AVINASH BABU (210797714)
a.babu@se21.qmul.ac.uk

ECS708U/ECS708P Machine Learning

Assignment 1: Part 1 – Linear Regression

AVINASH BABU (210797714)
a.babu@se21.qmul.ac.uk

1. Linear Regression with One Variable

Task 1

calculate_hypothesis.py

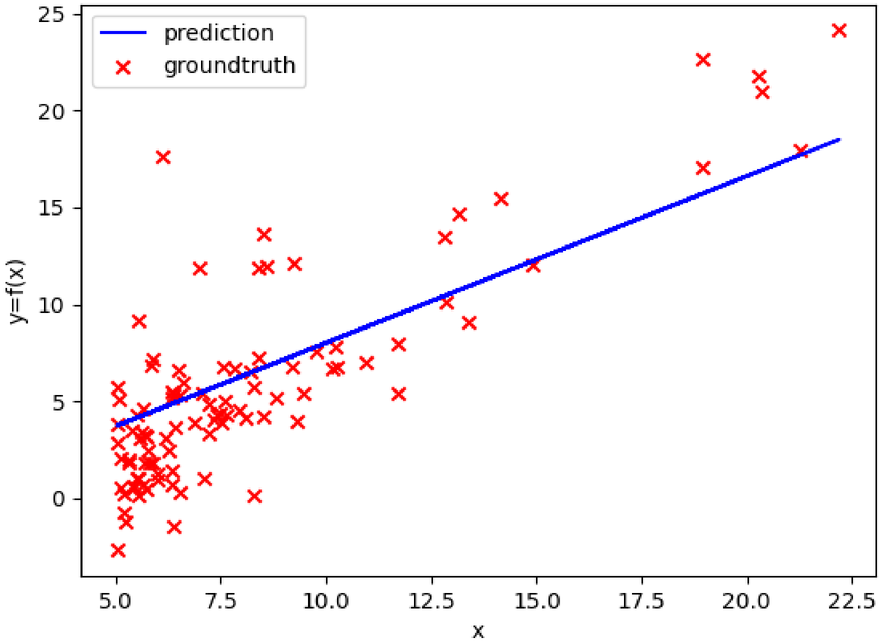
```
import numpy as np

def calculate_hypothesis(X, theta, i):
    """
        :param X            : 2D array of our dataset
        :param theta        : 1D array of the trainable parameters
        :param i            : scalar, index of current training sample's
row
    """

    hypothesis = 0.0
    #########################################
    # Write your code here
    # You must calculate the hypothesis for the i-th sample of X, given X,
theta and i.
    hypothesis = theta[0]*X[i][0] + theta[1]*X[i][1]
    #########################################
    return hypothesis
```

