Assignment 1 Part 1: Supervised Learning

Student Name: Luca J. Ricagni

Student Number: 200894968

Module Name: Machine Learning

Module Code: ECS708U/ECS708P

Due Date: 09/11/2020

Contents

Task 1 – Linear regression with one variable 3
Task 2 – Linear regression with multiple variables 7
Task 3 – Regularized linear regression 10
Appendices 13
Appendix 1 – Table 1 13
Appendix 2 – Table 2 13

Task 1 – Linear regression with one variable

The original *calculate_hypothesis* function simply set the hypothesis to 0 and returned it. The first part of task 1 was to modify this function in order to use the correct hypothesis function shown in equation 1.

$$h_0(x) = \theta_0 x_0 + \theta_1 x_1 \quad (1)$$

This can be done in one of two ways; either by simply computing the function in a single line shown in Figure 1, or in a more generic way using a loop as shown in Figure 2.

```
hypothesis = X[i, 0] * theta[0] + X[i, 1] * theta[1]
```

Figure 1 – Code to calculate the hypothesis using a single line

```
hypothesis = 0.0
for j in range(0, len(theta)):
    hypothesis += X[i, j] * theta[j]
```

Figure 2 – Code to calculate hypothesis using a loop

Using this more generic method has the advantage of the *calculate_hypothesis* function working for any number of values for θ and x.

The next stage was to modify the *gradient_descent* function to use the new *calculate_hypothesis* function rather than simply doing the calculation itself. This could be done with the call shown in Figure 3. This call can be found on lines 38 and 54 of gradient_descent.py.s

```
hypothesis = calculate_hypothesis(X, theta, i)
```

Figure 3 – Shows code to call the calculate_hypothesis function

Having now successfully modified the code to use the correct calculations the next stage was to adjust the value of the learning rate (alpha) in order to get the best possible fit. If the program is run with the high value of alpha = 1 the minimum cost is 172570.09522. The graphic output is shown in Figure 4.

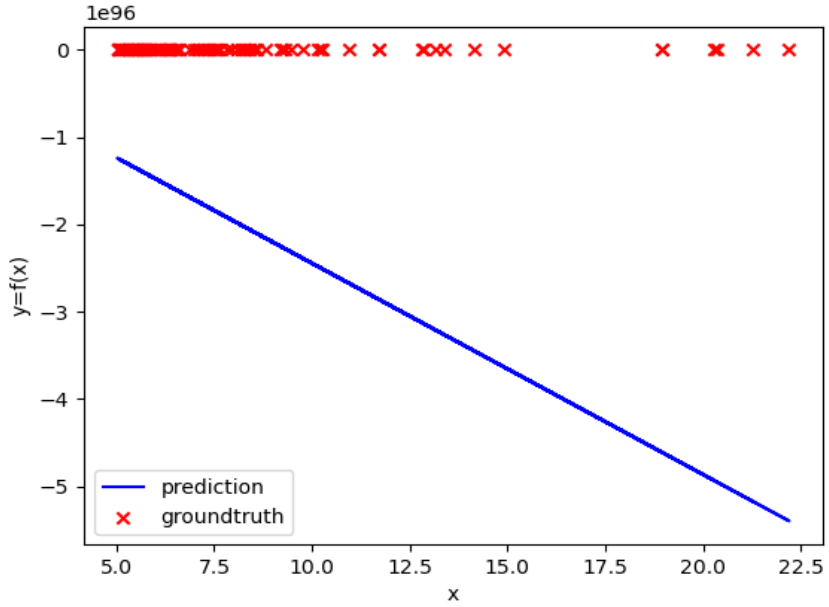

Figure 4 – Shows plot of hypothesis with alpha = 1 and 50 iterations

This is obviously incorrect for two main reasons: the cost is far too high, which can also be seen in Figure 5, and the prediction (blue) line in Figure 4 has no correlation to the ground truth (training data) shown in red crosses. Furthermore, the huge spike in cost seen around iteration 48 in Figure 5 after it has been flatlined is indicative of the fact that the system never properly fit the data.

