COS 314
Artificial Intelligence
Assignment Three

24 May 2025

Dewald Colesky u23536030
Liam van Kasterop u22539761
Henco Pretorius u23525381
Herrie Engelbrecht u22512374

Table of contents:

Genetic Programming Classification Algorithm

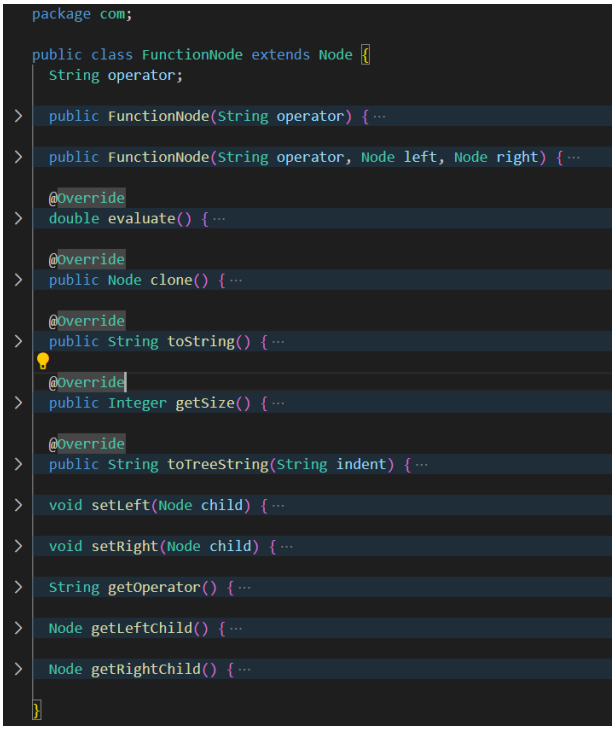
Introduction

For this assignment we were tasked with implementing a Genetic Programming (GP) Algorithm used for classification in whether a financial stock should be purchased based on historical data. The system evolves mathematical expressions that analyze input features to make buy/sell decisions.

Representation

Solutions are represented as expression trees with:

- Function nodes: Mathematical, comparison and logical operators(+, -, *, /, >, <, AND, OR, etc.)



```
package com;

public class FunctionNode extends Node {
  String operator;

  public FunctionNode(String operator) {…

  public FunctionNode(String operator, Node left, Node right) {…

  @Override
  double evaluate() {…

  @Override
  public Node clone() {…

  @Override
  public String toString() {…

  @Override
  public Integer getSize() {…

  @Override
  public String toTreeString(String indent) {…

  void setLeft(Node child) {…

  void setRight(Node child) {…

  String getOperator() {…

  Node getLeftChild() {…

  Node getRightChild() {…
}
```

- Terminal nodes: Input features (X1-X5) representing historical data

```java
package com;

public class TerminalNode extends Node {
  double Value;
  String label;

  public TerminalNode(String label) {…

  public TerminalNode(String label, Node left, Node right) {…

  void SetValue(double val) {…

  @Override
  double evaluate() {…

  @Override
  public Node clone() {…

  @Override
  public String toString() {…

  @Override
  public Integer getSize() {…

  @Override
  public String toTreeString(String indent) {…

  double getValue() {…

}
```

Initial Population Generation

The method used for initial tree generation was the full method. We first experimented with the ramped half-and-half method, but after testing it was found that the full method showed better results.

```java
ArrayList<Individual> GenerateInitialPopulation() {
  ArrayList<Individual> tempSolution = new ArrayList<>();
  Individual tempIndiv;
  for (int i = 0; i < PopulationSize; i++) {
    tempIndiv = fullGeneration(currentDepth:0);
    tempSolution.add(tempIndiv);
  }

  return tempSolution;
}
```

```java
Individual fullGeneration(int currentDepth) {
  if (currentDepth >= MaxDepth) {
    return new Individual(new TerminalNode(TerminalSet[random.nextInt(TerminalSet.length)]));
  } else {
    String operator = getRandomFunction();
    Node newNode = new FunctionNode(operator);

    if (operator.equals("NOT") || operator.equals("sqrt") || operator.equals("abs")) {
      newNode.setLeft(fullGeneration(currentDepth + 1).getRoot());
    } else {
      newNode.setLeft(fullGeneration(currentDepth + 1).getRoot());
      newNode.setRight(fullGeneration(currentDepth + 1).getRoot());
    }

    return new Individual(newNode);
  }
}
```