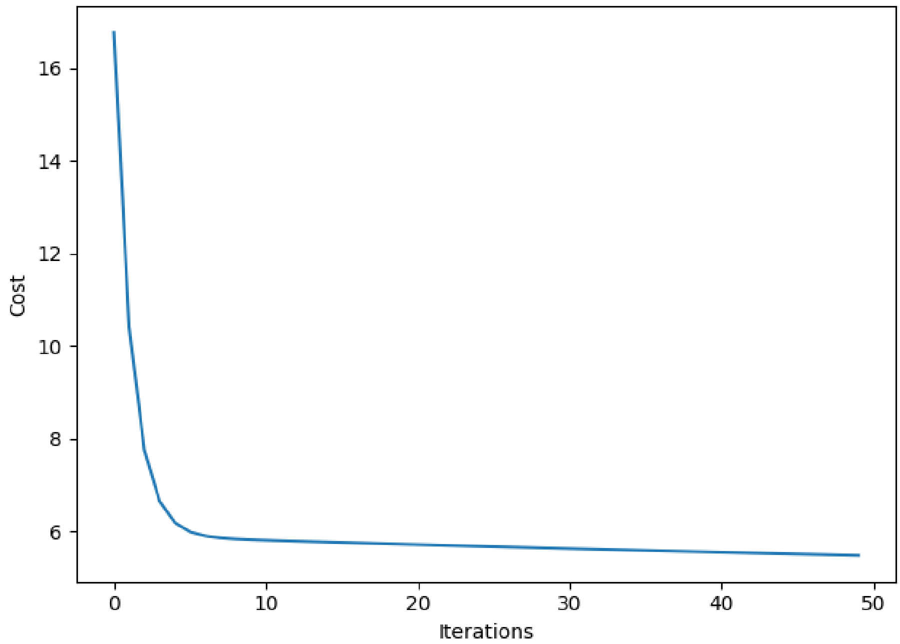
gradient_descent.py

```
def gradient_descent(X, y, theta, alpha, iterations, do_plot):
    """
        :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
    """

    # Create just a figure and only one subplot
    fig, ax1 = plt.subplots()
    if do_plot==True:
        plot_hypothesis(X, y, theta, ax1)

    m = X.shape[0] # the number of training samples is the number of rows of
array X
    cost_vector = np.array([], dtype=np.float32) # empty array to store the
cost for every iteration

    # Gradient Descent
    for it in range(iterations):
        # get temporary variables for theta's parameters
        theta_0 = theta[0]
        theta_1 = theta[1]

        # update temporary variable for theta_0
```

```
        sigma = 0.0
        for i in range(m):
            ########################################
            # Write your code here
            # Replace the above line that calculates the hypothesis, with a
call to the "calculate_hypothesis" function
            hypothesis = calculate_hypothesis(X, theta, i)
            ########################################/
            output = y[i]
            sigma = sigma + (hypothesis - output)
        theta_0 = theta_0 - (alpha/m) * sigma

        # update temporary variable for theta_1
        sigma = 0.0
        for i in range(m):
            ########################################
            # Write your code here
            # Replace the above line that calculates the hypothesis, with a
call to the "calculate_hypothesis" function
            hypothesis = calculate_hypothesis(X, theta, i)
            ########################################/
            output = y[i]
            sigma = sigma + (hypothesis - output) * X[i, 1]
        theta_1 = theta_1 - (alpha/m) * sigma

        # update theta, using the temporary variables
        theta = np.array([theta_0, theta_1])

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

        # plot predictions for current iteration
        if do_plot==True:
            plot_hypothesis(X, y, theta, ax1)
    ####################################################

    # plot predictions using the final parameters
    plot_hypothesis(X, y, theta, ax1)
    # save the predictions as a figure
    plot_filename = os.path.join(os.getcwd(), 'figures', 'predictions.png')
    plt.savefig(plot_filename)
    print('Gradient descent finished.')

    # Plot the cost for all iterations
    plot_cost(cost_vector)
    min_cost = np.min(cost_vector)
    argmin_cost = np.argmin(cost_vector)
    print('Minimum cost: {:.5f}, on iteration #{}'.format(min_cost,
argmin_cost+1))

    return theta
```

Setting the learning rate is as 0.02 and Iterations as 50 generates the following graph





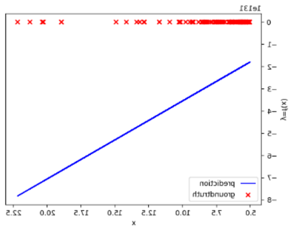
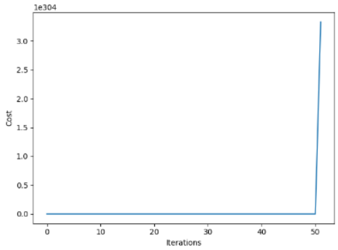
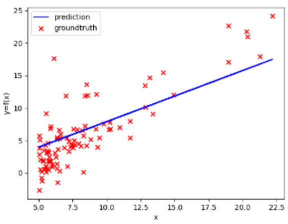
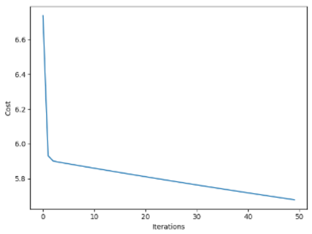
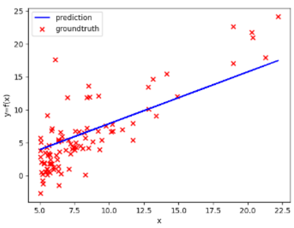
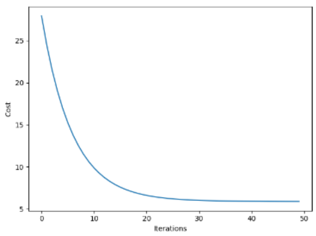
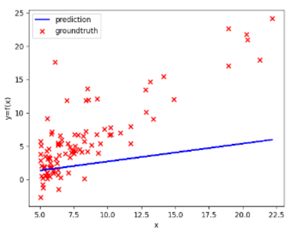
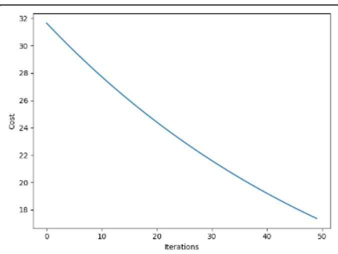
Minimum cost: 5.47965, on iteration #50

