# ASSIGNMENT 1, PART 1: LINEAR REGRESSION.

## Machine Learning.

Carolina Raymond Rodrigues.

Student ID: 210607000

ec21209@qmul.ac.uk

## Index

## Figures Index

## Tables Index

## 1. Linear Regression with One Variable.

**Task 1** Modify the function *calculate_hypothesis.py* to return the predicted value for a single specified training example. Include in the report the corresponding lines from your code.

The first thing to do is to fit a line for the provided data. Since one variable X is given to predict one output Y, Linear Regression (1) will be used as the hypothesis function, enabling to fit a straight line to the training data.

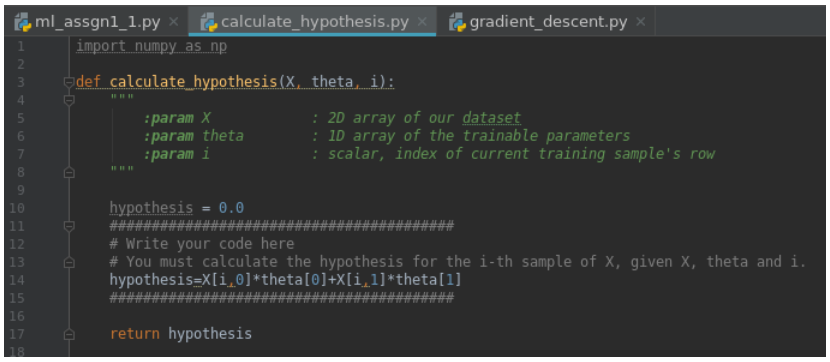
$$y = f(x) = \theta_0 + \theta_1 x = \theta^T[1, x] \tag{1}$$

where:

- $x \equiv input\ feature;$
- $\theta \equiv weights.$

As it can be seen in function (1), a bias value equal to 1 must be included. However, the input X of the *calculate_hypothesis* function already includes this value. Consequently, for the ith sample of X, we will just need to add a single line that will estimate the hypothesis according to (1).

In Fig. 1, line 14, the hypothesis is calculated as it has been stated in the previous lines.

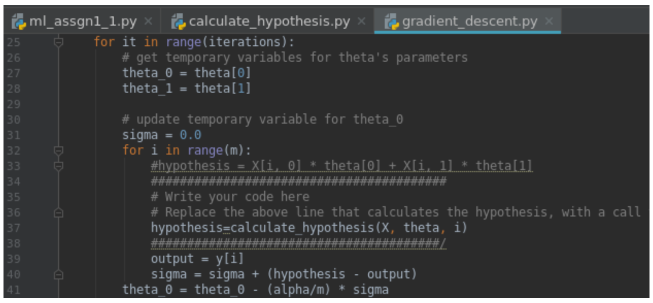


```
1   import numpy as np
2
3   def calculate_hypothesis(X, theta, i):
4       """
5           :param X            : 2D array of our dataset
6           :param theta        : 1D array of the trainable parameters
7           :param i            : scalar, index of current training sample's row
8       """
9
10      hypothesis = 0.0
11      ########################################
12      # Write your code here
13      # You must calculate the hypothesis for the i-th sample of X, given X, theta and i.
14      hypothesis=X[i,0]*theta[0]+X[i,1]*theta[1]
15      ########################################
16
17      return hypothesis
18
```

*Fig. 1 calculate_hypothesis for linear regression with one variable.*

When this is completed, run *ml_assgn1_1.py* again and you should see the gradient of the hypothesis better fit the model and the cost going down over time. Notice that the hypothesis function is not being used in the *gradient_descent* function. Modify it to use the *calculate_hypothesis* function. Include the corresponding lines of the code in your report.

The resulting lines of code can be observable in Fig. 2, in which *calculate_hypothesis* will be called in line 37, and the previous hypothesis calculation, line 13, will be removed since it will not be needed from now on.



```
25      for it in range(iterations):
26          # get temporary variables for theta's parameters
27          theta_0 = theta[0]
28          theta_1 = theta[1]
29
30          # update temporary variable for theta_0
31          sigma = 0.0
32          for i in range(m):
33              #hypothesis = X[i, 0] * theta[0] + X[i, 1] * theta[1]
34              ########################################
35              # Write your code here
36              # Replace the above line that calculates the hypothesis, with a call
37              hypothesis=calculate_hypothesis(X, theta, i)
38              ########################################/
39              output = y[i]
40              sigma = sigma + (hypothesis - output)
41          theta_0 = theta_0 - (alpha/m) * sigma
```

*Fig. 2 gradient_descent function with calculate_hypothesis function added.*

When the *ml_assgn1_1.py* is run, changes in the cost and prediction graph are reflected concerning the old plots with no modifications (Fig. 3a and 3b). However, the new prediction function (Fig. 3c) still does not fit the data correctly.

The reason behind this can be noticeable in the new cost plot (Fig. 3d). Multiple lines with different slopes can be used to fit our data. Nevertheless, the purpose is to find the line that provides the most optimal predictions. To quantify how well the linear regression function (1) fits our training data, the Sum of Squared Errors (SSE) is used.

Gradient descent will help to find the values that minimize the Sum of Square errors, and consequently the optimal weights $(\theta)$ will be obtained. Therefore, as time increases, it is expected that the cost function converges to the global minima. Nevertheless, this does not occur in the new model (Fig. 3d), the cost function is 0 until iteration 48, where the value rises dramatically. Thus, changing the value of the learning rate might be a good start for appropriately fitting the line to the data.