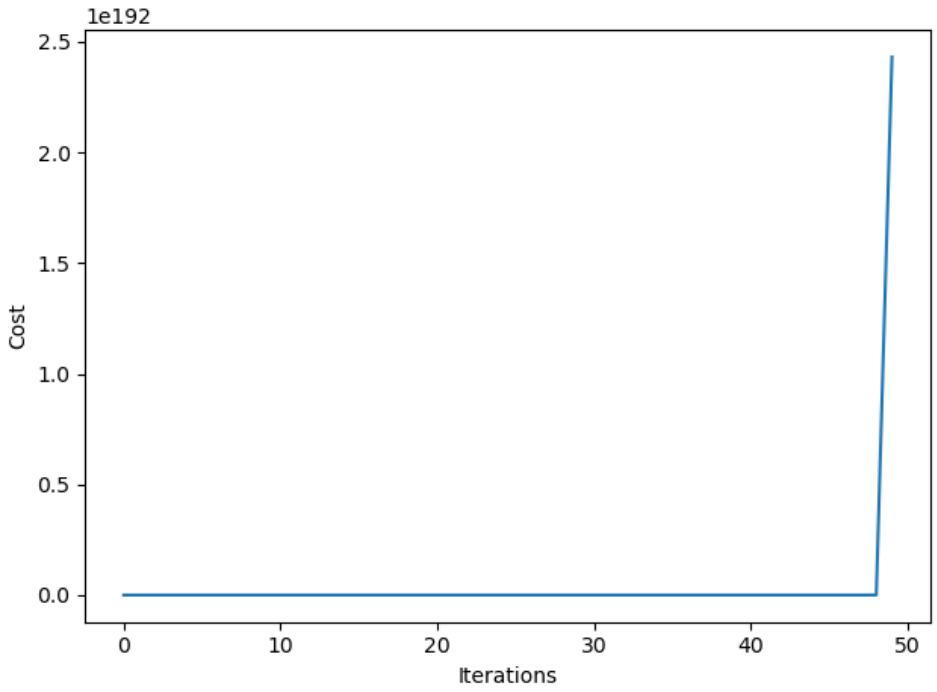

Figure 5 – Shows plot of the cost with alpha = 1 and 50 iterations

If the program is run using a very low value of alpha, 0.0000001 for instance, the minimum cost is 32.05123 and the graphic output is shown in Figure 6. The cost graph is shown in Figure 7.

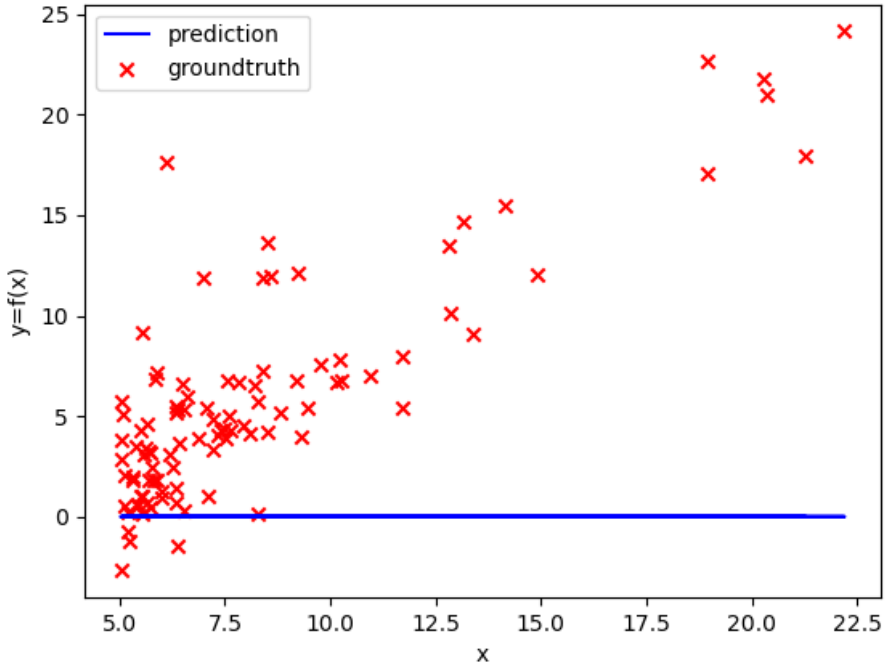

Figure 6 – Shows plot of hypothesis with alpha = 0.0000001 and 50 iterations

This output is better than using a high value of alpha because, while the prediction does not match the ground truth, it is closer than before as evidenced both visually and by comparing the relevant costs.