AVINASH BABU (210797714)
a.babu@se21.qmul.ac.uk

Observations:

The learning rate determines how aggressive each step the algorithm makes. The value chosen has an impact on how quickly the algorithm learns and whether or not the cost function is minimised. Gradient descent, when the learning rate is too high, inadvertently increases rather than decreases the training error. When the learning rate is too low, training not only becomes slower, but also sometimes becomes stuck with a high training error.

When we put very low learning rate such as 0.0001, higher number of iterations are needed for the graph to converge. When we put higher learning rate such as 1 the graph fails to converge, or even diverges and overshoots the parameters.

S.No	Learning Rate	Iterations	Figure 1 (predictions.png)	Figure 2 (cost.png)
1	5	50		
2	0.01	50		
3	0.001	50		
4	0.0001	50		

AVINASH BABU (210797714)
a.babu@se21.qmul.ac.uk

2. Linear Regression with Multiple Variables

Task 2

calculate_hypothesis.py

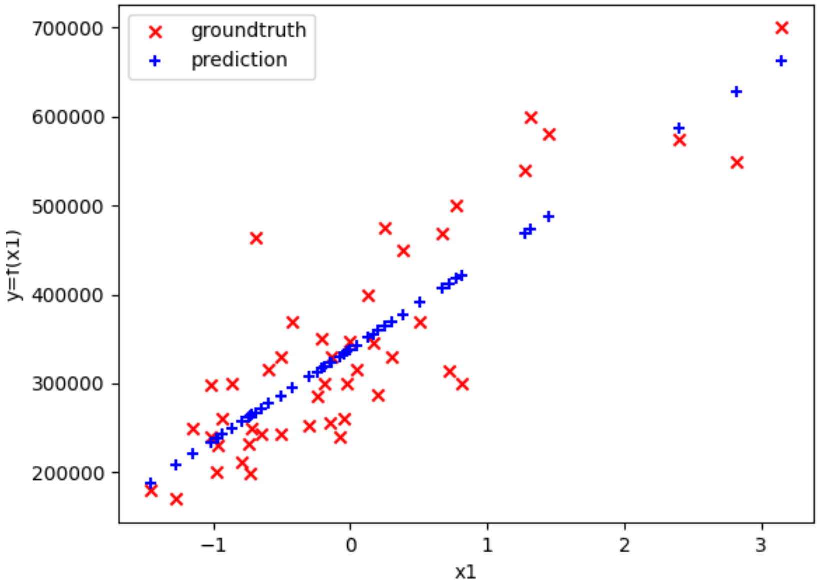
```
def calculate_hypothesis(X, theta, i):
    """
        :param X            : 2D array of our dataset
        :param theta        : 1D array of the trainable parameters
        :param i            : scalar, index of current training sample's row
    """

    ########################################
    # Write your code here
    # You must calculate the hypothesis for the i-th sample of X, given X, theta and i.
    x = X[i]
    hypothesis = np.dot(theta,x)
    ########################################
    return hypothesis
```