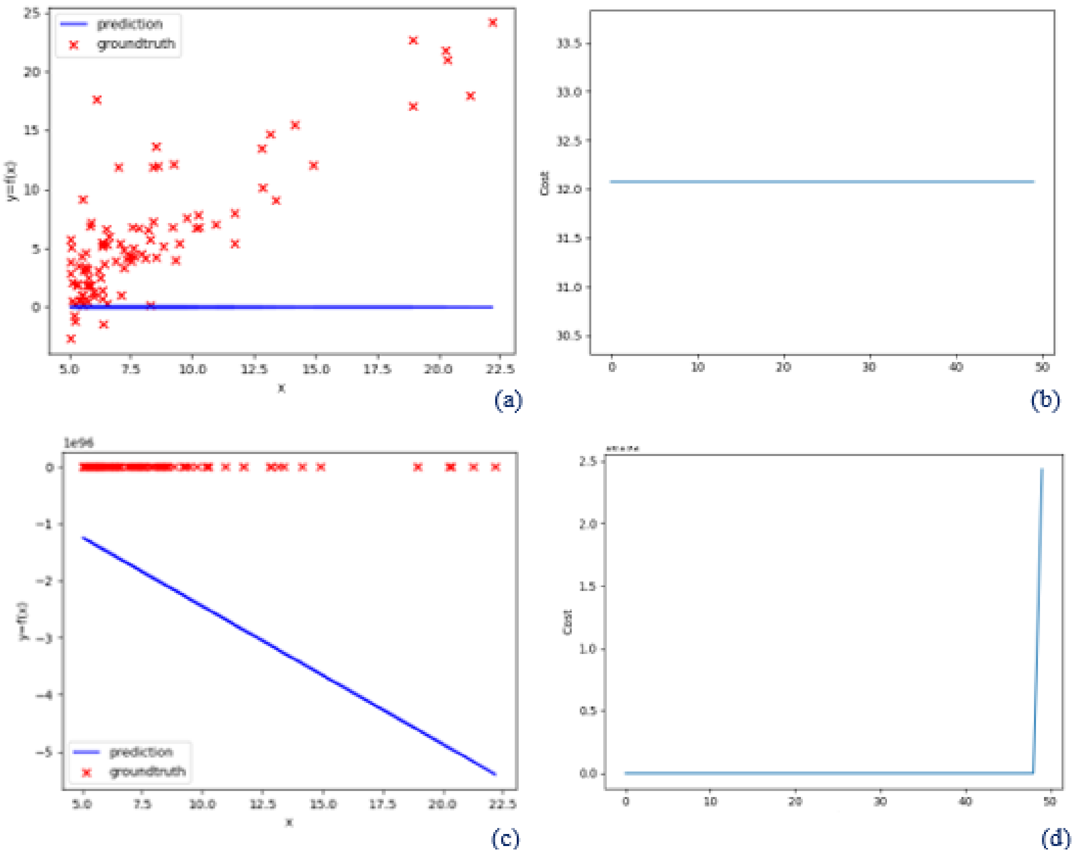


***Fig. 3**: Prediction and cost graph before (a,b) and after adding (c, d) the hypothesis function in gradient_descent.py*

Now modify the values for the learning rate, alpha in *ml_assgn1_1.py*. Observe what happens when you use a very high or very low learning rate. Document and comment on your findings in the report.

Different values for the learning rate have been used. Table 1 summarises the obtained information.

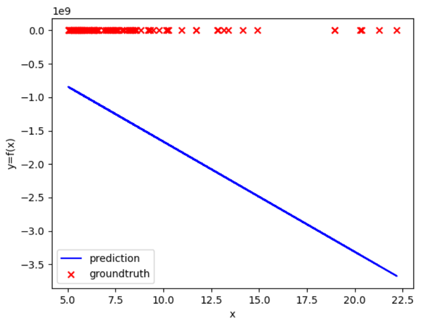
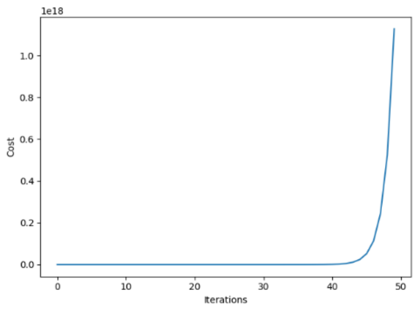
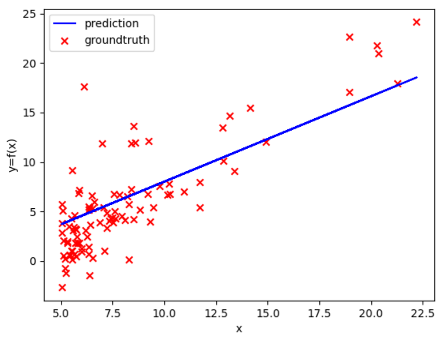
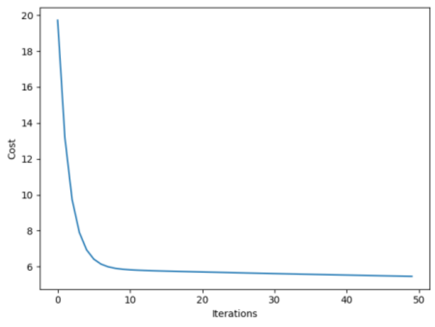
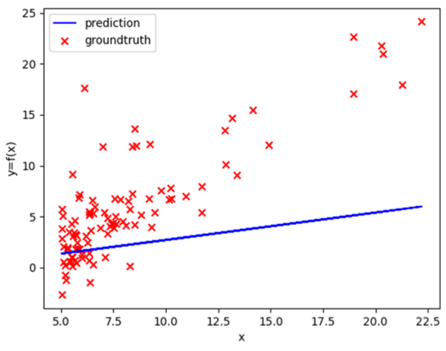
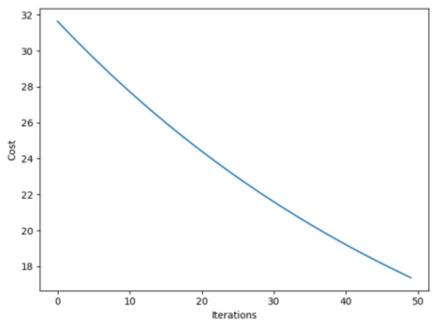
| **$\alpha$** | **Prediction** | **Cost** | **Minimum cost and iteration** |
|---|---|---|---|
| 0.03 |  |  | C:62.17<br>It: #1 |
| 0.0021 |  |  | C: 5.46<br>It #50 |
| 0.0001 |  |  | C: 17.37<br>It #50 |

***Table 1***. *Prediction graph, cost graph and minimum cost and iteration obtained for different learning rates in task 1.*

When $\alpha = 0.03$ we can see no change regarding $\alpha = 1$ (Fig. 3c, and 3d), the model is still a bad fit for our training data. This might be due to an elevated learning rate. When this value is too high, gradient descent will not converge as it is desired, instead, it will overshoot and diverge, breaching its purpose.

Moreover, if the value is too low, $\alpha = 0.0001$, the fitted line will underfit our data, and consequently, a poor generalisation of the data is made.

Nevertheless, when $\alpha = 0.0021$ a good fit is made. As it can be seen in Table 1, the predictions are close to the ground truth values. Besides, the cost functions slowly decrease to 5.5 as the number of iterations increases. This assures that the gradient descent function converges and therefore the optimal values for the weights ($\theta$) will be obtained.