The best value of alpha was found to be in the range of 0.001-0.0235, as they all converged onto the training data with a cost in the range 5-6, with higher alpha values converging faster.

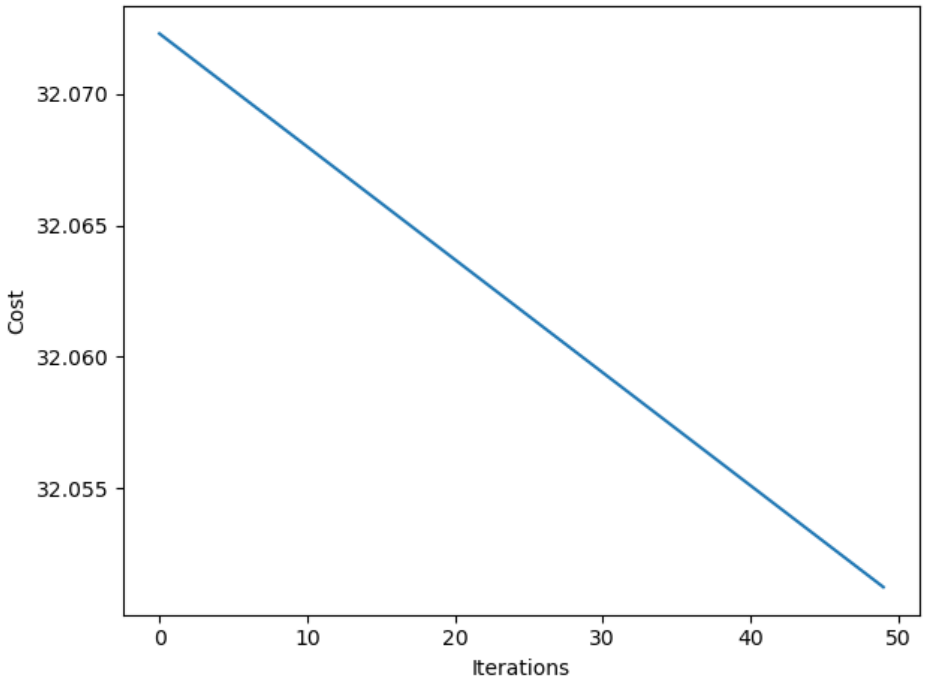

Figure 7 – Shows plot of cost with alpha = 0.0000001 and 50 iterations

Figures 8 & 9 show the hypothesis and cost plots when alpha = 0.01, the minimum cost is 5.67829. Using this alpha value, a fairly good hypothesis of the training data can be produced.