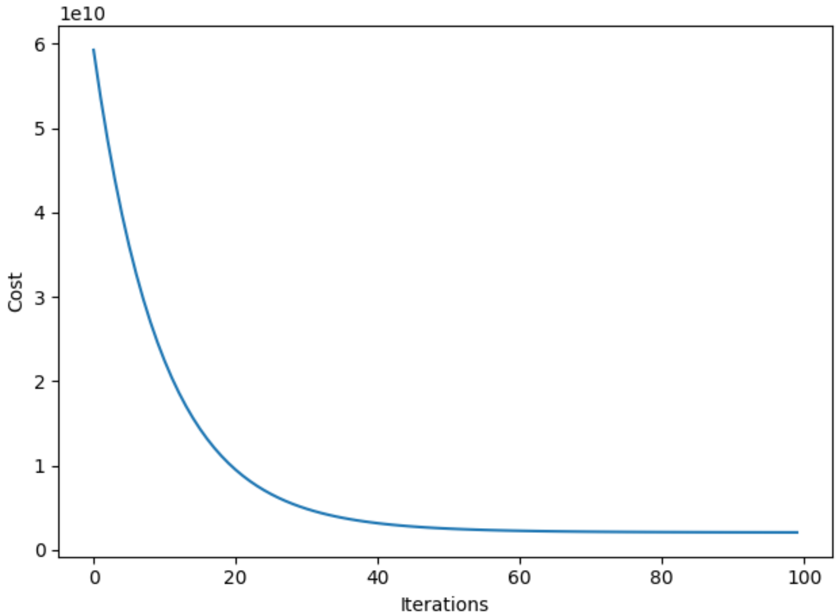
gradient_descent.py

```
def gradient_descent(X, y, theta, alpha, iterations, do_plot):
    """
        :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
    """

    # Create just a figure and only one subplot
    fig, ax1 = plt.subplots()
    if do_plot==True:
        plot_hypothesis(X, y, theta, ax1)

    m = X.shape[0] # the number of training samples is the number of rows of
array X
    cost_vector = np.array([], dtype=np.float32) # empty array to store the
cost for every iteration

    # 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_temp, i)
            ########################################
            output = y[i]
            ########################################
            # Write your code here
            # Adapt the code, to compute the values of sigma for all the
elements of theta
            for j in range(len(theta)):
                sigma[j] = sigma[j] + (hypothesis - output) * X[i, j]
            ########################################/

        # update theta_temp
        ########################################
        # Write your code here
        # Update theta_temp, using the values of sigma
        for j in range(len(theta)):
            theta_temp[j] = theta_temp[j] - (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(X, y, theta)
        cost_vector = np.append(cost_vector, iteration_cost)

        # plot predictions for current iteration
        if do_plot==True:
            plot_hypothesis(X, y, theta, ax1)
    ####################################################

    # plot predictions using the final parameters
    plot_hypothesis(X, y, theta, ax1)
    # save the predictions as a figure
    plot_filename = os.path.join(os.getcwd(), 'figures', 'predictions.png')
    plt.savefig(plot_filename)
    print('Gradient descent finished.')

    # Plot the cost for all iterations
    plot_cost(cost_vector)
    min_cost = np.min(cost_vector)
    argmin_cost = np.argmin(cost_vector)
    print('Minimum cost: {:.5f}, on iteration #{}'.format(min_cost,
argmin_cost+1))

    return theta
```

Setting the learning rate as 0.05 and the number of iterations to 100 generates the following graph