Fitness Function

The fitness function in our Genetic Programming implementation measured the accuracy of buy or sell predictions made by the mathematical expression represented by each individual. During evaluation, each individual's expression was computed using input features from the training dataset. If the resulting value was greater than 1, the prediction was classified as a "buy"; if the value was less than 0, it was classified as a "sell." This prediction was then compared against the actual label in the training set. For every correct prediction, a counter was incremented. The final value of this counter served as the individual's fitness score, reflecting its predictive accuracy on the training data.

```
void CalculateFitness(ArrayList<Individual> temp) {
  for (Individual current : temp) {
    int CorrectCount = 0;
    for (int i = 0; i < testingSet.getSize(); i++) {
      double[] inputSet = testingSet.getFeatures(i);
      Integer expectedOutput = testingSet.getExpected(i);

      bindInputs(current.getRoot(), inputSet);

      double result = current.getRoot().evaluate();
      if ((result >= 1 && expectedOutput == 1) || (result < 1 && expectedOutput == 0)) {
        CorrectCount++;
      }
    }
    current.setFitness(CorrectCount);
  }
}
```

Selection

For our Genetic Programming algorithm, we employed tournament selection as the method for selecting parents. From the current population, two individuals were randomly chosen by selecting two random indices. The fitness scores of these two individuals were then compared, and the individual with the higher fitness was selected as a parent for generating offspring. This method helps to balance exploration and exploitation by introducing a level of randomness while still favoring fitter individuals.

```
Individual Tournament(ArrayList<Individual> tempPopulation) {
  Individual a = tempPopulation.get(random.nextInt(tempPopulation.size()));
  Individual b = tempPopulation.get(random.nextInt(tempPopulation.size()));
  return a.fitness > b.fitness ? a : b;
}
```

Genetic Operators

In our Genetic Programming (GP) implementation, we utilized two primary genetic operators: crossover and mutation. These operators are crucial for introducing diversity into the population and guiding the evolutionary process toward optimal solutions. The process begins with the crossoverAndMutate() method, which takes two parent individuals and generates offspring. Initially, each offspring is created as a deep clone of its corresponding parent to ensure that the original individuals remain unchanged.

The crossover operation is performed through the performCrossover() method. Here, a random subtree is selected from each parent using the getRandomSubtree() method, which gathers all nodes in the tree and selects one at random. These selected subtrees are cloned and then swapped between the two offspring using the replaceSubtree() method. This subtree crossover allows for meaningful genetic material to be exchanged between individuals, potentially combining beneficial traits from both parents.

