Part One: Neural Networks for Classification

Question One:

Number of layers: 3

Units in input layer: 4

Units in hidden layer: 3

Units in output layer: 3

In this assignment we were to assume that there is only going to be one hidden layer. This means we have a total of three layers, one input, one hidden and one output. This is because for this particular problem, one hidden layer is sufficient. Adding more layers would just add unnecessary complexity and take more epochs to get meet the *termination criteria*.

I decided on having four **input nodes** as there are four different properties of the Iris. This means we can utilise all four off these attributes to give us the most accurate classification output. I have three output nodes, as that gives us one node for each of the potential classes the Iris could be classified as, and it performed the best from carrying out tests with various values for the output.

In terms of the node in the **hidden layer**, according to Jeff Heaton, author of Introduction to Neural Networks, "the optimal size of the hidden layer is usually between the size of the input and size of the output layers". [1] In my case, I found having three units in the hidden layer performed the best. The more nodes in the hidden layer, the more likely you are to overfit the training data, this is discussed further in Question Four.

Question Two:

Learning rate: 0.2

Momentum: 0.0

Random range: 0.5

For the **learning rate**, a good place to start is 0.2. If the learning rate is too high, then the weights will diverge and never find the ideal values. If it is too low then then the algorithm will be very slow. As you can see 0.2 is a good middle group and works well.
The **momentum** changes the learning rate as it progresses, in this case momentum wasn't needed as from my tests adding momentum didn't increase the accuracy or decrease epochs.
The **random range** means the weights start as a random value between -0.5 and 0.5. This isn't very important as they will very quickly change anyway.

Question Three:

Stopping classification accuracy: 101.0%
Critical error: 0.002

By setting the termination **classification accuracy** to be over 100% it enables us to refine the termination criteria to just be the critical error, as 101% accuracy will never be reached. When training the algorithm, it can very quickly achieve 100% accuracy on the training data (approximately 50 epochs). However, at this point the mean squared error (mse) is 0.025, which just isn't good enough. If we stopped training at this point when we run the test data on the neural network we get 16/75 incorrectly predicted (mse of 0.086). By terminating using the **critical error** instead of the percentage, we can achieve a lot more accurate neural network. In this case we trained until we get a **critical error** of 0.002 which give an average of 2/75 incorrect on the test data, as shown in Figure 2

Question Four:

I simulated ten runs of the training and testing the neural network and produced a table with the results and averages below. This is using 75 instances for the training, and another 75 for the testing.

10 Trials from Training and Testing the Neural Network

Trial	Training MSE	Num Incorrect Training	Test MSE	Num Incorrect (Test)
1	0.001	0	0.018	2
2	0.005	1	0.014	2
3	0.008	1	0.011	2
4	0.000	0	0.021	3
5	0.015	2	0.010	1
6	0.000	0	0.019	2

Trial	Training MSE	Num Incorrect Training	Test MSE	Num Incorrect (Test)
7	0.001	0	0.019	2
8	0.000	0	0.020	2
9	0.000	0	0.022	3
10	0.015	2	0.010	1
Avg	**0.005**	**0.6**	**0.016**	**2**

Figure 1: Shows the 10 independent trials and how they performed. The mean average for each column is shown also.

I found it interesting that trials that had the best performing training, resulted in the worst performance on the test data. I predict this to be caused from overfitting of the data. To try and reduce this, I set the nodes in the hidden layer to be as few as possible (three). As mentioned in Question one, the optimal nodes in the hidden layer should be between the size of the input and output nodes. In my case the input size is four, and output is three- so further reducing the number of hidden layers nodes will decrease the performance of the algorithm.

Overall the average number of incorrect classifications was 0.6 on the training data with a mean squared error of 0.005, which is considered pretty good. On the un-seen (test) data, the neural network averaged 2/75 incorrect classifications. This gives us an accuracy of 97.33% and a mse of 0.016.

Question Five:

To compare the neural network to the nearest neighbour method its important to have the same training and test data. I ensured this was the same and the preceded with the tests.

K Nearest Neighbour vs Neural Network in Terms Of Accuracy

Method	Training Accuracy	Test Accuracy
K Nearest Neighbour (Consistent)	97.3%	92.0%
Neural Network (Average of 10 trials)	99.2%	97.3%

Figure 2: Shows the Neural Network out performing the K Nearest Neighbour on training and testing

The K-nearest-neighbour generated consistent results every time it was run, as for the neural network- it varied every time because it always had different weights. Therefore, I took the average of 10 trials to ensure I got an accurate measurement of its performance.

