TAWANDA JIMU

COS 314 Assignment 3 – Machine Learning

U20494166

SYSTEM ENVIROMENT:

The performance of a search algorithm can depend on a variety of factors, including the hardware and software environment of the computer on which it is executed.This assignment was completed on the following system:

OS Name Microsoft Windows 10 Home Single Language
System Model HP Pavilion Aero Laptop 13-be0xxx
System Type x64-based PC
Processor AMD Ryzen 7 5800U with Radeon Graphics, 1901 Mhz, 8 Core(s), 16 Logical Processor(s)
Installed Physical Memory (RAM) 8,00 GB
Physical Disk Type Solid State Drive 512.00 GB

Task 1: Neural Network

TASK 2: GP Classification Algorithms

To solve the classification problem, I followed the following steps:

1. Dataset Split: I randomly split the dataset into an 80:20 ratio, where 80% of the data was used for training and 20% for testing. The training data was saved in a file called "training_data.data," and the remaining 20% was saved in a file called "testing_data.data."
2. Initial Population Generation: I initialized the population using the *population.evolve()* function with the following parameters:
    - Number of Generations: 50 (as specified in the assignment)
    - Population Size: 100 (as specified in the assignment)
    - Crossover Rate: 0.5 (chosen to balance exploration and exploitation. This rate allows for diverse combinations of solutions while refining good ones)
    - Mutation Rate: 0.2 (chosen to maintain population diversity and prevent premature convergence)
    - Max tree depth of 10 (It allows the trees to have a sufficient level of complexity to capture relationships and patterns in the data while preventing overfitting.)

    For generating the initial population, I used the ramped half and half tree generation method. This method combines the full and grow methods, creating trees of varying depths. This ensures exploration at different levels, facilitating the discovery of both shallow and deep solutions and promoting diversity.

3. **Fitness Function:** The fitness function is implemented in the *getFitness()* function within the *Population* class. It takes a decision tree and a dataset as inputs and returns the fitness value. To calculate fitness, the function uses the *classifyInstances* method of the *DecisionTree* class. This method classifies each instance in the dataset using the decision tree, and the number of correctly classified instances is recorded. The fitness value is then calculated as the ratio of correct classifications to the total number of instances in the dataset.

   The fitness value obtained from this function represents the accuracy of the decision tree on the given dataset. A higher fitness value indicates better performance in correctly classifying instances.

4. **Selection:** I employed tournament selection for parent selection during crossover. The *tournamentSelection* method in the *Population* class implements this selection method.

   During crossover, a parent tree is selected using the tournament selection function with a tournament size of 5. The *tournamentSelection* function randomly selects 5 trees from the population and chooses the one with the highest fitness as the parent for crossover.

5. **Genetic Operators:** The genetic operators used in my implementation are crossover and mutation.

   a. **Crossover:** The crossover operator combines genetic material from two parent trees to create new offspring trees. In my code, the *crossover* method performs the crossover operation. It takes a child tree and a parent tree as input. The child tree's root attribute is replaced with the parent tree's root attribute, resulting in the child inheriting a portion of the parent's genetic material.

   b. **Mutation:** The mutation operator introduces random changes in the genetic material of an individual tree. In the code, the *mutation* method performs the mutation operation. It takes a tree as input and randomly modifies its attributes. There is a 20% chance of either changing the root attribute to a random attribute from the predefined attribute set or recursively applying the mutation operation to the child branches of the tree.

   Both crossover and mutation operations contribute to the exploration and exploitation of the search space. Crossover combines good attributes from different individuals, while mutation introduces random changes that can potentially lead to new and improved solutions. These genetic operators help the algorithm search for better solutions over multiple generations.

6. **Population Replacement:** I used generational replacement as the population replacement method in my implementation.

   Generational replacement involves creating a new population of individuals in each generation, replacing the entire previous population. In my code, the **evolve** method performs the population replacement. It initializes a new population of decision trees with random individuals and evolves this population over multiple generations.

