Part 1:

a) Report the output and predicted class of the first instance in the dataset using the provided weights

```
[[-0.28 -0.22]
 [ 0.08  0.2 ]
 [-0.3   0.32]
 [ 0.1   0.01]]
First instance has label Adelie, which is [0] as an integer, and [1. 0. 0.] as a list of outputs.

Predicted label for the first instance is: ['Chinstrap']
```

b) Report the updated weights of the network after applying a single back-propagation based on only the first instance in the dataset.

```
epoch =  0
acc =  0.5149253731343284
Weights after performing BP for first instance only:
Hidden layer weights:
 [[-0.26293231 -0.38229496]
 [ 0.18978343  0.4868215 ]
 [-0.17585303  0.15099507]
 [ 0.23102589 -0.11557548]]
Output layer weights:
 [[-0.97267967 -0.50843807 -0.11240592]
 [-0.66959968 -0.6057472  -0.65137609]]
```

c) Report the final weights and accuracy on the test set after 100 epochs. Analyse the test accuracy and discuss your thoughts.

Results shown below. The accuracy at the end of the training caps out at 82.83582089552238%.

The test accuracy is 81.53846153846154%. While the results are good, it is clear that our neural network could still be improved significantly. The results are somewhat expected though, as we are not updating our bias.

Note: Screenshots are taken before implementing biases.

```
In [3]: # TODO: Train for 100 epochs, on all instances.
        nn.train(instances, onehot_encoded, 100)
        print('\nAfter training:')
        print('Hidden layer weights:\n', nn.hidden_layer_weights)
        print('Output layer weights:\n', nn.output_layer_weights)
```

```
acc =  0.8283582089552238
epoch =  96
acc =  0.8283582089552238
epoch =  97
acc =  0.8283582089552238
epoch =  98
acc =  0.8283582089552238
epoch =  99
acc =  0.8283582089552238

After training:
Hidden layer weights:
 [[ 0.9386876  -9.84076342]
 [-7.30590284  5.21616974]
 [ 2.3914977  -1.4072903 ]
 [ 2.47644876  1.44156165]]
Output layer weights:
 [[ -9.71047409  -2.44670946   3.24345684]
 [  4.91368601  -2.87797267 -11.68759154]]
```

```
Accuracy on test set:
acc =  0.8153846153846154
```

d) Discuss how your network performed compared to what you expected. Did it converge quickly? Do you think it is overfitting, underfitting or neither?

My network converged relatively quickly, hitting ~80% after only 6 epochs.

I don't think it is overfitting much at all, as the accuracy on the test set has a similar accuracy to the training set. Considering it is implemented without biases, the performance is relatively similar to what I expected.

1.1) Accuracy of 93.84615384615385%

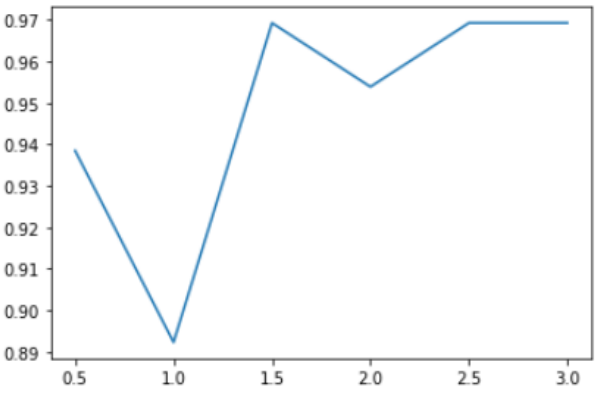
Evidently, the accuracy is much higher, more than 10% higher than the test accuracy without bias nodes implemented. The accuracy is much higher as adding bias nodes allows the neural network to shift the output of the activation function left or right

```
Accuracy on test set:
acc =  0.9384615384615385
```

1.2) Sensitivity testing
Results below.
As shown, test accuracy caps out at about 97% for learning rates of 1.5, 2.5 and 3.0.
Learning rates of 0.5, 1.0 are much lower, reaching as low as only 89%.

```
In [10]: import matplotlib.pyplot as plt
         #Plot of learning rates to Test accuracy
         plt.plot(learning_rates, testAcc)
```

```
Out[10]: [<matplotlib.lines.Line2D at 0x29b98130fa0>]
```



Part 2

a) Justify the terminal set you used for this task.

I use the terminals of {-1, 0, 1}. This is because without the terminals, my output function was often just 1 above or below the actual function, but with the terminals the adjustment makes the function very close.

b) Justify the function set you used for this task.

The function set is the np add, multiply, subtract and negative functions, as represent the functions that were given us in the assignment. I left out the divide function, as that caused errors in my code when division by 0 occurs.

c) Formulate the fitness function and describe it using plain language (and mathematical formula, or other formats you think appropriate, e.g. good pseudo-code).

```
def evaluateRegression(individual, points, pset):
    func = gp.compile(expr=individual, pset=pset)
    sqerrors = (func(points)-yVals)**2
    return (numpy.sqrt(numpy.sum(sqerrors) / len(points)),)
```

The fitness function I used is shown above.

The points array is the set of 20 x values between -2 and 2.75 that were given to us in regression.txt. yVals is the set of y values given in regression.txt.

Thus, sqerrors is the squares of the difference between each point given by the function we are evaluating and the actual yVals, and the fitness is determined by taking the square root of the average of those sqerrors.