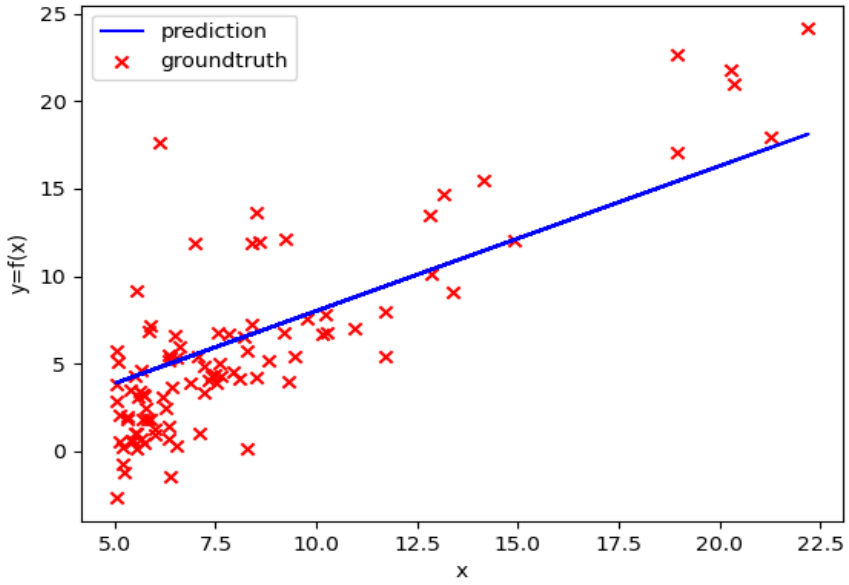

Figure 8 - Shows plot of hypothesis with alpha = 0.01 and 50 iterations

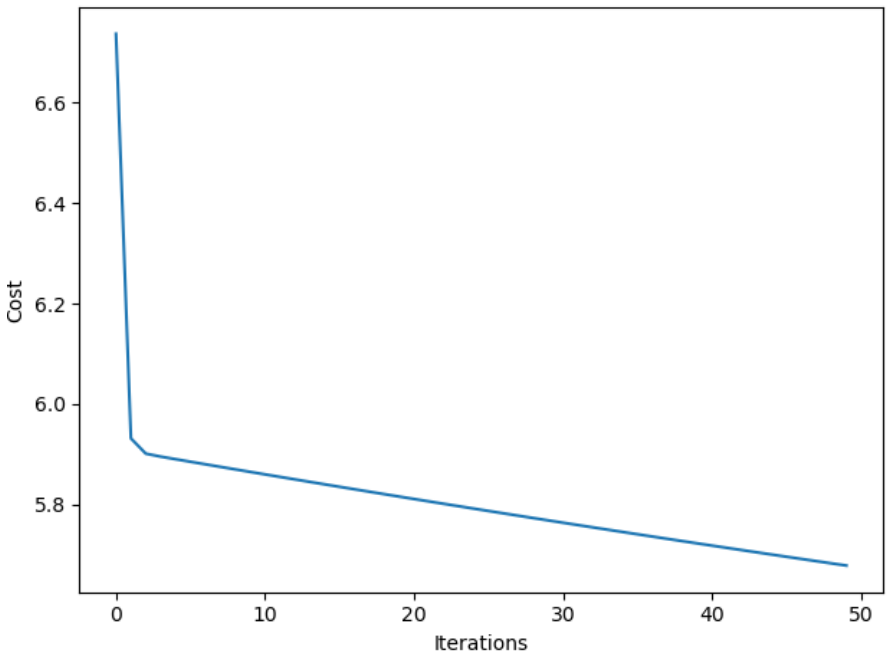

Figure 9 - Shows plot of cost with alpha = 0.01 and 50 iterations

Task 2 – Linear regression with multiple variables

This task is looking at how house prices related to the area of the house in square feet and the number of bedrooms.

The first part of this task was to again modify the *calculate_hypothesis* function in order to use the new hypothesis function shown in equation 2.

$$h_0(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 \quad (2)$$

Where x_0 is the bias term (set to 1 for all), x_1 is the area of the house in square feet and x_2 is the number of bedrooms in the house. This hypothesis function was implemented as shown in Figure 10.

```
hypothesis = 0.0  # Initialise hypothesis

for j in range(len(theta)):
    term = X[i, j] * theta[j]
    hypothesis += term
```

Figure 10 – Code to calculate new hypothesis function

This was done in the same way as Task 1 using a loop, this means that this function can be used for any number of variables. This function was then called on line 36 of *gradient_descent*:

Having now implemented the correct *calculate_hypothesis* function the program can now be run and outputs the graphs shown in Figure 11 & 12.