Following crossover, each offspring undergoes mutation with a probability defined by the MutationRate(). If selected for mutation, a random subtree within the individual is chosen and replaced with a newly generated subtree produced by the generateRandomSubtree() method. This method uses a tree generation strategy such as the full method, typically generating a small subtree of fixed depth (e.g., depth 3). The replaceSubtree() method is again used to perform the substitution.

```
Individual[] crossoverAndMutate(Individual parent1, Individual parent2) {
  Individual offspring1 = parent1.clone();
  Individual offspring2 = parent2.clone();

  performCrossover(offspring1, offspring2);

  if (Math.random() < MutationRate) {
    mutate(offspring1);
  }
  if (Math.random() < MutationRate) {
    mutate(offspring2);
  }

  return new Individual[] { offspring1, offspring2 };
}
```

```
void performCrossover(Individual ind1, Individual ind2) {
  Node node1 = getRandomSubtree(ind1.getRoot());
  Node node2 = getRandomSubtree(ind2.getRoot());

  Node node1Copy = node1.clone();
  Node node2Copy = node2.clone();

  replaceSubtree(ind1.getRoot(), node1, node2Copy);
  replaceSubtree(ind2.getRoot(), node2, node1Copy);
}

void mutate(Individual individual) {
  Node mutationPoint = getRandomSubtree(individual.getRoot());
  Node newSubtree = generateRandomSubtree();
  replaceSubtree(individual.getRoot(), mutationPoint, newSubtree);
}

Node generateRandomSubtree() {
  return fullGeneration(currentDepth:3).getRoot();
}

Node getRandomSubtree(Node root) {
  List<Node> nodeList = new ArrayList<>();
  collectNodes(root, nodeList);
  return nodeList.get(new Random().nextInt(nodeList.size()));
}
```

```java
void collectNodes(Node node, List<Node> nodes) {
  if (node == null)
    return;
  nodes.add(node);
  if (node instanceof FunctionNode) {
    collectNodes(((FunctionNode) node).getLeftChild(), nodes);
    collectNodes(((FunctionNode) node).getRightChild(), nodes);
  }
}

boolean replaceSubtree(Node root, Node target, Node replacement) {
  if (root instanceof FunctionNode) {
    FunctionNode fn = (FunctionNode) root;

    if (fn.getLeftChild() == target) {
      fn.setLeft(replacement);
      return true;
    } else if (fn.getRightChild() == target) {
      fn.setRight(replacement);
      return true;
    } else {
      return replaceSubtree(fn.getLeftChild(), target, replacement)
          || replaceSubtree(fn.getRightChild(), target, replacement);
    }
  }
  return false;
}
```

Population Replacement

In our GP implementation, we employed the steady-state replacement method for population updating. The replacement function receives both the current generation and the newly created offspring. For each offspring, a random individual from the existing population is selected. If the offspring has a higher fitness value than the selected individual, it replaces that individual in the population. This method ensures that only fitter solutions survive, promoting steady improvement over time while preserving genetic diversity within the population.

```java
ArrayList<Individual> SteadyState(ArrayList<Individual> oldGeneration, ArrayList<Individual> offspring) {
  ArrayList<Individual> newGeneration = new ArrayList<>(oldGeneration);

  for (Individual child : offspring) {
    boolean replaced = false;
    for (int attempts = 0; attempts < 10; attempts++) {
      int index = random.nextInt(newGeneration.size());
      if (child.fitness > newGeneration.get(index).fitness) {
        newGeneration.set(index, child);
        replaced = true;
        break;
      }
    }
  }

  return newGeneration;
}
```

Termination

The algorithm terminates after a max number of generations. This can be specified by the user or the default value of 100 generations will be used.

GP Algorithm Implementation

```
public void Algorithm() {
  ArrayList<Individual> tempPopulation = GenerateInitialPopulation();
  int count = 0;
  while (count < MaxGenerations) {
    CalculateFitness(tempPopulation);

    Individual parentOne = Tournament(tempPopulation);
    Individual parentTwo = Tournament(tempPopulation);

    Individual[] offspring = {parentOne.clone(), parentTwo.clone()};
    if (Math.random() < CrossoverRate) {
      offspring = crossoverAndMutate(parentOne, parentTwo);
    }

    ArrayList<Individual> OffSpring = new ArrayList<>();
    for (Individual curIndividual : offspring) {
      OffSpring.add(curIndividual);
    }
    CalculateFitness(OffSpring);

    tempPopulation = SteadyState(tempPopulation, OffSpring);

    count++;
  }

  Population = tempPopulation;
}
```

After the algorithm terminates, the individual with the highest fitness is selected from the final population. This best-performing individual is then evaluated on the testing dataset to measure its predictive accuracy. Our results demonstrate that the algorithm is capable of consistently evolving individuals that achieve up to 100% accuracy on the test data.