## 2. Linear Regression with Multiple Variables.

**Task 2.** Modify the functions *calculate_hypothesis* and *gradient_descent* to support the new hypothesis function. Your new hypothesis function's code should be sufficiently general so that we can have any number of extra variables. Include the relevant lines of the code in your report.

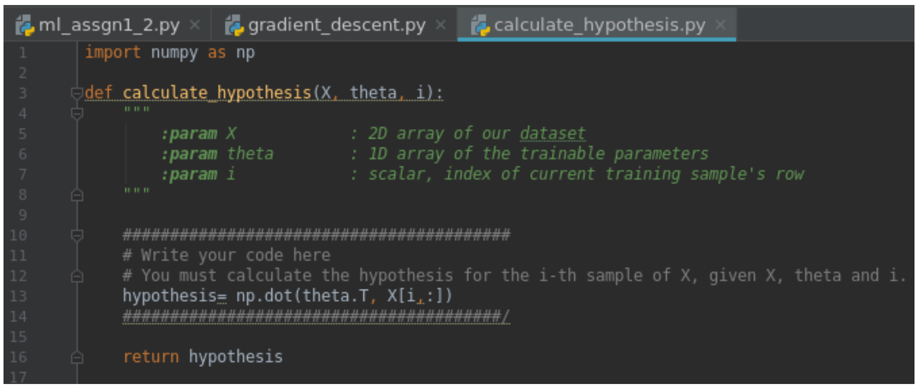
Just as before, it is necessary to compute the hypothesis function in *calculate_hypothesis.py*. Since multiple input features X are given to predict an input value Y, a new linear regression function (2) will be used to fit the training data correctly.

$$y = f(x) = \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n \qquad (2)$$

where:

- $x \equiv input\ feature;$
- $\theta \equiv weights;$
- $n \equiv number\ of\ input\ features.$

Function (2) has been included in line 13 of the following figure:



```
1   import numpy as np
2
3   def calculate_hypothesis(X, theta, i):
4       """
5           :param X            : 2D array of our dataset
6           :param theta        : 1D array of the trainable parameters
7           :param i            : scalar, index of current training sample's row
8       """
9
10      #########################################
11      # Write your code here
12      # You must calculate the hypothesis for the i-th sample of X, given X, theta and i.
13      hypothesis= np.dot(theta.T, X[i,:])
14      ########################################/
15
16      return hypothesis
17
```

*__Fig. 4__ calculate_hypothesis for linear regression with multiple variables.*

Additionally, just like in task 1, this function is included *gradient_descent.py* (Fig.5, line 36). However, this is not the only change that must be made. The calculation of the $\sigma$ and the update of the weights must be done too.

The Sum of Squared Errors (SSE) is defined as

$$E(\theta) = \sum_{i=1}^{N} \left(y_i - f(x_i; \theta)\right)^2 \qquad (3)$$

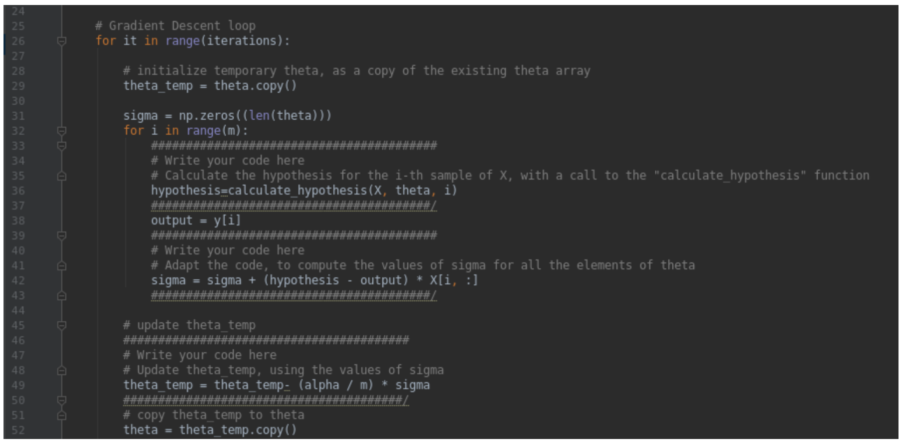
To find the optimal value of the weights, the SSE must be minimised. Since the weights are linear, if SSE is plotted against the weights, a convex cost function with a single minimum will be obtained. With help of the gradient descent, global minima can be found to then update the values of the weights, using function (4).

$$\theta^{s+1} := \theta^{s} - \alpha \frac{dE(\theta)}{d\theta} \qquad (4)$$

where:

$$\frac{dE(\theta)}{d\theta} = \sigma = \sum_i x_i(y_i - \theta^T x_i) \tag{5}$$

Since the purpose is to update the values of the weights, these two equations (4) (5) have been added in lines 42 and 49 of Fig. 5.



```
24
25      # Gradient Descent loop
26      for it in range(iterations):
27
28          # initialize temporary theta, as a copy of the existing theta array
29          theta_temp = theta.copy()
30
31          sigma = np.zeros((len(theta)))
32          for i in range(m):
33              ########################################
34              # Write your code here
35              # Calculate the hypothesis for the i-th sample of X, with a call to the "calculate_hypothesis" function
36              hypothesis=calculate_hypothesis(X, theta, i)
37              ########################################/
38              output = y[i]
39              ########################################
40              # Write your code here
41              # Adapt the code, to compute the values of sigma for all the elements of theta
42              sigma = sigma + (hypothesis - output) * X[i, :]
43              ########################################/
44
45          # update theta_temp
46          ########################################
47          # Write your code here
48          # Update theta_temp, using the values of sigma
49          theta_temp = theta_temp- (alpha / m) * sigma
50          ########################################/
51          # copy theta_temp to theta
52          theta = theta_temp.copy()
```

***Fig. 5*** *gradient_descent.py for linear regression with multiple variables.*

Once the corresponding bits of code have been added, the predictions and cost plot changes from a poor fit (Fig. 6a) and high cost (Fig. 6b) to having a linear regression function that generalizes well all the data and is a good fit for our test data (Fig. 6c). Besides the value of the cost decreases as the number of iterations increases (Fig. 6d), meaning that gradient descent converges as expected.