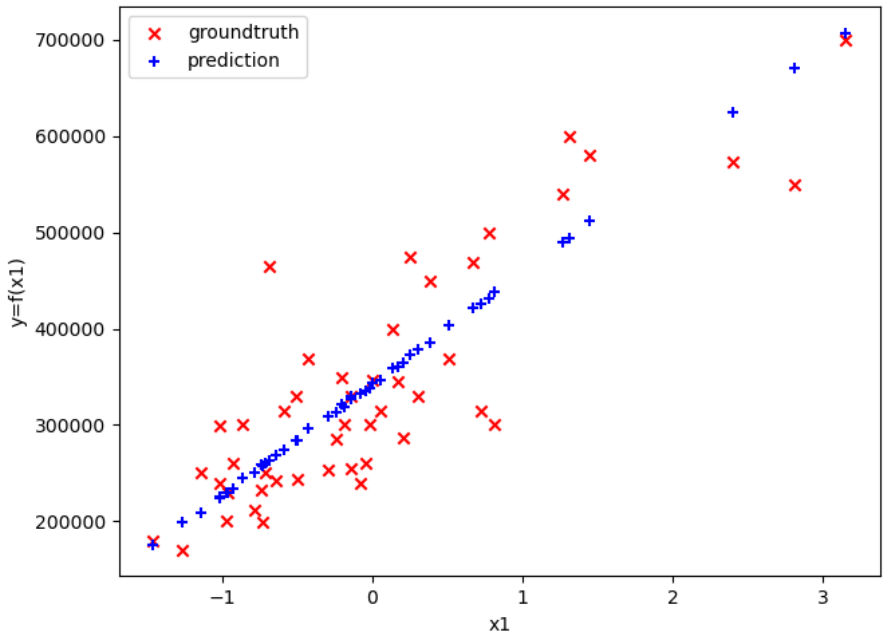

Figure 11 - Shows plot of hypothesis with alpha = 0.01 and 50 iterations

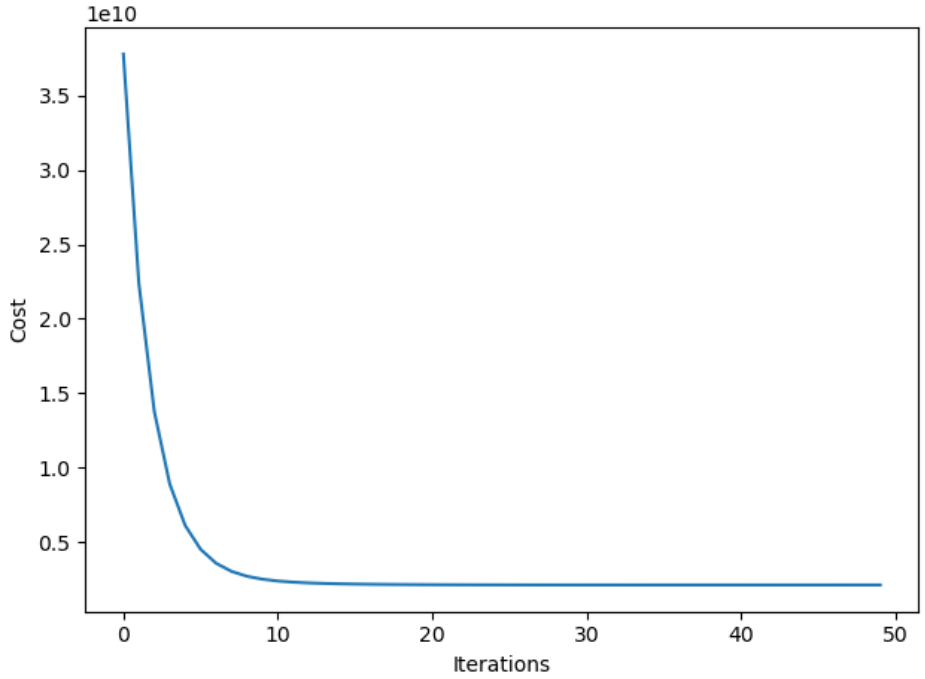

Figure 12 - Shows plot of cost with alpha = 0.01 and 50 iterations

Figure 11 shows the plot of hypothesis against the ground truth, visually this seems like a good approximation of the training data. As each row of data is given to the prediction it becomes more accurate, converging to the centre. While doing this it was observed that the more the prediction converged the slower it got. This is because the gradient descent algorithm works by finding the local minima by minimising the mean squared error, the step size is directly related to the rate of change of this error. Therefore, as it converges on the minima the gradient decreases, meaning that the step size decreases, which has the visual effect of slowing down the movement of the prediction.

After running the code, the final theta values observed are shown in Figure 13.

```
Theta_0:  343095.1359663433
Theta_1:  117002.04248758791
Theta_2:  -1819.7797110376812
```

Figure 13 – Shows outputted values of theta

This is surprising because Theta_2 is negative. This shows that there is a negative correlation between house prices and the number of bedrooms, i.e. a higher number of bedrooms lowers the price of a house. However, the value for Theta_2 has a significantly smaller magnitude than Theta_0 and Theta_1, meaning that while it is a negative correlation, it is a weak one.