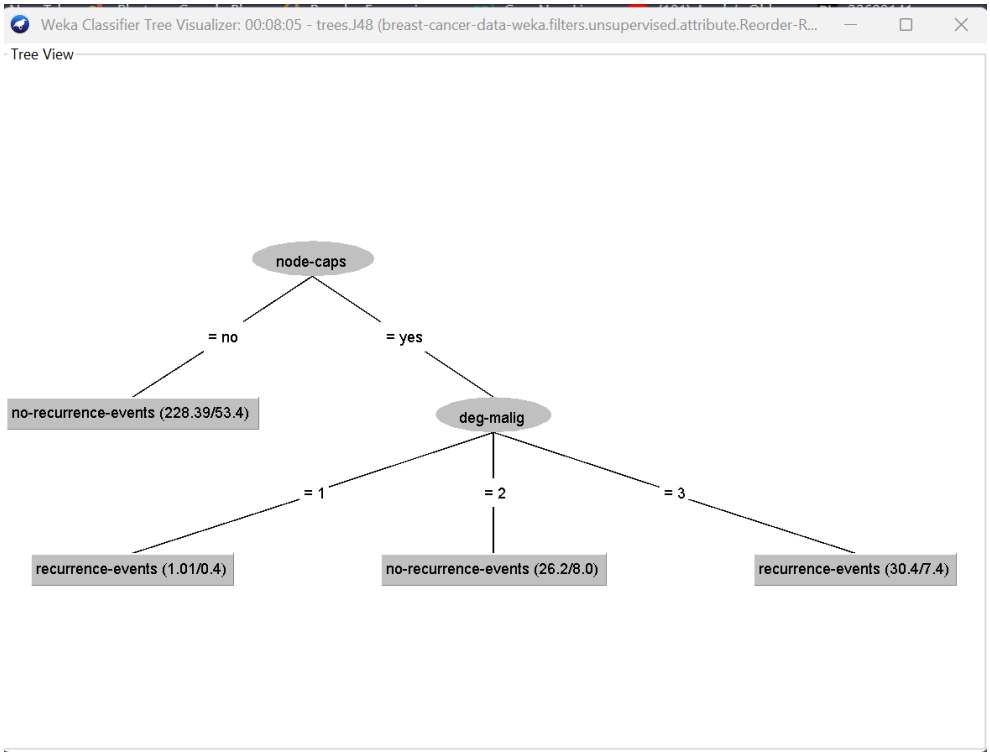
   In each generation, the algorithm applies crossover and mutation genetic operators to create new individuals (trees) based on the existing population. The fitness of each individual is evaluated based on its performance on the training dataset. The best individual (tree) is determined based on its fitness and is stored as the best tree so far. The process continues for the specified number of generations.

   By replacing the entire population in each generation, the generational replacement strategy allows the algorithm to explore the search space and potentially discover better solutions over time. The best individual from each generation (the best tree) is preserved, ensuring that the algorithm does not lose the best solution found so far.