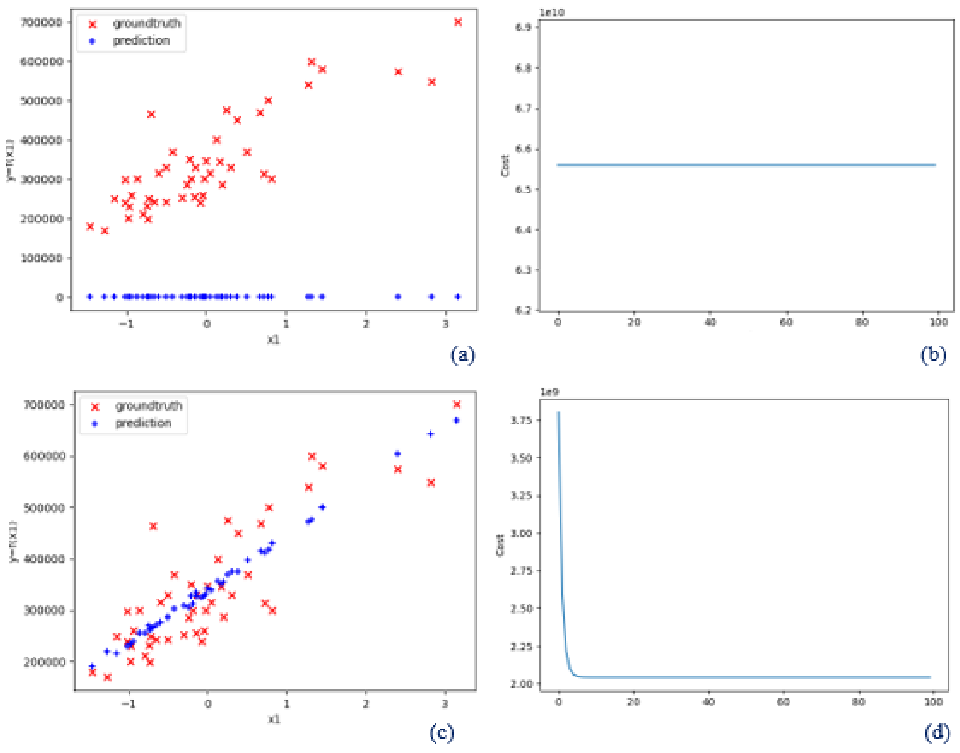


***Fig. 6*** *Predictions and cost plot before (a,b) and after (c,d) including hypothesis and gradient descent calculations.*

Run ml_assgn1_2.py and see how different values of alpha affect the convergence of the algorithm. Print the theta values found at the end of the optimization. Does anything surprise you? Include the values of theta and your observations in your report.

Different values of $\alpha$ have been used in this section. Table 2 summarises the information obtained. Additionally, Table 3 includes the value of $\theta$ for each $\alpha$.

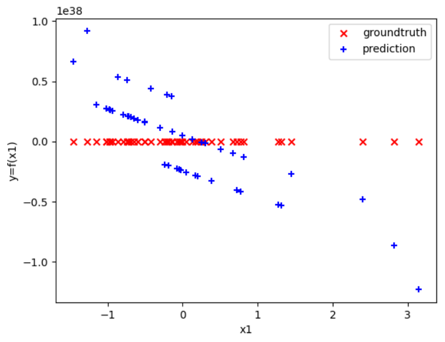
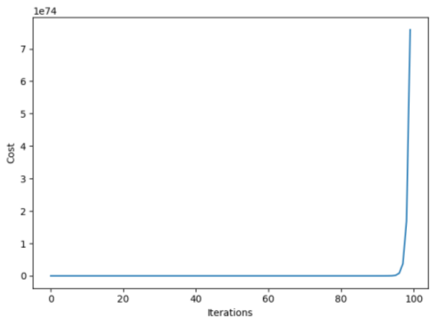
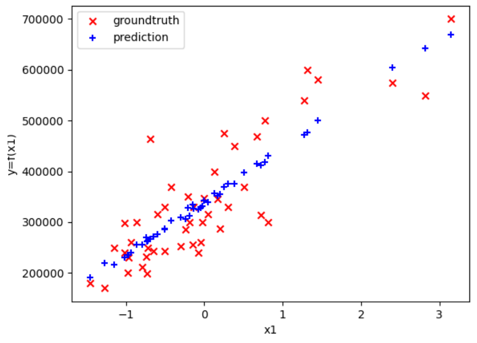
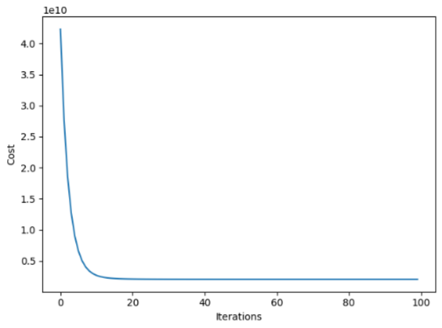
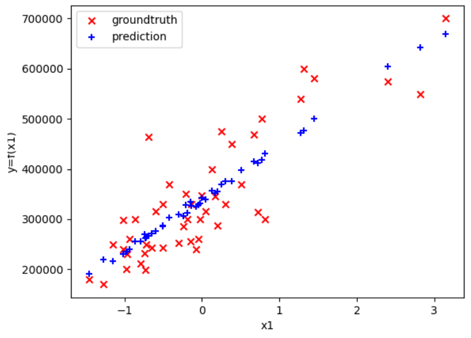
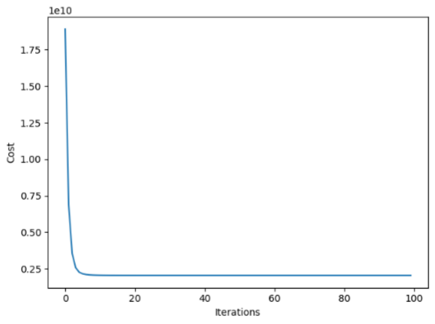
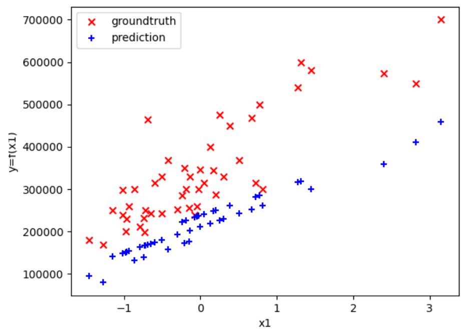
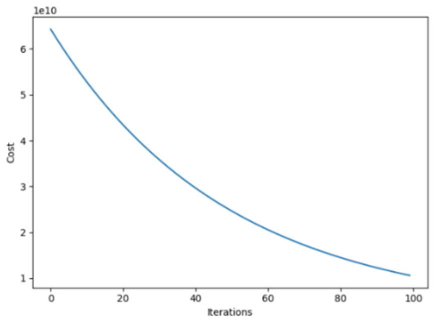
| $\alpha$ | **Prediction** | **Cost** | **Minimum cost and iteration** |
|---|---|---|---|
| 2 |  |  | C:$7.86 \cdot 10^{10}$<br>It: #1 |
| 0.2 |  |  | C:$2.04 \cdot 10^{9}$<br>It #100 |
| 0.48 |  |  | C: $2.04 \cdot 10^{9}$<br>It #77 |
| 0.01 |  |  | C: $1.1 \cdot 10^{10}$<br>It #100 |