The next stage of this task was to use the trained values of theta to make a prediction if given some new data. Two new samples were to be used, a house with an area of 1650 square feet and 3 bedrooms, and a house with an area of 3000 square feet and 4 bedrooms. The two samples where initialised as shown in Figure 14.

```
sample1 = [1650, 3]
sample2 = [3000, 4]
```

Figure 14 – Code to initialise samples

These samples must then be normalised using the saved values for the mean and standard deviation stored in *mean_vec* and *std_vec* and was accomplished like this:

In order for these samples to be passed to the hypothesis function a column of 1s had to be append to the data for the value of x_0. The code to perform this calculation can be found in Figure 15, while the values after normalization can be found in Figure 16.

```
normalized_sample1 = [1, (sample1[0] - mean_vec[0, 0]) / std_vec[0, 0], ((sample1[1] - mean_vec[0, 1]) / std_vec[0, 1])]
normalized_sample2 = [1, (sample2[0] - mean_vec[0, 0]) / std_vec[0, 0], ((sample2[1] - mean_vec[0, 1]) / std_vec[0, 1])]
```

Figure 15 – Code to calculate the normalized samples

```
Normalized sample 1:  [1, -0.4460438603276164, -0.22609336757768828]
Normalized sample 2:  [1, 1.2710707457752388, 1.1022051669412318]
```

Figure 16 – Both samples after normalization

Each normalized sample was then used to estimate and print the value of the property. This is shown in Figure 17.

```
estimate1 = ((normalized_sample1[0] * theta_final[0]) + (normalized_sample1[1] * theta_final[1]) + (normalized_sample1[2] * theta_final[2]))
print("Estimated value of 1st house is: ", estimate1)

estimate2 = ((normalized_sample1[0] * theta_final[0]) + (normalized_sample2[1] * theta_final[1]) + (normalized_sample2[2] * theta_final[2]))
print("Estimated value of 2nd house is: ", estimate2)
```

Figure 17 – Code to calculate and print each estimate from the normalized samples

The estimated values of the properties are shown in Figure 18.

```
Estimated value of 1st house is:  291318.53339208185
Estimated value of 2nd house is:  489807.23876806727
```

Figure 18 – The final estimates for each sample

It was expected that the second property would have a higher value than the first because although having an extra bedroom would lower the value, the area being almost double has a much higher weighting.

Task 3 – Regularized linear regression

This task deals with creating a model to fit data which is not linear. In order to do this a new hypothesis function must be used which is shown in equation 3.

$$h_0(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_2 x_1^3 + \theta_2 x_1^4 + \theta_2 x_1^5 \quad (3)$$

Similarly to the previous tasks this was implemented in the *calculate_hypothesis* function using a loop, meaning that it will work for any order of polynomial. This is shown in Figure 19.

```
hypothesis = X[i, 0] * theta[0]
for j in range(1, len(theta)):
    hypothesis += X[i, j] * (theta[j] ** j)
```

Figure 19 – Code to calculate new hypothesis function

Because the number of points the model is to be fit is very small (7 points) there is a high chance of overfitting, to accommodate this regularization must be applied. The cost function changes and is shown in equation 4.

$$J_0 = \frac{1}{2m}\left[\sum_{i=1}^{m}\left(h_0\left(x^{(i)}\right) - y^{(i)}\right)^2 + \lambda \sum_{j=1}^{n} \theta_j^2\right] \quad (4)$$

This function makes use of regularization parameter λ, which controls how much punishment is given for getting to close to the true values. Although this may seem counter intuitive it is important because the system may have a great deal of noise in it and fitting the model to these points would be overfitting. It is also worth noting that using regularization is likely to increase the cost, because the cost is minimised by being as close as possible to the data points and this will likely push it further away.

The next stage was to modify the *gradient_descent* function in order to use the *compute_cost_regularised* function instead of the *compute_cost* function. This was done using a simple function call as shown in Figure 20.

```
iteration_cost = compute_cost_regularised(X, y, theta, l)
```

Figure 20 – Code to call the regularised cost function

In order to accommodate this new *compute_cost_regularised* function some further changes need to be made to *gradient_descent;* it must now accept *l (λ) as an argument, shown in Figure 21.*

```
def gradient_descent(X, y, theta, alpha, iterations, do_plot, l):
    """
        :param X            : 2D array of our dataset
        :param y            : 1D array of the groundtruth labels of the dataset
        :param theta        : 1D array of the trainable parameters
        :param alpha        : scalar, learning rate
        :param iterations   : scalar, number of gradient descent iterations
        :param do_plot      : boolean, used to plot groundtruth & prediction values during the gradient descent iterations
        :param l            : scalar, used to represent lambda, used for regularization
    """
```

Figure 21 – The function header for gradient_descent with the new parameter l

In addition, the *gradient_descent* call in main must be modified to accept this new parameter as well. Shown in Figure 22.

```
theta_final = gradient_descent(X, y, theta, alpha, iterations, do_plot, l)
```

Figure 22 – Code to call gradient_descent function1

Now that the code has been updated to incorporate these changes it can now be run. With alpha and lambda set to their default values of 1 and 0 respectively the code crashes due to an overflow error in both the *compute_cost_regularised* and *calculate_hypothesis* functions. This is because alpha is so large it causes the calculated value to be larger than can be stored.