In more readable pseudo-code:

```
fitness(f, x, targetY) {
    errors = (f(x) - targetY) ** 2
    return sqrt(sum(errors) / len(x))
```

d) Justify the parameter values and the stopping criteria you used.

cxpb – The probability of mating two individuals. Set to 0.5

mutpb – The probability of mutating an individual. Set to 0.2

I chose 0.5 and 0.2 after playing around with them for a little bit through trial and error. Increasing them any further and the resulting equations end up very long due to excessive mutations and mating, whilst any lower decreases accuracy.

ngen – The number of generations

I chose 50 for this, as it was a number that consistently gave a good equations, and also wasn't too high and thus won't take too long to run

e) Different GP runs will produce a different best solution. List three different best solutions evolved by GP and their fitness values (you will need to run code several times with different random seeds).

Equation 1:

add(multiply(add(-1, ARG0), multiply(ARG0, subtract(multiply(ARG0, ARG0), ARG0))), add(multiply(0, multiply(multiply(add(ARG0, ARG0), multiply(ARG0, subtract(multiply(ARG0, ARG0), ARG0))), subtract(multiply(ARG0, multiply(0, ARG0)), add(-1, ARG0)))), multiply(1, 1)))

Fitness for 1: 2.651650429537579e-06

Equation 2

add(add(subtract(ARG0, subtract(ARG0, 1)),
multiply(subtract(multiply(subtract(add(subtract(multiply(subtract(add(0, -1),
negative(add(negative(subtract(ARG0, ARG0)), ARG0))), ARG0),
add(add(multiply(subtract(ARG0, 0), subtract(ARG0, multiply(add(0,
negative(negative(subtract(multiply(subtract(add(subtract(ARG0, ARG0), -1),
negative(add(negative(1), ARG0))), ARG0), -1)))), multiply(ARG0, ARG0)))), 0), -1)), -1),
negative(add(negative(add(0, -1)), ARG0))), ARG0), ARG0), 0)), multiply(add(0,
negative(negative(subtract(multiply(subtract(add(0, -1), negative(add(negative(1), ARG0))),
ARG0), negative(1))))), multiply(ARG0, ARG0)))

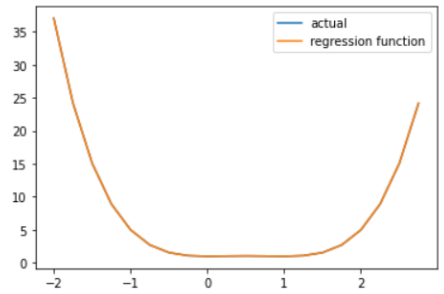
Fitness (2.651650429537579e-06,)

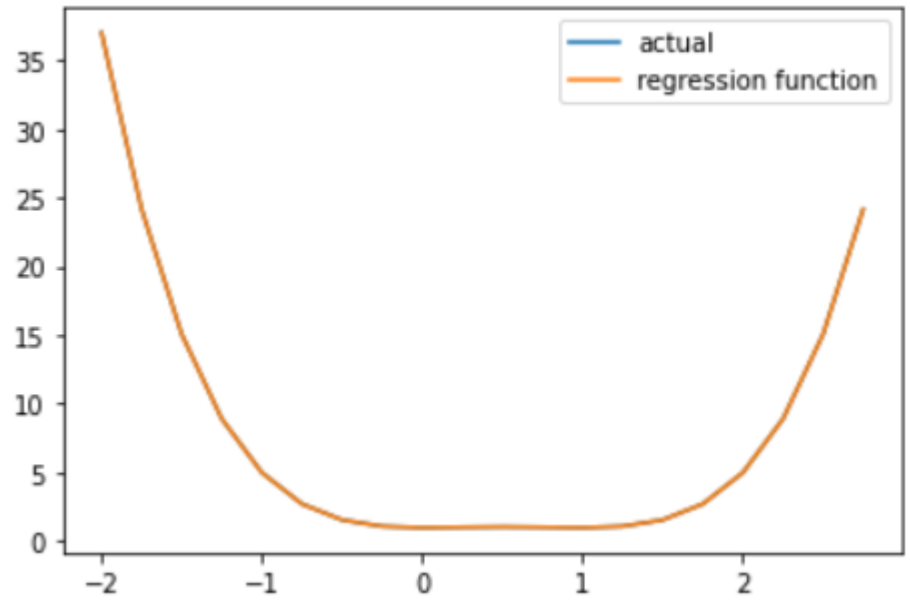
Equation 3:

subtract(subtract(add(ARG0, 1), multiply(add(0, ARG0), ARG0)), multiply(add(-1,
negative(multiply(subtract(ARG0, 1), ARG0))), multiply(subtract(ARG0, 1), ARG0)))

Fitness (2.651650429537579e-06,)

```
add(multiply(add(-1, ARG0), multiply(ARG0, subtract(multiply(ARG0, ARG0), ARG0))), add(multiply(0, multiply(multiply(add(ARG0,
ARG0), multiply(ARG0, subtract(multiply(ARG0, ARG0), ARG0))), subtract(multiply(ARG0, multiply(0, ARG0)), add(-1, ARG0)))), mul
tiply(1, 1)))
Fitness (2.651650429537579e-06,)
```






```
subtract(subtract(add(ARG0, 1), multiply(add(0, ARG0), ARG0)), multiply(add(-1, negative(multiply(subtract(ARG0, 1), ARG0))), m
ultiply(subtract(ARG0, 1), ARG0)))
Fitness (2.651650429537579e-06,)
```