*Table 2. Predictions plots, cost graphs and minimum cost and iteration for different values of alpha in linear regression for multiple variables.*

| $\boldsymbol{\alpha}$ | $\boldsymbol{\theta_0}$ | $\boldsymbol{\theta_1}$ | $\boldsymbol{\theta_2}$ |
|---|---|---|---|
| 2 | $-1.022 \cdot 10^{22}$ | $-2.21 \cdot 10^{37}$ | $-2.21 \cdot 10^{37}$ |
| 0.2 | $340.41 \cdot 10^{3}$ | $109.45 \cdot 10^{3}$ | $-6578.3$ |
| 0.48 | $340.41 \cdot 10^{3}$ | $109.45 \cdot 10^{3}$ | $-6578.3$ |
| 0.01 | $215.81 \cdot 10^{3}$ | $61.45 \cdot 10^{3}$ | $20.07 \cdot 10^{3}$ |

***Table 3** Values of theta for different alphas.*

As it can be seen in Table 2, with $\alpha = 2$, the cost function does not converge, therefore this value of $\alpha$ is discarded. Moreover, for $\alpha = 0.01$ the prediction plot shows a poor generalisation of the data. Predictions of the tested data are far from its real value; thus, the data is underfitted.

However, with $\alpha = 0.2\ and\ 0.48$ the obtained prediction plots are quite similar to the ground truth values. Nevertheless, even though the theta values in both cases are the same, when $\alpha = 0.2$ the minimum cost is found at iteration number 100, while in the case of $\alpha = 0.48$ the minimum cost is at iteration 77. The reason behind this difference is that the learning rate defines how fast the gradient descent is going to converge to the global minima. Therefore, in our samples, when $\alpha$ is high the minimum cost is found before. However, if we keep increasing the value we are at risk of overshooting, just as it can be seen when $\alpha = 2$.

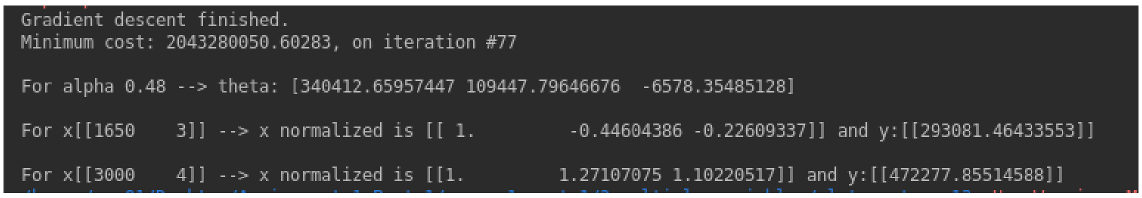
In this case, we must find a balance between fast learning vs stability. Consequently, we will choose 0.48 as the learning rate since it converges before 0.2 and keeps making a good prediction of our tested data.

Finally, we would like to use our trained theta values to make a prediction. Add some lines of code in ml_assgn1_2.py to make predictions of house prices. How much does your algorithm predict that a house with 1650 sq. ft. and 3 bedrooms cost? How about 3000 sq. ft. and 4 bedrooms?
To make a prediction you will need to normalize the two variables using the saved values for mean and standard deviation.
Add the lines of the code that you wrote in your report. Include the predictions as well that you make for the prices of the houses above.

To predict the cost of the houses, function (2) will be used to calculate the predicted value. For that, the final values of $\theta$ obtained in the previous section are utilised. However, it is important to normalize first the data and append the bias term before estimating the hypothesis.

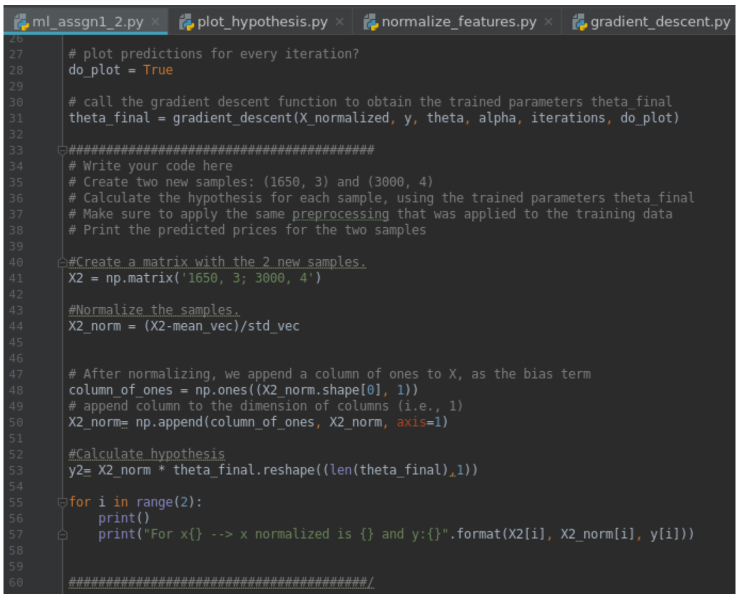
In Fig. 8, lines 40-60 includes the necessary code to predict the value of a house with 1650 sq. ft. and 3 bedrooms and another house of 3000 sq. ft. and 4 bedrooms.