After some testing while keeping lambda set to 0 the best value of alpha (which meant the cost was as low as possible) was 0.198 which gives a cost of 0.00721 and the following plots:

Other values of alpha tested and their results can be found in Table 1 in Appendix 1.

As can be seen in Figure 23 this gives a good approximation of the function as it follows the general shape of the data.

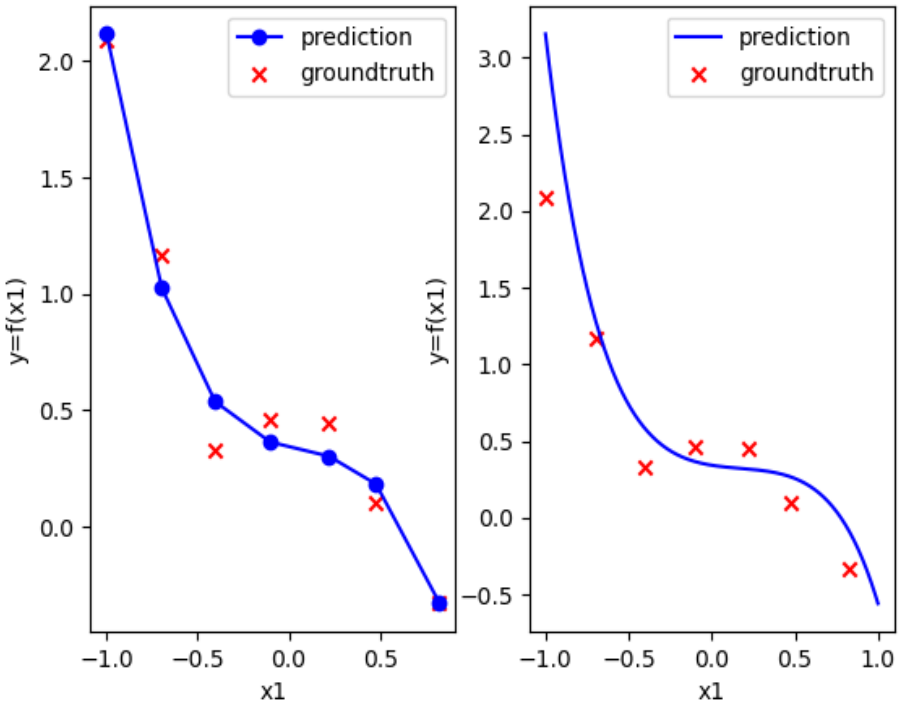

Figure 23 - Shows plot of hypothesis with alpha = 0.198, lambda = 0 and 200 iterations

The next stage is to experiment with different values of lambda while keeping the value of alpha the same at 0.198. When lambda is set to a high value such as 1 the cost is 0.1513 and the graph is shown in Figure 24.

Because the value of lambda is so high the punishment for getting too close to the data is higher, resulting in the prediction being further away from the ground truth. This is known as underfitting and can be seen in Figure 24.

