Run	Accuracy	*Ranking	F1 Score	*Ranking
1	0.9962	10	0.9962	10
2	0.4867	7	0.3299	4
3	0.4867	7	0.4507	6
4	0.5081	9	0.9776	9
5	0.2548	3	0.2012	1
6	0.3612	5	0.3942	5
7	0.4867	7	0.3249	3
8	0.4943	8	0.4000	5
9	0.3350	4	0.3707	4
10	0.3764	6	0.2773	2

The sum of the positive accuracy rankings is: 0
The sum of the negative accuracy rankings is: 55
The sum of the positive F1 score rankings is: 0
The sum of the negative F1 score rankings is: 55

Wilcoxon test statistic:

W = 0

Small p value (< 0.001), highly significant difference

***Note: Ranking**
If ranking is the same, use average ranking i.e. 7,8,9 are tied, therefore (7+8+9)/3 = 8, each gets a ranking of 8.

Analysis

All ranks are negative, which means that MLP consistently outperformed GP across every run. There is a statistically significant difference in performance between GP and MLP models. The MLP significantly outperforms GP in both accuracy and F1 score when predicting whether to buy a stock based on historical data. This justifies further investigation into neural-based models for financial forecasting over symbolic evolution-based approaches in this context.

4.4 Performance Comparison

Model	**Seed Value**	**Training**		**Testing**	
		Acc	**F1**	**Acc**	**F1**
Genetic Programming	1742828254320L	0.5591	0.6884	0.7452	0.7988
Multi-Layer Perceptron	21234	0.9809	0.9817	1	1
Decision Tree	N/A	1	1	0.6768	0.67529

***Note: No seed for Decision Tree**
In the implementation based on the C4.5 algorithm, the construction of the decision tree is entirely deterministic. This means that every decision - such as which feature to split on and where to split - is made using clear, mathematical criteria like Information Gain Ratio.

As a result, given the same input dataset, the algorithm will consistently produce the same decision tree. There is no element of randomness in the splitting process, and therefore, no random seed is required.

5. Conclusion

In this assignment, we tested and compared three machine learning models, namely Genetic Programming (GP), Multi-Layer Perceptron (MLP), and Decision Trees, in order to see how well they could be used for financial forecasting. The goal was to predict whether a stock should be bought based on historical data.

The GP model gave decent results and was quite useful in terms of interpretability, since it produced mathematical expressions that we could understand. However, when it came to actual performance, the MLP clearly outperformed it in both accuracy and F1 score.

The MLP, which we set up with a single hidden layer and sigmoid activation, consistently scored over 90% on both training and testing accuracy. This showed how effective it was at capturing more complex relationships in the data despite having not met the MSE convergence criteria.

To make sure the difference in performance wasn't just random, we used the Wilcoxon signed-rank test to compare the results from GP and MLP. The test gave us a p-value of less than 0.001, which means the improvement from MLP was statistically significant.

The Decision Tree also showed good training accuracy, sometimes even perfect, but its test results were a bit inconsistent. This suggests that it might be overfitting the training data and not generalizing as well to new inputs.

Final Thoughts:

- MLP was the best overall performer and showed the most consistent results.
- GP was easier to interpret and good for understanding the logic behind decisions, but it didn't match MLP in terms of accuracy
- Decision Trees were fast and simple to use, but they might need extra tuning to avoid overfitting

6. References

1. Koza, J.R., 1994. *Genetic programming as a means for programming computers by natural selection*.
2. Popescu, M.C. et al., 2009. *Multilayer perceptron and neural networks*.
3. Singhal, S. and Jena, M., 2013. *A study on WEKA tool for data preprocessing, classification and clustering*.
4. Jaiswal, J.K. and Das, R., 2017. *Application of artificial neural networks with backpropagation technique in the financial data.*