As you can see the Neural network outperforms the K nearest neighbour hugely in terms of accuracy. There was ~2% variance in accuracy between the two algorithms on the training (already learnt) data. However when they were to be performed on unseen data (test) the neural network achieved an astonishing 5.3% more accuracy. This is the most important accuracy we want to compare, as when its used in the real world we wont be re-classifying the known training data. Instead, it will be trying to classify un-seen (test) data. If we were to decide on an algorithm to use in the real world, without a doubt we would want to use the neural network as its considerably more accurate.

Another attribute we have to take into consideration is processing time. For the Neural network it requires 1373 epochs to effectively learn the training data, where as for every iris to learn in k-nearest-neighbour it requires n distance comparisons, which gives it the complexity O(n2) for classifying training data, and the same for testing. This makes the neural network a lot more efficient, especially on large data sets. Say there were 1000 instances in the training set, for the n-nearest-neighbour that is going to require at least 1000^2 = 1,000,000 comparisons just to classify one instance. Where as for the neural network, it simply needs to provide the inputs of the instance, traverse the neural network and receive an outputted classification. This is another reason why the neural network is more preferable.

Part 2: Genetic Programming for Symbolic Regression

Question One:

Terminal Set:

- Variable "X"
- Terminal (an Integer between two and ten)

Question Two:

Function Set:

- Multiply
- Divide
- Subtract

- Add
- Power

Question Three:

I created a Fitness Function to evaluate the performance of a program. I did this by iterating through the length of the inputs, in this case its 20. For every input I set the variable, x, to be the x[i] value, as this means when the program is evaluated it will use a value that we know produces a specific output. We can the run the program and see what it determines y to be. This can then be compared to the true value of y (y[i]). Using this difference we can sum the total error over all 20 programs and get an estimation of how it performs. The lower the total error (return value) the better the program performs.

Pseudocode:

```
x[] =  inputed x values
y[] = inputed y values
variableX = variable "x" in the node set.
totalError = 0

for i in 0.. inputs.length:
        variableX = x[i]
        result = program.execute()
        totalError += Abs(result - y[i])
if  totalError < MIN_ACCEPTABLE_ERROR:
         return 0
return totalError
```

Question Four:

Configuration Properties:

Max Init Depth : 4
Population Size: 1000
Max Crossover Depth: 8
Strict Program Creation: true
Crossover Prob: 0.9

Mutation Prob: 35.0
Reproduction Prob: 0.2

The **max init depth** is typically set to 4 in most GP programs, as it's a good starting point.
I found a **population size** of 1000 to work well for this particular application, as its a large enough populate to generate satisfactory results, but isn't too big.
Setting an incorrect **crossover depth** can lead to a bias which will have a negative effect on the result, however I found 8 to work good. After discussing with my class mates, they found it to work well too.
A **crossover probability** of 0.9 allowed for a good *mate* which was enough to generate new formulas from two candidates in the last generation to be added to the candidate pool. It's recommended to use 0.9 for most applications.
We want **strict program creation** incase there is a problem with evolving, and the solution doesn't contain a terminal value (X) we want it to throw an error instead of trying to use new, random values. Although this isn't very important as the function always contains X as it leads to the most successful programs anyway.
Having a **mutation probability** of 35 was very tricky to find. I found there was a very fine line between generating functions that are too random, and having enough variation in the candidates. When the reproduction probability is too small, it doesn't allow for much variation and the evolution would sometimes stick at a good (but not the best) solution without further evolving.
Having a **reproduction probability** of 0.2 allows a copy of a some solutions to be added to the candidate pool, which essentially ensures that it will continue to the next generation.

Termination Criteria:

Max Evolutions: 1000
Minimum Acceptable Error: 0.001 (0.1%)

The termination criteria I used was primarily based on the **minimum acceptable error** (0.001). I choose this over the **maximum evolutions** as it is a lot more precise way of controlling the programs accuracy as it will stop evolving once it has fully developed. It is "fully developed" once it provides an accurate (within 0.1% across all 20 inputs) transformation of x to y using the best program. The maximum evolutions is never reached, but is there to provide a *safeguard* incase the training got stuck at a particular accuracy. Without a maximum number of evolutions it would continue evolving infinitely.

Question Five:

Below is three outputted functions from my Regression solver.