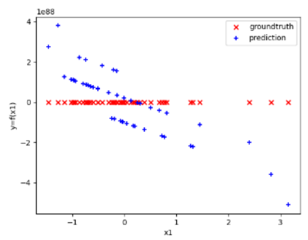
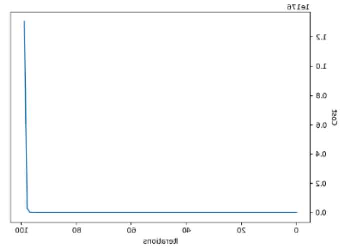
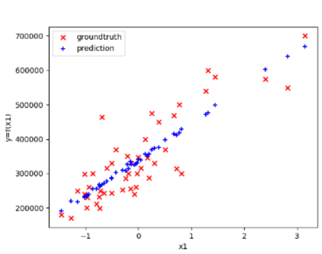
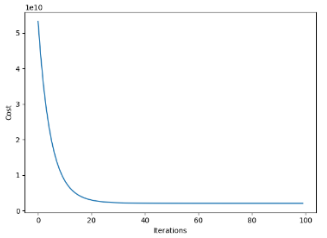
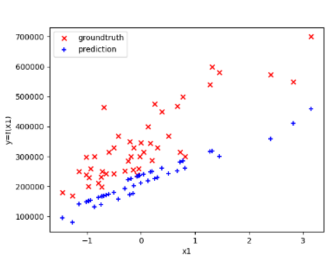
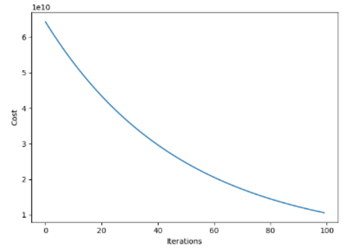
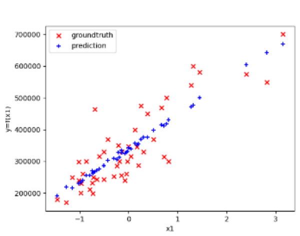
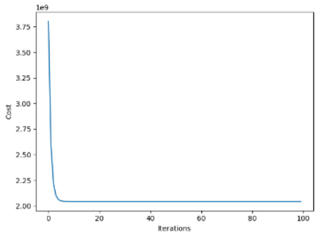
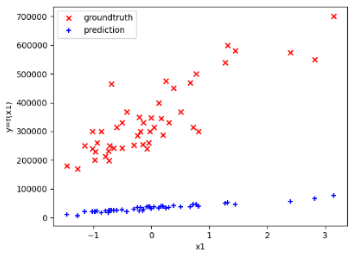
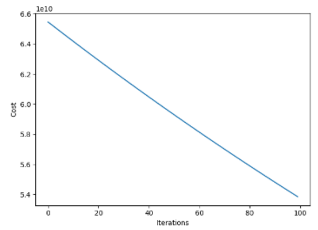




Minimum cost: 2062616003.82372, on iteration #100

Final Theta Values: [3.38397236e+05 1.03161481e+05 -3.22620198e+02]

AVINASH BABU (210797714)
a.babu@se21.qmul.ac.uk

S.No	Learning Rate	Theta	Figure 1 (predictions.png)	Figure 2 (cost.png)
1	5	[-4.48411088e+72 -9.14324521e+87 -9.14324521e+87]		
2	0.1	[340403.61773803 108803.37852266 -5933.9413402]		
3	0.01	[215810.61679138 61446.18781361 20070.13313796]		
4	1	[340412.65957447 109447.79646964 -6578.35485416]		
5	0.001	[340412.65957447 109447.79646964 -6578.35485416]		

ml_assgn1_2.py *(Code added to make predictions of houses)*

```
#########################################
# 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 train-
ing data
# Print the predicted prices for the two samples
#########################################

checkPrice = np.array([[1650, 3], [3000, 4]])
X_normalized, mean_vec, std_vec = normalize_features(checkPrice)

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

prediction_one = calculate_hypothesis(X_normalized, theta_final, 0)
print(prediction_one)
prediction_two = calculate_hypothesis(X_normalized, theta_final, 1)
print(prediction_two)
```

Prediction of House Prices

S.No	Learning Rate	Iterations	Price of House with 1650 sq. ft. and 3 bedrooms	Price of House with 3000 sq. ft. and 4 bedrooms
1	0.1	100	237534.18055557134	443273.05492049025
2	0.01	250	211969.2796795118	413669.349333568
3	0.5	100	211969.2796795118	413669.349333568
4	1	100	237543.21795898757	443282.1011899486

AVINASH BABU (210797714)
a.babu@se21.qmul.ac.uk

3. Regularized Linear Regression

Task 3

calculate_hyphothesis.py

```
def calculate_hypothesis(X, theta, i):
    """
        :param X            : 2D array of our dataset
        :param theta        : 1D array of the trainable parameters
        :param i            : scalar, index of current training sample's
row
    """
    ##########################################
    # Write your code here
    # You must calculate the hypothesis for the i-th sample of X, given X,
theta and i.
    x = X[i]
    hypothesis = theta[0]*x[0] + theta[1]*x[1] + theta[2]*pow(x[1], 2) +
theta[3]*pow(x[1], 3) + theta[4]*pow(x[1], 4) + theta[5]*pow(x[1], 5)
    ########################################/

    return hypothesis
```