The results obtained are reflected in the following figure. Where we can see that for 1650 sq., ft and 3 bedrooms, the cost of the house is 293.081,5 while the value for a 3000 sq., ft and 4 bedrooms is 472.277,86



```
Gradient descent finished.
Minimum cost: 2043280050.60283, on iteration #77

For alpha 0.48 --> theta: [340412.65957447 109447.79646676  -6578.35485128]

For x[[1650    3]] --> x normalized is [[ 1.         -0.44604386 -0.22609337]] and y:[[293081.46433553]]

For x[[3000    4]] --> x normalized is [[1.         1.27107075 1.10220517]] and y:[[472277.85514588]]
```

***Fig. 7** Predicted value of new data.*



ml_assgn1_2.py plot_hypothesis.py normalize_features.py gradient_descent.py

```python
# plot predictions for every iteration?
do_plot = True

# call the gradient descent function to obtain the trained parameters theta_final
theta_final = gradient_descent(X_normalized, y, theta, alpha, iterations, do_plot)

########################################
# Write your code here
# Create two new samples: (1650, 3) and (3000, 4)
# Calculate the hypothesis for each sample, using the trained parameters theta_final
# Make sure to apply the same preprocessing that was applied to the training data
# Print the predicted prices for the two samples

#Create a matrix with the 2 new samples.
X2 = np.matrix('1650, 3; 3000, 4')

#Normalize the samples.
X2_norm = (X2-mean_vec)/std_vec


# After normalizing, we append a column of ones to X, as the bias term
column_of_ones = np.ones((X2_norm.shape[0], 1))
# append column to the dimension of columns (i.e., 1)
X2_norm= np.append(column_of_ones, X2_norm, axis=1)

#Calculate hypothesis
y2= X2_norm * theta_final.reshape((len(theta_final),1))

for i in range(2):
    print()
    print("For x{} --> x normalized is {} and y:{}".format(X2[i], X2_norm[i], y[i]))


########################################/
```

*Fig. 8 Prediction of new data code.*

## 3. Regularized Linear Regression.

**Task 3.** Modify *gradient_descent* to use the *compute_cost_regularised* method instead of *compute_cost*. Include the relevant lines of the code in your report and a brief explanation. Next, modify *gradient_descent* to incorporate the new cost function. Include the relevant lines of the code in your report. After *gradient_descent* has been updated, run ml_assgn1_3.py. This will plot the hypothesis function found at the end of the optimization.

Regularization is used to add a penalty to our cost to discourage heavy use of the weights and therefore avoid overfitting. For this section, Ridge regression will be used for applying regularisation terms and will affect the sigma as it can be seen in the following formula

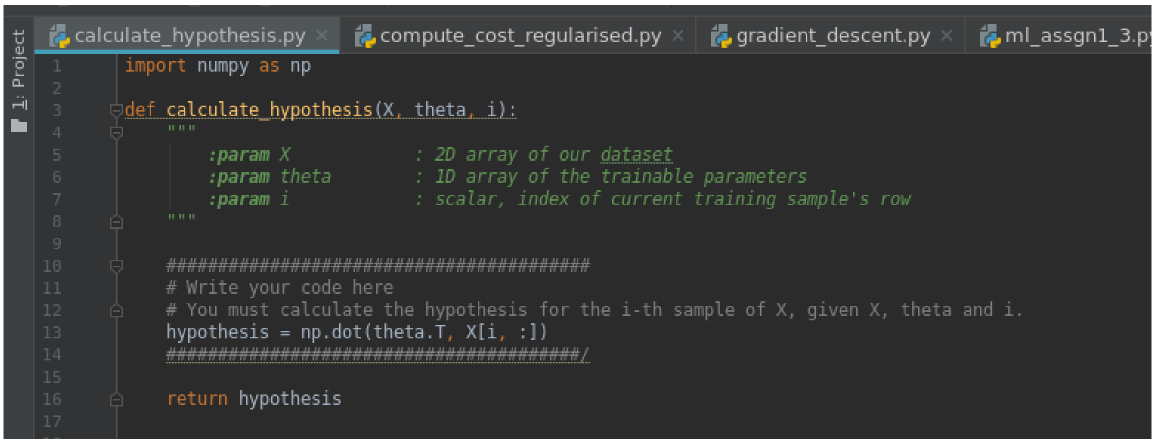
$$\frac{dE(\theta)}{d\theta} = \sigma = \sum_i \phi(x_i)^T\left(y_i - \theta^T\phi(x_i)\right) + \lambda\theta^T\theta \tag{6}$$

Where $\lambda$ will control the degree of over/underfitting.
However, we don't want to punish the bias term, therefore the update of the weights will follow the resulting formulas:

$$\theta_0 := \theta_0 - \alpha \cdot \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^i\right) - y^i\right)x_o^i \tag{7}$$

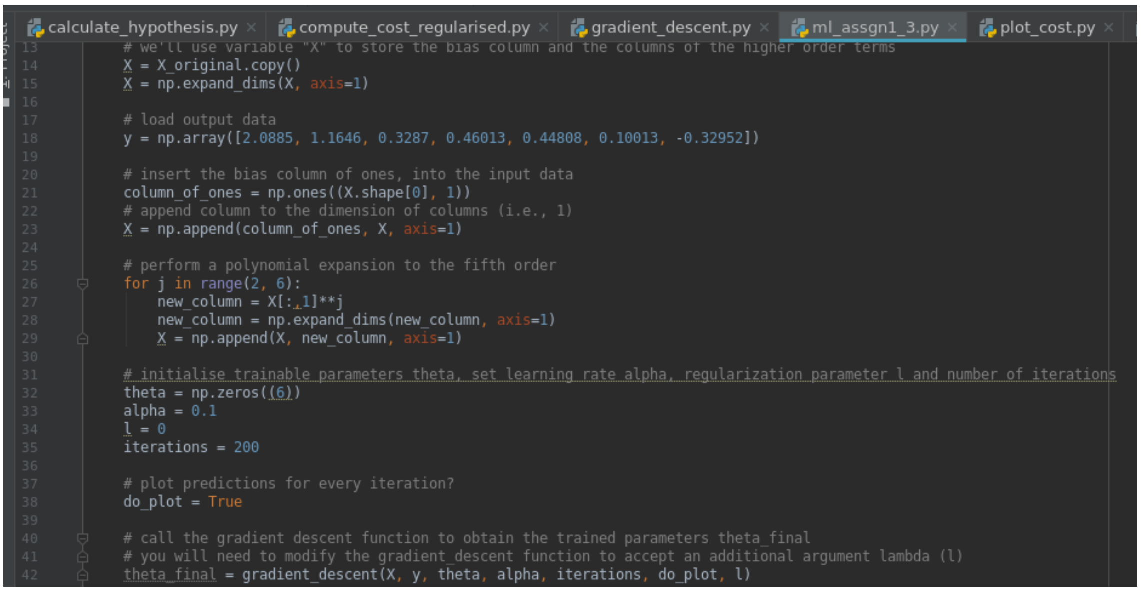
$$\theta_j := \theta_j\left(1 - \alpha \cdot \frac{\lambda}{m}\right) - \alpha \cdot \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)} \tag{7}$$

All these changes have been reflected in line 13 of *calculate_hypothesis* (Fig. 9), line 42 of *ml_assgn1_3* (Fig. 10), and lines 9 and 38 to 56 in *gradient_descent* (Fig. 11).