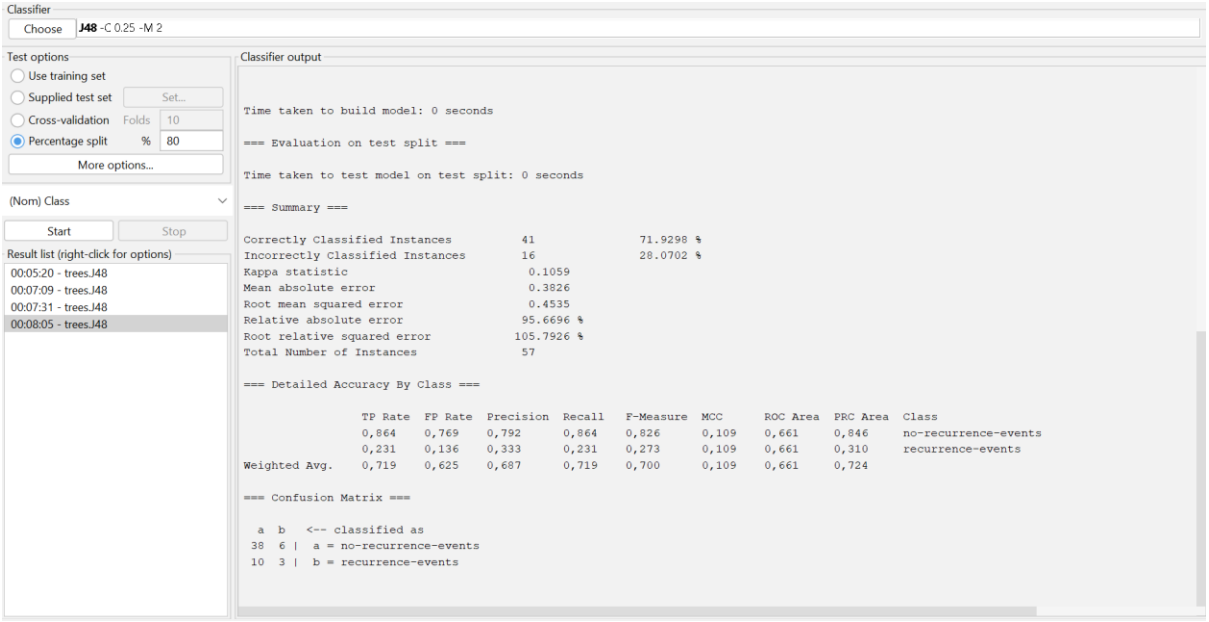
7. **Termination Condition:** The termination condition in my implementation is a fixed number of generations (50). The evolve method runs for a specified number of generations, which is passed as a parameter to the method.

TASK 3: WEKA C45 Decision Tree

My first Step was to import the breast-cancer dataset into Weka. I then used the J48 decision tree method as a classifier. I used the 80:20 percentage split where 80% will be used to train the model and 20% for the testing model. Here are the results :



```
=== Run information ===

Scheme:       weka.classifiers.trees.J48 -C 0.25 -M 2
Relation:     breast-cancer-data-weka.filters.unsupervised.attribute.Reorder-R1,3,4,5,6,7,8,9,10,2-weka.filters.unsupervised.attribute.Re
Instances:    286
Attributes:   10
              menopause
              tumor-size
              inv-nodes
              node-caps
              deg-malig
              breast
              breast-quad
              irradiat
              age
              Class
Test mode:    split 80.0% train, remainder test

=== Classifier model (full training set) ===

J48 pruned tree
------------------

node-caps = no: no-recurrence-events (228.39/53.4)
node-caps = yes
|   deg-malig = 1: recurrence-events (1.01/0.4)
|   deg-malig = 2: no-recurrence-events (26.2/8.0)
|   deg-malig = 3: recurrence-events (30.4/7.4)

Number of Leaves  :     4

Size of the tree :      6
```



Algorithms			
	Accuracy %	F Measure	Time (seconds)
Neural Network			
Genetic Programming	68.88888888888889	0.815	0.0057293
Decision trees Weka	71.9298	0.7	0

COMPARATIVE ANALYSIS

We can make the following observations:

1. Accuracy:
    - The J48 implementation in Weka achieved a slightly higher accuracy of 71.93% compared to the Genetic Programming approach, which achieved an accuracy of 68.89%.
    - This indicates that, in this specific analysis, the decision tree model built using the J48 algorithm in Weka performed slightly better in terms of accuracy.
2. F-Measure:
    - The Genetic Programming approach achieved a higher F-Measure of 0.815, indicating a better balance between precision and recall compared to the F-Measure of 0.7 obtained by the J48 implementation in Weka.
    - F-Measure is a measure that combines precision and recall, providing insight into the model's performance in handling both false positives and false negatives.
3. Execution Time:
    - The Genetic Programming approach took 0.0057293 seconds, while the J48 implementation in Weka took 0 seconds.
    - Although the execution time difference seems negligible, it's worth noting that the Genetic Programming approach took some computational time to evolve and optimize the decision tree population.

Overall, while the J48 implementation in Weka achieved slightly higher accuracy, the Genetic Programming approach showed a higher F-Measure, suggesting a better balance between precision and recall. However, the execution time for both approaches is comparable, with the J48 implementation having a slight advantage in terms of efficiency.