gradient_descent.py

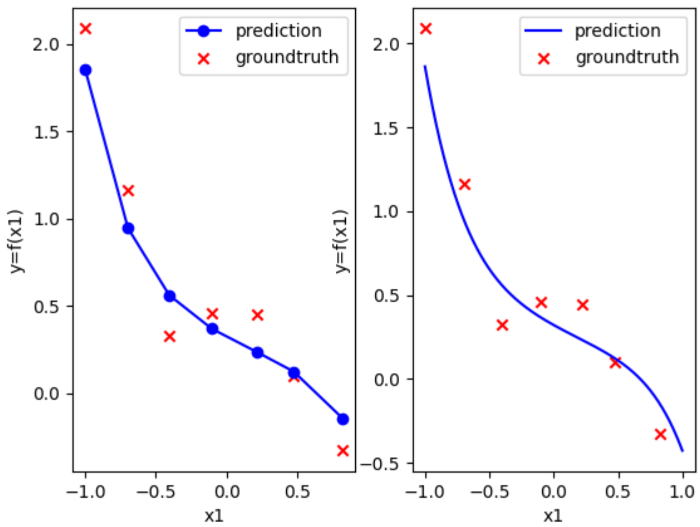
```
def gradient_descent(X, y, theta, alpha, iterations, do_plot, l):
    """
        :param X            : 2D array of our dataset
        :param y            : 1D array of the groundtruth labels of the
dataset
        :param theta        : 1D array of the trainable parameters
        :param alpha        : 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            : lamda
    """

    # Create just a figure and two subplots.
    # The first subplot (ax1) will be used to plot the predictions on the
given 7 values of the dataset
    # The second subplot (ax2) will be used to plot the predictions on a
densely sampled space, to get a more smooth curve
    fig, (ax1, ax2) = plt.subplots(1, 2)
    if do_plot==True:
        plot_hypothesis(X, y, theta, ax1)

    m = X.shape[0] # the number of training samples is the number of rows of
array X
    cost_vector = np.array([], dtype=np.float32) # empty array to store the
cost for every iteration

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

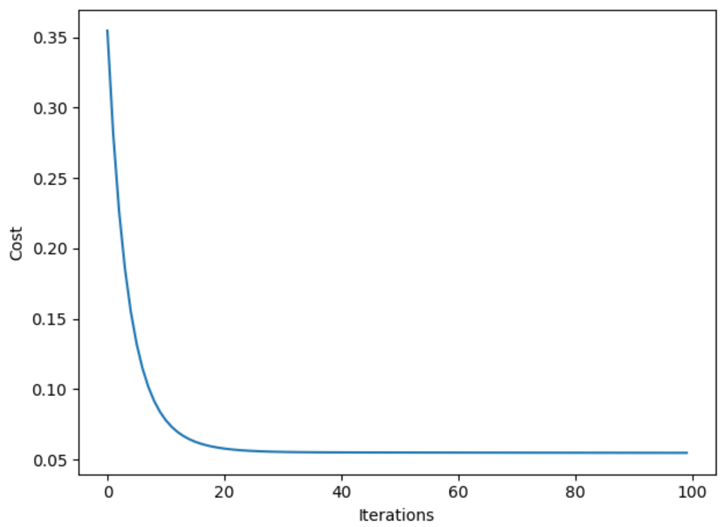
```
        # initialize temporary theta, as a copy of the existing theta array
        theta_temp = theta.copy()

        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_temp, i)
            ########################################/
            output = y[i]
            ########################################
            # Write your code here
            # Adapt the code, to compute the values of sigma for all the
elements of theta
            for j in range(len(theta)):
                sigma[j] = sigma[j] + (hypothesis - output) * X[i, j]
            ########################################/

        # update theta_temp
        ########################################
        # Write your code here
        # Update theta_temp, using the values of sigma
        # Make sure to use lambda, if necessary
        for j in range(len(theta)):
            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
        if do_plot==True:
            plot_hypothesis(X, y, theta, ax1)
    ####################################################

    # plot predictions on the dataset's points using the final parameters
    plot_hypothesis(X, y, theta, ax1)
    # sample 1000 points, from -1.0 to +1.0
    X_sampled = np.linspace(-1.0, 1.0, 1000)
    # plot predictions on the sampled points using the final parameters
    plot_sampled_points(X, y, X_sampled, theta, ax2)

    # save the predictions as a figure
    plot_filename = os.path.join(os.getcwd(), 'figures', 'predictions.png')
    plt.savefig(plot_filename)
    print('Gradient descent finished.')