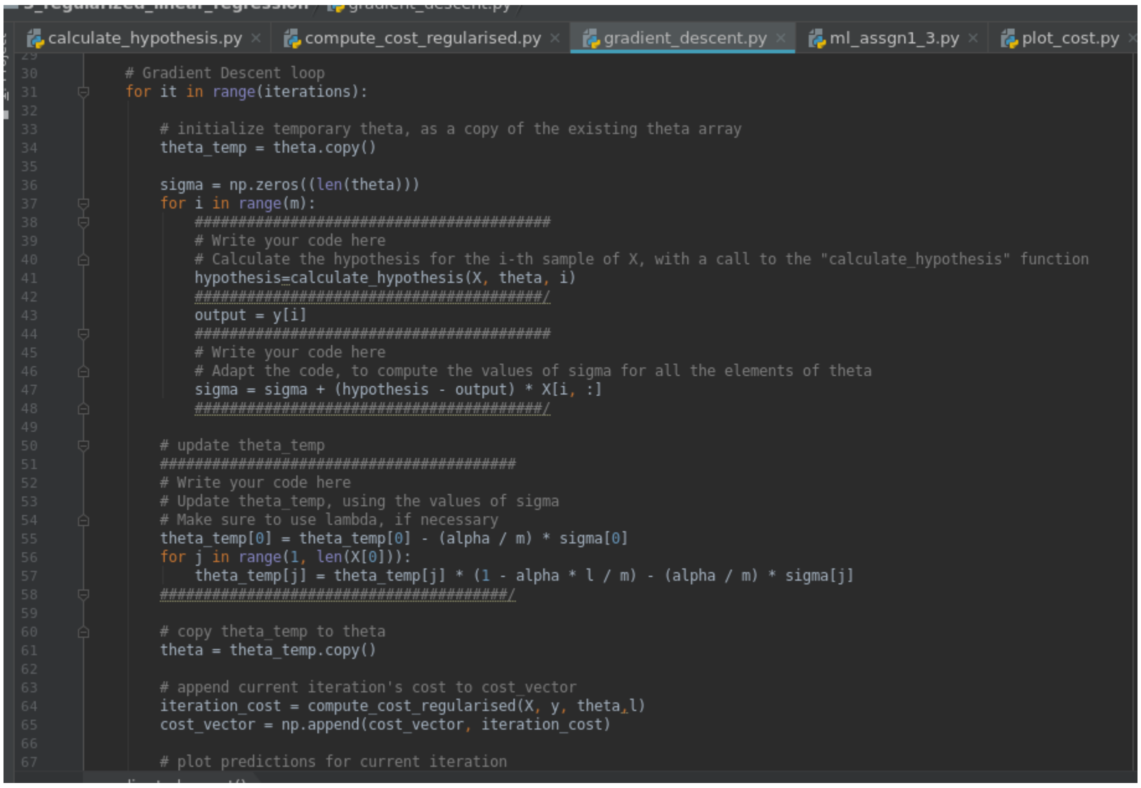
```
1   import numpy as np
2
3   def calculate_hypothesis(X, theta, i):
4       """
5           :param X            : 2D array of our dataset
6           :param theta        : 1D array of the trainable parameters
7           :param i            : scalar, index of current training sample's row
8       """
9
10      ##########################################
11      # Write your code here
12      # You must calculate the hypothesis for the i-th sample of X, given X, theta and i.
13      hypothesis = np.dot(theta.T, X[i, :])
14      ##########################################/
15
16      return hypothesis
17
```

***Fig.* 9** *Calculate hypothesis for regularization.*



```
    # we'll use variable "X" to store the bias column and the columns of the higher order terms
    X = X_original.copy()
    X = np.expand_dims(X, axis=1)

    # load output data
    y = np.array([2.0885, 1.1646, 0.3287, 0.46013, 0.44808, 0.10013, -0.32952])

    # insert the bias column of ones, into the input data
    column_of_ones = np.ones((X.shape[0], 1))
    # append column to the dimension of columns (i.e., 1)
    X = np.append(column_of_ones, X, axis=1)

    # perform a polynomial expansion to the fifth order
    for j in range(2, 6):
        new_column = X[:,1]**j
        new_column = np.expand_dims(new_column, axis=1)
        X = np.append(X, new_column, axis=1)

    # initialise trainable parameters theta, set learning rate alpha, regularization parameter l and number of iterations
    theta = np.zeros((6))
    alpha = 0.1
    l = 0
    iterations = 200

    # plot predictions for every iteration?
    do_plot = True

    # call the gradient descent function to obtain the trained parameters theta_final
    # you will need to modify the gradient_descent function to accept an additional argument lambda (l)
    theta_final = gradient_descent(X, y, theta, alpha, iterations, do_plot, l)
```

*Fig. 10 ml_assgn1_3 for regularization*



```
    # Gradient Descent loop
    for it in range(iterations):

        # initialize temporary theta, as a copy of the existing theta array
        theta_temp = theta.copy()

        sigma = np.zeros((len(theta)))
        for i in range(m):
            ########################################
            # Write your code here
            # Calculate the hypothesis for the i-th sample of X, with a call to the "calculate_hypothesis" function
            hypothesis=calculate_hypothesis(X, theta, i)
            ########################################/
            output = y[i]
            ########################################
            # Write your code here
            # Adapt the code, to compute the values of sigma for all the elements of theta
            sigma = sigma + (hypothesis - output) * X[i, :]
            ########################################/

        # update theta_temp
        ########################################
        # Write your code here
        # Update theta_temp, using the values of sigma
        # Make sure to use lambda, if necessary
        theta_temp[0] = theta_temp[0] - (alpha / m) * sigma[0]
        for j in range(1, len(X[0])):
            theta_temp[j] = theta_temp[j] * (1 - alpha * l / m) - (alpha / m) * sigma[j]
        ########################################/

        # copy theta_temp to theta
        theta = theta_temp.copy()

        # append current iteration's cost to cost_vector
        iteration_cost = compute_cost_regularised(X, y, theta,l)
        cost_vector = np.append(cost_vector, iteration_cost)

        # plot predictions for current iteration
```

*Fig. 11 gradient descent calculation with regularization.*

First of all, find the best value of alpha to use in order to optimize best. Report the value of alpha that you found in your report.