Multi-Layer Perceptron

Introduction

Multi-Layer Perceptron (MLP) is a feedforward neural network, with input, hidden, and output layers. Each layer is fully connected to the next and uses backpropagation. Our task was to predict the output given five features of Bitcoin stock.

Initial Solution

User specifies the hidden layer size as well as amount of hidden layers. Each neuron in the hidden layer uses xavier initialization for weights and assigns a small gaussian to the bias.

```java
public Neuron(int inputSize, long seed) {
    weights = new double[inputSize];
    Random r = new Random(seed);
    double limit = Math.sqrt(6.0 / (inputSize + 1));
    for (int i = 0; i < inputSize; i++) { // small random weight for each
        weights[i] = r.nextDouble() * 2 * limit - limit; // xavier init
    }
    bias = r.nextGaussian() * 0.01;

}
```

Implementation

The MLP was implemented using Java (Maven). The training CSV inputs were read into memory and normalized to [-1,1]. Inputs are activated in batches, as well as backpropagation being in batches. For testing, CSV is read into memory and the output of the neural network is compared to the expected output. Accuracy, and F1 is calculated while testing the MLP. The loss function used was binary cross entropy.

Decision Tree

We use Weka's J48 (an open-source C4.5 implementation) to predict price movement direction ("UP"/"DOWN") on the BTC dataset, enhanced with technical indicators. The tree facilitates interpretability by showing which features and thresholds drive each buy/sell decision.

Implementation

- **Feature Engineering**: In addition to raw OHLC data, we computed seven technical indicators—SMA_5, SMA_10, RSI, volatility (high–low ratio), price-change %, high/low ratio, and volume-price trend—and appended them to the original attributes.
- **Data Preparation**: Both training and test CSVs were loaded via Weka's `CSVLoader`, converted to nominal class labels ("UP"/"DOWN") using 33rd/67th percentile thresholds, and set class index to the last attribute.
- **Evaluation**: After building the classifier on the training set, we recorded tree size, number of leaves, training accuracy, 10-fold CV accuracy, test accuracy, F1 score, and confusion matrix.

Experimental Setup

Algorithm Used

Genetic Programming Classification Algorithm (GP)

- **Max Num Generations:** 100
- **Population Size:** 20
- **Crossover Rate:** 0.8
- **Mutation Rate:** 0.12

Multi-Layer Perceptron (MLP)

- **Parameter Tuning:** learning rate, batch size, hidden layer size, hidden layer amount, patience, minimum improvement, max iterations were tuned to gain an overall best parameter setup.
- **Learning Rate:** The amount weights and bias are updated each iteration. (0.01)
- **Batch Size:** The amount of inputs are averaged and processed at once. (16)
- **Hidden Layer Size:** The amount of neurons per hidden layer. (32)
- **Hidden Layer Amount:** The amount of hidden layers. (2)
- **Patience:** How many iterations allowed without improvement. (50)
- **Minimum Improvement:** The smallest difference to count as an improvement. (0.01)
- **Max Iterations:** The maximum iterations. (1000)

Decision Tree (DT)

- **J48 Configuration**:
    - Confidence factor (`-C`): 0.1
    - Minimum instances per leaf (`-M`): 10
    - Unpruned: **false** (pruning enabled)
    - Binary splits (`-B`): enabled
    - Subtree raising (`-S`): enabled
    - No MDL correction (`-J`): enabled
    - Seed (`-Q`): 1

Experimental Environment

Hardware

- **Processor**: AMD Ryzen 5 7600 × 12 cores
- **Memory**: 32 GB DDR5 RAM
- **System**: Micro Star International Co., Ltd MS7E28
- **Operating System**: Ubuntu 24.04.1 LTS

Software

- **Language**: Java 21
- **IDE**: Visual Studio Code

Benchmark and Test Set

The following data sets will be tested:

- BTC_test.csv - 5 features, 1 output

Evaluation Metrics

- **Time Taken**: Total time required by algorithms to find a solution.
- **Iterations Completed**: Number of iterations performed before convergence or stopping.
- **Solution Quality**: The final accuracy and F1 score.