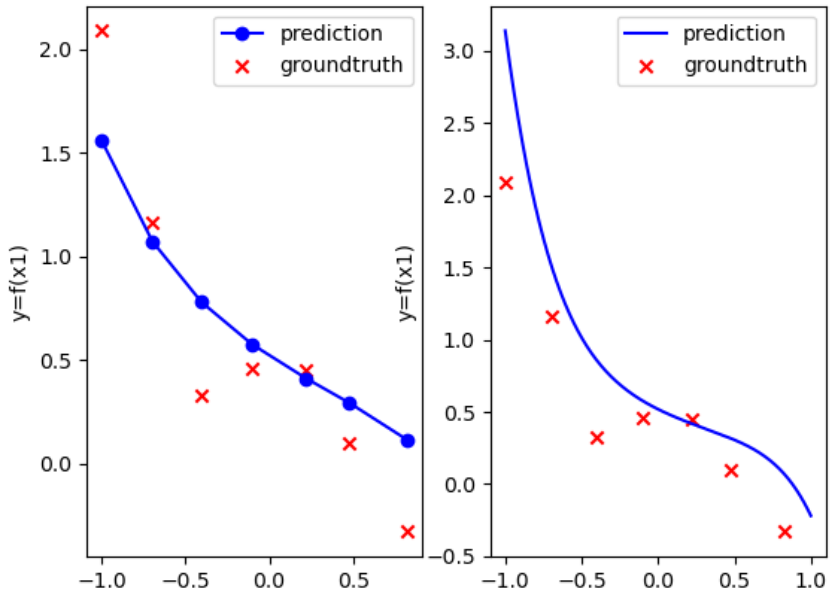

Figure 24 - Shows plot of hypothesis with alpha = 0.198, lambda = 1 and 200 iterations

After some testing the best value of lambda was determined to be 0.001. This combined with the best value of alpha gave a minimum cost of 0.00738 and a graph shown in Figure 25.

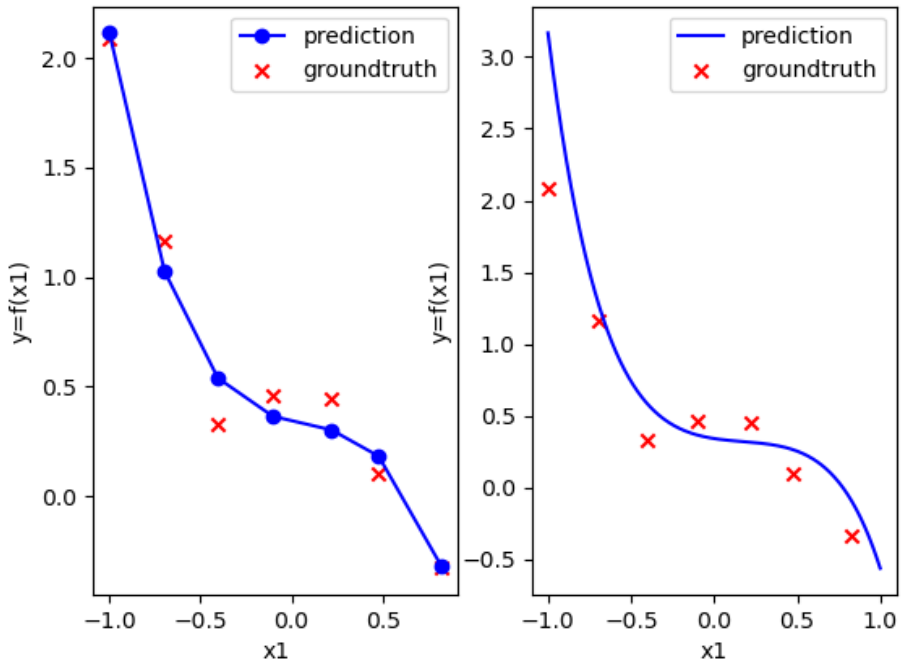

Figure 25 - Shows plot of hypothesis with alpha = 0.198, lambda = 0.001 and 200 iterations

The range of lambda values tested can be found in Table 2 in Appendix 2.

Appendices

Appendix 1 – Table 1

alpha	lamda	min cost
1	0	Does not run
0.01	0	0.01469
0.1	0	0.0082
0.11	0	0.00804
0.12	0	0.0079
0.13	0	0.00778
0.14	0	0.00767
0.15	0	0.00757
0.16	0	0.00748
0.17	0	0.00739
0.18	0	0.00732
0.19	0	0.00726
0.2	0	0.00906
0.191	0	0.00725
0.192	0	0.00724
0.193	0	0.00724
0.194	0	0.00723
0.195	0	0.00723
0.196	0	0.00722
0.197	0	0.00722
0.198	0	0.00721
0.199	0	0.00732

Appendix 2 – Table 2

alpha	lamda	min cost
0.198	1	0.1513
0.198	0.5	0.0916
0.198	0.1	0.0257
0.198	0.01	0.00896
0.198	0.02	0.01081
0.198	0.001	0.00738
0.198	0.002	0.00755
0.198	0.003	0.00772
0.198	0.004	0.00789
0.198	0.005	0.00807
0.198	0.006	0.00824
0.198	0.007	0.00842
0.198	0.008	0.0086
0.198	0.009	0.00878