Three Outputted Functions

Trial	Evolutions	Fitness	Function
1	42	0.0	(3.0 / 3.0) + (((X * X) - X) * ((X * X) - X))
2	115	0.0	(((X * X) - X) * ((X * X) - X)) + 1
3	59	0.0	((X - 1)* X)^2.0 + 1
Avg	**72**	**0.0**	-

Figure 3: Showing the three best outputted functions that are on average evolved in 72 generations

Question Six:

I am going to examine the best function from Question Five and reveal how and why it can solve the problem at hand. The best program achieved was: ((X - 1)* X)^2.0 + 1. This can be expanded out to y = $x^4 - 2x^3 + x^2 + 1$.

21 Trials of Testing The Regression Program

Input	X	Y	Program Determined Y	Difference
1	-2.00	37.00000	37.00000	0.00000
2	-1.75	24.16016	24.16016	0.00000
3	-1.50	15.06250	15.06250	0.00000
4	-1.25	8.91016	8.91016	0.00000
5	-1.00	5.00000	5.00000	0.00000
6	-0.75	2.72266	2.72266	0.00000
7	-0.50	1.56250	1.56250	0.00000
8	-0.25	1.09766	1.09766	0.00000
9	0.00	1.00000	1.00000	0.00000
10	0.25	1.03516	1.03516	0.00000
11	0.50	1.06250	1.06250	0.00000
12	0.75	1.03516	1.03516	0.00000
13	1.00	1.00000	1.00000	0.00000
14	1.25	1.09766	1.09766	0.00000
15	1.50	1.56250	1.56250	0.00000

Input	X	Y	Program Determined Y	Difference
16	1.75	2.72266	2.72266	0.00000
18	2.00	5.00000	5.00000	0.00000
19	2.25	8.91016	8.91016	0.00000
20	2.50	15.06250	15.06250	0.00000
21	2.75	24.16016	24.16016	0.00000

So as we can see, if measured to 5 d.p. (accuracy of the inputted data) the program produces the exact output. After 5 d.p. it's not going to be exactly accurate e.g. expected y value is 24.16016 but program produces 24.16015625, however there is not a lot we can do about that. We can only be as accurate as the information given to us.

We assume that because it can solve all 20 examples accurately in the training set, unseen x values in the test set would yield similar results.

Part 3: Genetic Programming for Classification

Question One:

Terminal Set:

- Terminal (an Integer between negative one and ten)
- Patient Attributes

Patient Attributes:

- Clump Thickness
- Uniformity of Cell Size
- Uniformity of Cell Shape
- Marginal Adhesion
- Single Epithelial Cell Size
- Bare Nuclei
- Bland Chromatin
- Normal Nucleoli
- Mitoses

Question Two:

Function Set:

- Multiply
- Divide
- Subtract
- Add
- Power

Question Three:

I created a Fitness Function to evaluate the performance of a program. I did this by iterating through the length of the training data, in this case its 349. For every training patients, I set the all the patient attributes in the terminal set to be the corresponding value of the patient. This means when the program is evaluated it will use a value that we know produces a specific outcome (classification). We can then run the program and see what it determines the patients classification to be. If its < 0, we assume it predicted it to be class 2, else its class 4. This result can then be compared to the true classification of the patient to see if it was correctly classified.

Using the number of correctly calculated patients we can transform this into a percentage and return that as the result of the fitness function. The higher the percentage is (return value) the better the program performs, and the more likely the program is to be selected to evolve.

Pseudocode:

```
patients =  patients to classify
programVariables = represents the 9 attributes of the patient in the terminal set
correct = 0
for i in 0.. patients.length:
        programVariables = patient.attributes
        result = program.execute()
        if result < 0:
                predictedClass = 2
        else:
                predictedClass = 4
        if predictedClass == patient.class:
                correct++
return correct / patients.length
```

Question Four:

Configuration Properties:

Max Init Depth : 4

Population Size: 1000

Max Crossover Depth: 6

Crossover Prob: 0.9

Strict Program Creation: true

Mutation Prob: 0.2

Reproduction Prob: 0.05

The **max init depth** is typically set to 4 in most GP programs, as it's a good starting point.
I found a **population size** of 1000 to work well for this particular application, as its a large enough population to generate satisfactory results, but isn't too big.
Setting the incorrect **max crossover depth** can lead to a bias which will have a negative effect on the result, however I found 6 to work just fine.
A **crossover probability** of 0.9 allowed for a good *mate* which was enough to generate new formulas from two candidates in the last generation to be added to the candidate pool. It's recommended to use 0.9 for most applications.
We want **strict program creation** incase there is a problem with evolving, and the solution doesn't contain a terminal value we want it to throw an error instead of trying to use new, random values.
Having a **mutation probability** of 0.2 was very tricky to find. I found there was a very fine line between generating random functions, and having enough variation. When the reproduction probability is too small, it doesn't allow for much variation and the evolution would sometimes stick at a good (but not the best) solution without further evolving.
Having a **reproduction probability** of 0.05 allows a copy of a some solutions to be added to the candidate pool, which essentially ensures that it will continue to the next generation.