Experimental Procedure

1. **Input Data**: Read in data representing features and expected result.
2. **User Input**:Run the program (With fixed or random seed)
3. **Algorithm Execution**:
    - For each problem instance, run the algorithm for the defined number of runs.
    - MLP will run for specified iterations, with a set amount of no improvement iterations, and until minimum improvement is not reached.
4. **Data Collection**:
    - Save the following data into a file:
        - Seed used for each run.
        - Class folder of MLP.
        - Run points for MLP (WithStops/name-here.json).
        - Accuracy for testing and training.
        - F1 score for testing and training.

Comparison

The performance of algorithms will be compared based on:

- **Accuracy**: Which had the highest cost.
- **F1**: The sum of costs of each instance.
- **Runtime**: How many iterations/time (directly corresponding) are needed to reach the best solutions or stopping criteria.
- **Wilcoxon signed-rank test:** A statistical analysis between GP and MLP.

Results

COS314A3Results - All results and workings for wilcoxon with some extra details

Model	Seed value	Training		Testing	
		Acc(%)	F1	Acc(%)	F1
Genetic Programming	8458451357342157870	87.45	0.8491	100	1
MLP	7338150357446147275	88.78	0.8821	99.62	0.9962
Decision Tree	5927381049672518364	99.10	0.9905	98.86	0.9886

Wilcoxon sign rank test between MLP and GP

MLP	GP
94.68,	100.0,
96.58,	100.0,
91.63,	100.0,
89.35,	73.76425855513308,
90.87,	100.0,
92.78,	88.212927756654,
98.86,	72.6235741444867,
87.83,	100.0,
95.44,	74.90494296577947,
92.4,	100.0,
96.2,	81.36882129277566,
88.21,	76.80608365019012,
87.83,	95.43726235741445,
95.82,	100.0,
86.69,	98.47908745247148,
89.73,	79.84790874524715,
89.35,	72.6235741444867,
94.68,	77.9467680608365,

93.16,	100.0,
93.16,	99.61977186311786,
94.3,	90.11406844106465,
92.02,	100.0,
96.58,	100.0,
97.34,	87.45247148288973,
90.11,	76.80608365019012,
87.07,	100.0,
92.78,	100.0,
94.68,	77.56653992395437,
94.3,	98.09885931558935,
91.63	73.38403042

Using **alpha=0.01**, **two tailed**
H0: No difference between MLP and GP
Ha: There is a difference between MLP and GP

W-value: 169
Mean Difference: -7.46
Sum of pos. ranks: 296 (GP > MLP)
Sum of neg. ranks: 169 (MLP > GP)
Z-value: -1.3061
Mean (W): 232.5
Standard Deviation (W): 48.62
Sample Size (N): 30

Result 1 - Z-value
The value of z is-1.3061. The p-value is .1902.
The result is not significant at p < .01.
Result 2 - W-value
The value of W is 169. The critical value for W at N = 30 (p < .01) is 109.
The result is not significant at p < .01.

W = 169 > 109
Not significant at p < 0.01
Not significant at p < 0.05 (W <= ~135)
Mean dif = -7.46
Mlp was on average 7.46 points lower than GP

Since p = 0.1902 > 0.05, we fail to reject the null hypothesis.
Although GP had a higher mean accuracy (by 7.46%), the Wilcoxon signed-rank test showed that the difference was not statistically significant.
Therefore, we cannot conclude that GP outperforms MLP with confidence.

Comparisons

Comparisons between , Decision tree, the MLP and GP.
In terms of Quantitative results, The Decision tree got 98.86% test accuracy and remained stable across various pruning settings, Training accuracy ranged from 99.1% to 99.5%, and cross validation scores stayed between 98.1% and 98.5% , showing that it generalises well without overfitting.