Three different values of $\alpha$ have been used for this section: 0.1, 0.8 and 0.01. the results obtained are summarized in the following table:

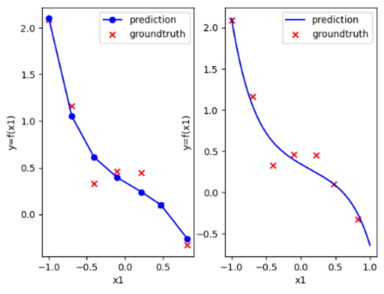
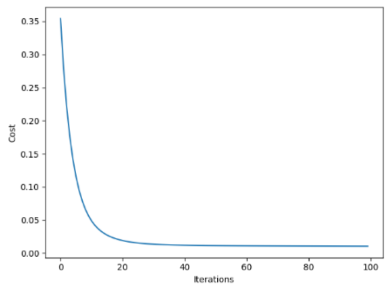
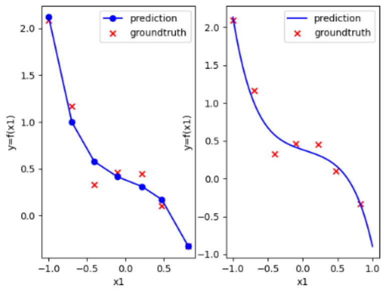
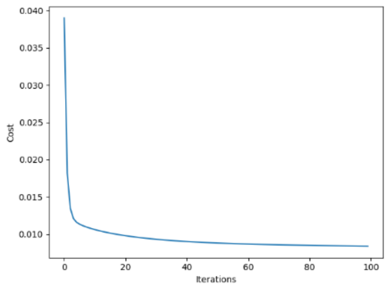
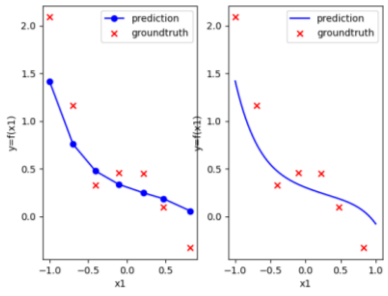
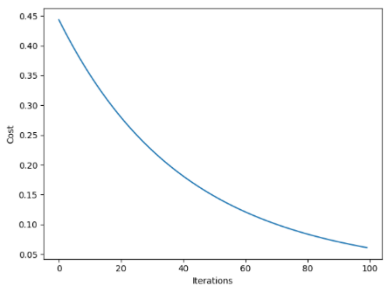
| $\alpha$ | **Prediction** | **Cost** | **Minimum cost and iteration** |
|---|---|---|---|
| 0.1 |  |  | C:0.01<br>It: #100 |
| 0.8 |  |  | C: 0.008<br>It #100 |
| 0.001 |  |  | C: 0.061<br>It #100 |

***Table 4*** *Prediction plot, cost graph and minimum cost value and iteration for different alphas after regularization*.

When $\alpha = 0.001$, the value is too small and therefore the fitted line fails to generalize well, a prove of this can be found in the predictions plot, where the prediction line is far from estimating values similar to the ground truth labels, therefore when $\alpha = 0.001$ the data is underfitted.

On the other hand, when $\alpha = 0.1$ the data does fit well the real values, but we might be at risk of overfitting the data. When $\alpha = 0.8$ the risk is lower as it can be seen, and therefore it will be the selected alpha. However, to avoid overfitting, generalization is used in the next section.

Next, experiment with different values and see how this affects the shape of the hypothesis. Note that *gradient_descent* will have to be modified to take an extra parameter, l (which represents λ, used for regularization). Include in your report the plots for a few different values and comments.

In this section, $\alpha = 0.8$ and *lambda* will take values: 0, 0.35 and 1. The resulting information is included in the following table.

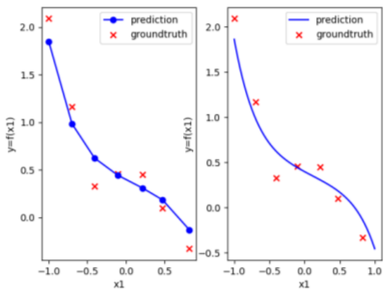
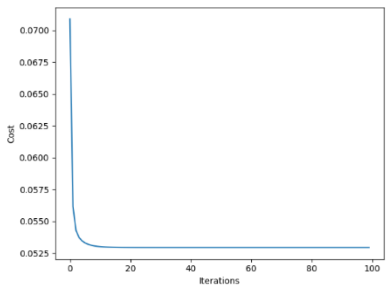
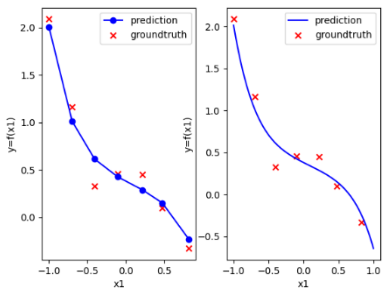
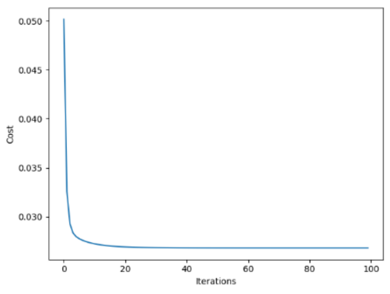
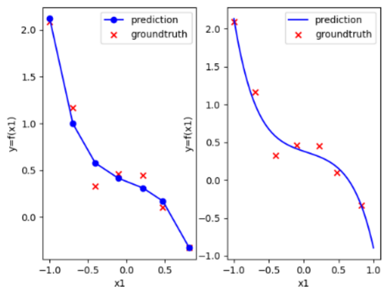
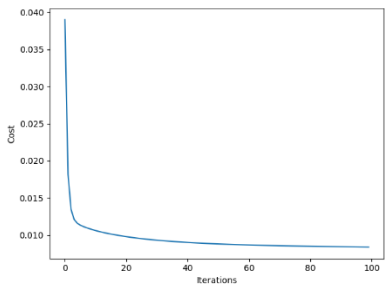
| $\lambda$ | Prediction | Cost | Minimum cost and iteration |
|---|---|---|---|
| 1 |  |  | C:0.05<br>It: #100 |
| 0.35 |  |  | C: 0.026<br>It #100 |
| 0 |  |  | C: 0.008<br>It #100 |

***Table 5*** *Prediction plot, cost graph and minimum cost value and iteration for different lambdas after regularization.*

As it can be observable in the previous table, once regularisation is added, the risk of overfitting has decreased as expected. When $\lambda = 0$ the model might have picked noise or random fluctuations in the data and learned as concepts, which negatively impacts our model. However, when $\lambda = 0.35$, we have discouraged the use of heavy weights, resulting in a better model generalization.