Termination Criteria:

Max Evolutions: 1000

Minimum Acceptable Error: 0.03 (3%)

The termination criteria I used was primarily based on the **minimum acceptable error** (0.03). I choose this over the **maximum evolutions** as it is a lot more precise way of controlling the programs accuracy as it will stop evolving once it has fully developed, and is very unlikely to further develop. It is "fully developed" once it provides an accurate (within 3% across all patients) prediction of the class of the patient, using its nine attributes. The maximum evolutions should never be reached, but is there to provide a *safeguard* incase the training got stuck at a particular accuracy and would continue evolving infinitely. Without a maximum number of evolutions it would continue evolving infinitely.

Question Five:

I split the training and test data equally with 50% of the data in each. I found a 50% split to be the most useful, as decreasing the size of the training data further resulted in a lower performing algorithms, where as increasing it resulted in no noticeable increase in accuracy. It also enabled me to have a large, diverse set to *test* the algorithm on. If I made the training set too large that would have a negative effect on the test set, and when the algorithm is tested on un-seen data we wouldn't be able to get accurate representation of how it *actually* performs. If we can't get an accurate idea of how it performs then we have no idea of how it will work once it was deployed in a real-life situation.

Question Six:

Accuracy on Test and Training Data

Trial	Evolutions	Training Fitness	Test Fitness
1	51	97.1347%	97.1347%
2	19	97.1347%	96.2751%
3	93	97.1347%	96.2751%
4	215	97.1347%	97.1347%
5	97	97.7077%	97.4212%
6	226	97.4212%	97.4212%
7	42	97.4212%	95.7020%
8	199	97.1347%	97.4212%
9	189	97.1347%	96.5616%
10	48	97.4212%	96.5616%
Average	**118**	**97.2780%**	**96.7908%**

Question Seven:

Solution One (97.1347% on training, 97.4212% on test, fitness value 0.02865)

Uniformity of Cell Shape + ((((Uniformity of Cell Size + Marginal Adhesion) * ((Clump Thickness - 5.0) + Bare Nuclei)) - 8.0) - 5.0)

Solution Two (97.1347% on training, 97.4212% on test, fitness value 0.02865)

(Marginal Adhesion / Bare Nuclei) + (Uniformity of Cell Size + ((((Clump Thickness - 4.0) + (Bare Nuclei - 6.0)) + Uniformity of Cell Shape) - 4.0))

Solution Three (97.4212% on training, 97.4212% on test, fitness value 0.02578)

Clump Thickness - (5.0 - ((Bare Nuclei * Marginal Adhesion) - (5.0 - ((Uniformity of Cell Shape + Bland Chromatin) - 6.0))))

Question Eight:

I am going to analyse Solution One from Question Seven to explain how it can solve the problem.

Function:

Uniformity of Cell Shape + ((((Uniformity of Cell Size + Marginal Adhesion) * ((Clump Thickness - 5.0) + Bare Nuclei)) - 8.0) - 5.0)

Proof:

Proving Solution One Works Using Three Patients

Trial	Patient #1	Patient #350	Patient #699
Clump Thickness	5	4	4
Uniformity of Cell Size	1	2	8
Uniformity of Cell Shape	1	3	8
Marginal Adhesion	1	5	5
Single Epithelial Cell Size	2	3	4
Bare Nuclei	1	8	5

Trial	Patient #1	Patient #350	Patient #699
Bland Chromatin	3	7	10
Normal Nucleoli	1	6	4
Mitoses	1	1	1
Result From Formula	-10	39	47
Predicted Classification	**2**	**4**	**4**
True Classification	**2**	**4**	**4**

Note that the predicated classification is based on whether the result is < 0.

Pseudocode:

```
result = calculation of formula with patient values substituted in
if result < 0:
    classification = 2
else:
    classification = 4
```

Result:

Therefore we can conclude that this function is successful as testing in on three patients (first, last and one in-between) yields correct classifications of the patient.

References

1. Heaton, J. (2008). Introduction to Neural Networks for Java, Second Edition. 1st ed. St. Louis, Mo: Heaton research.