In contrast, the Multi-Layer Perceptron (MLP) showed more fluctuation. Its test accuracy varied widely—from 90.49% to an impressive 99.62%—depending on the run. While it did reach the highest single accuracy score, its training accuracy also ranged considerably (from 86.97% to 97.49%), suggesting that its performance is quite sensitive to initial conditions and hyperparameter tuning.

Genetic Programming had the most unpredictable results. While it occasionally reached perfect accuracy (100%), it also dropped as low as 72.62% in other cases. This huge swing in performance shows that, although the method has high potential, it may not be dependable enough for consistent, real-world use.

Statistical Significance:
A Wilcoxon signed-rank test was used to compare the performance of the MLP and Genetic Programming approaches (α = 0.01, N = 30). Despite Genetic Programming showing a 7.46% higher average accuracy, the difference wasn't statistically significant. The test produced a W-value of 169, which is well above the critical value of 109, and a p-value of 0.1902. This suggests that the difference in performance is likely due to random variation rather than a true advantage of one method over the other.

The Decision Tree has a relatively rigid structure , which might make it less effective when market

Each algorithm has its own weaknesses, which point to ways we could improve them in the future. The Decision Tree, for example, is a bit rigid and might not handle sudden changes in

market behavior very well. To fix this, we could retrain it more often or combine multiple trees that each focus on different market situations.

The neural network (MLP) was very sensitive to how it was set up, especially things like settings and starting values. This means we might get better and more stable results by using smarter ways to choose those settings or by averaging the results from several networks.

Genetic Programming had the most unpredictable results—it sometimes did really well and other times poorly. To make it more reliable, we could improve the way it picks and combines solutions, so it finds good answers more often without going off track.

Conclusions

This comparison shows that when it comes to predicting cryptocurrency prices, the best results don't always come from the most complex algorithms. Instead, real success comes from smart use of domain knowledge—especially through well-designed features. The strong performance of the J48 Decision Tree is a good reminder that great results happen when we combine solid algorithms with a deep understanding of the market.

These findings have real-world value for anyone building crypto trading systems, and they also help guide future research in financial machine learning. As crypto markets grow and change, some key lessons will stay important: focus on consistency, make models easy to understand, and use expert knowledge to guide your approach.

Looking ahead, the future of crypto prediction isn't about chasing the most advanced "black-box" models. It's about creating smarter ways to understand what drives price changes. This study lays the groundwork for doing just that—and for building systems that work in the real world.

References

Shiksha.com, 2025. *Understanding Multilayer Perceptron (MLP) Neural Networks*. Available: https://www.shiksha.com/online-courses/articles/understanding-multilayer-perceptron-mlp-neural-networks/ [Accessed: 13-May-2025].

GeeksforGeeks, 2025. *Multi-Layer Perceptron Learning in TensorFlow*. Available: https://www.geeksforgeeks.org/multi-layer-perceptron-learning-in-tensorflow/ [Accessed: 13-May-2025].

Elcaiseri, 2025. *Building a Multi-Layer Perceptron from Scratch with NumPy*. Medium. Available: https://elcaiseri.medium.com/building-a-multi-layer-perceptron-from-scratch-with-numpy-e4cee82ab06d [Accessed: 15-May-2025].

GeeksforGeeks, 2025. *Backpropagation in Neural Network*. Available: https://www.geeksforgeeks.org/backpropagation-in-neural-network/ [Accessed: 15-May-2025].

Weka Documentation, 2025. *J48 Classifier (Decision Trees)*. Available: https://weka.sourceforge.io/doc.stable/weka/classifiers/trees/J48.html [Accessed: 23-May-2025].

Weka Wiki, 2025. *Optimizing Parameters in Weka*. Available: https://waikato.github.io/weka-wiki/optimizing_parameters [Accessed: 23-May-2025].

University of Edinburgh, 2025. *Machine Learning Lecture: Week 4*. Available: https://www.inf.ed.ac.uk/teaching/courses/dme/html/week4.html [Accessed: 23-May-2025].