    # Plot the cost for all iterations
    plot_cost(cost_vector)
```

```
    min_cost = np.min(cost_vector)
    argmin_cost = np.argmin(cost_vector)
    print('Minimum cost: {:.5f}, on iteration #{}'.format(min_cost,
argmin_cost+1))

    return theta
```

To avoid excessive fluctuations and ensure that the coefficients do not take extreme values, we used compute_cost_regularised function instead of compute_cost function in gradient descent.py.

Setting the learning rate to 0.1, the number of iterations to 100 and regularization parameter as 1 generates the following graph.





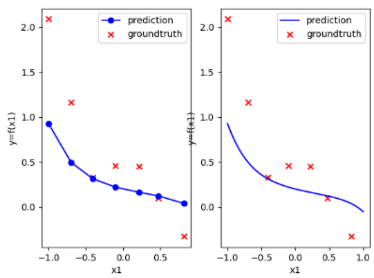
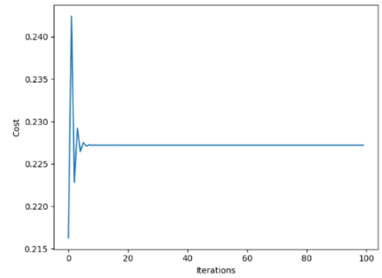
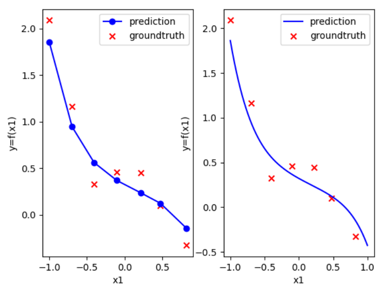
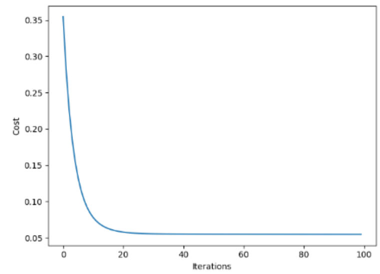
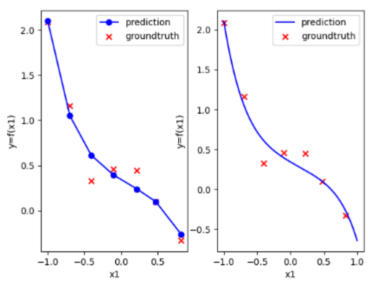
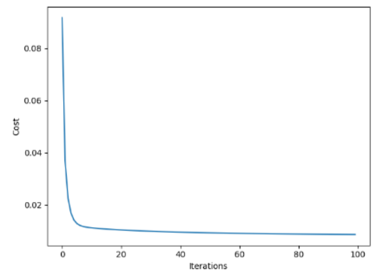
Minimum cost: 0.05478, on iteration #100

Observations:

The learning rate determines how aggressive each step the algorithm makes. When the learning rate is too high, the gradient descent inadvertently increases rather than decreasing the training error. When the learning rate is too low, training not only becomes slower, but also sometimes becomes stuck with a high training error and diverges.

The regularisation parameter (λ) is used in addition to the learning rate. As the magnitude of the fitting parameters increase, there will be an increasing penalty on the cost function which is dependent on the squares of the parameters as well as the magnitude of λ.

We risk underfitting the data if the λ value is too high, as the model won't be able to learn enough about the training data to make useful predictions. If the λ value is too low, we risk overfitting the data because the model will learn too much about the specifics of the training data and will be unable to generalise it.

S.No	Learning Rate	λ value	Figure 1 (cost.png)	Figure 2 (predictions.png)
1	0.1	10		
2	0.1	1		
3	0.1	0.001		

The above table shows the under fit, fit and over fit graphs respectively.