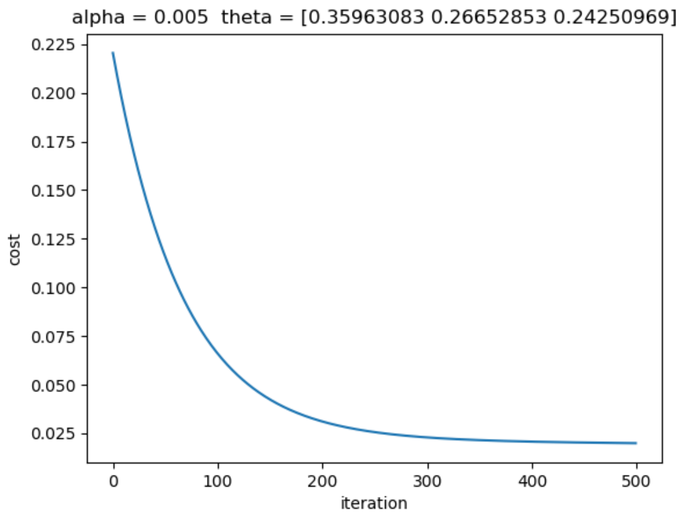
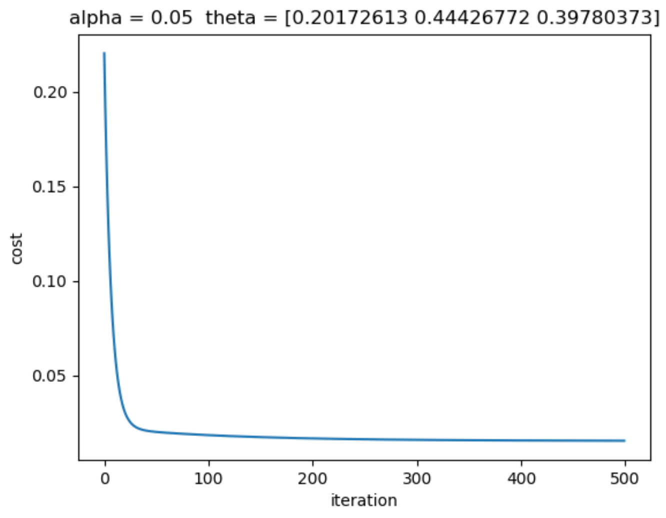
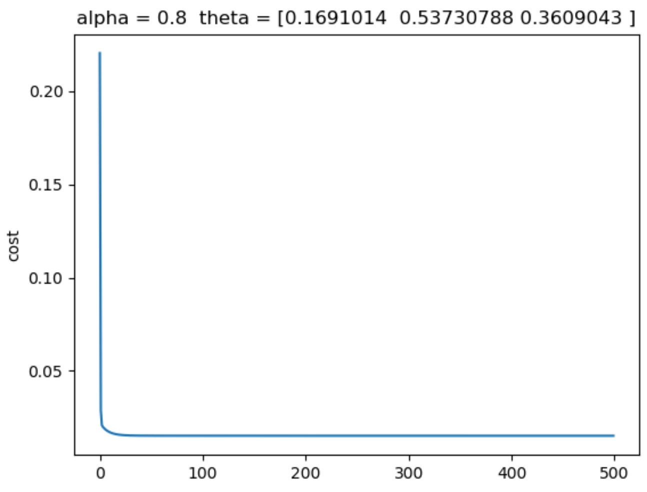
CS 596 Homework Assignment 2
Machine Learning

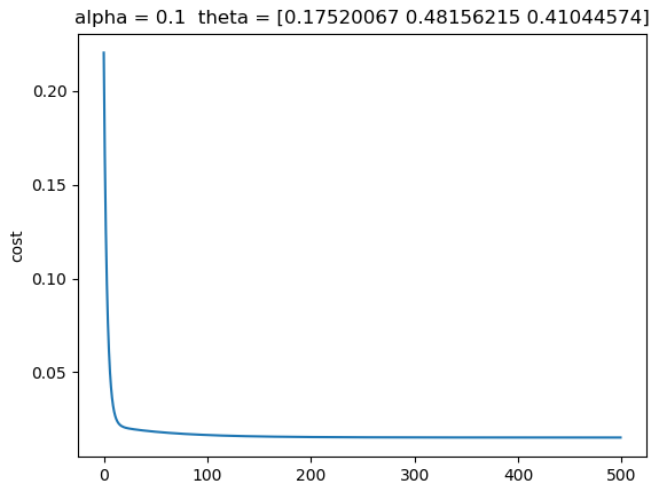
Module 1:

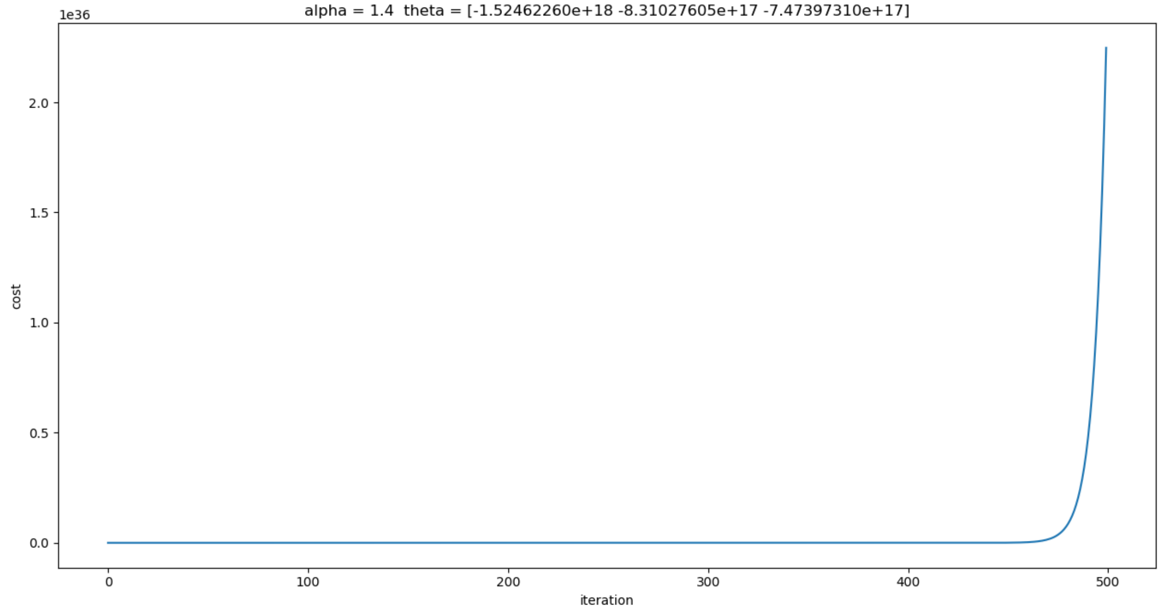
Samples with 500 Iterations:



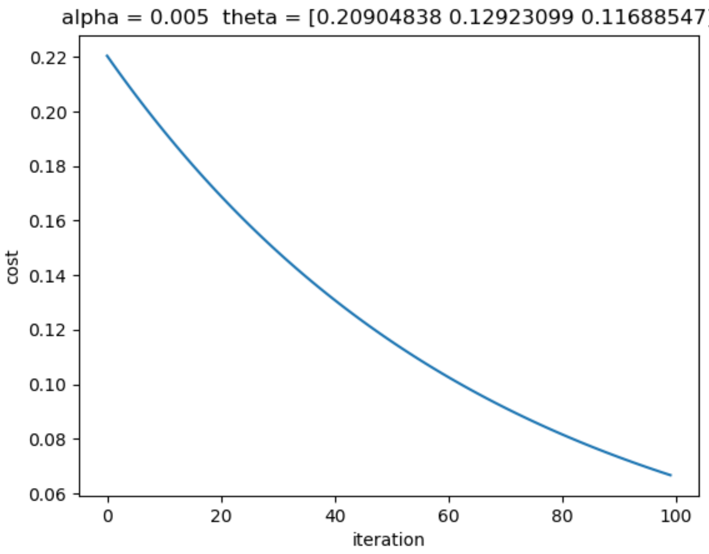


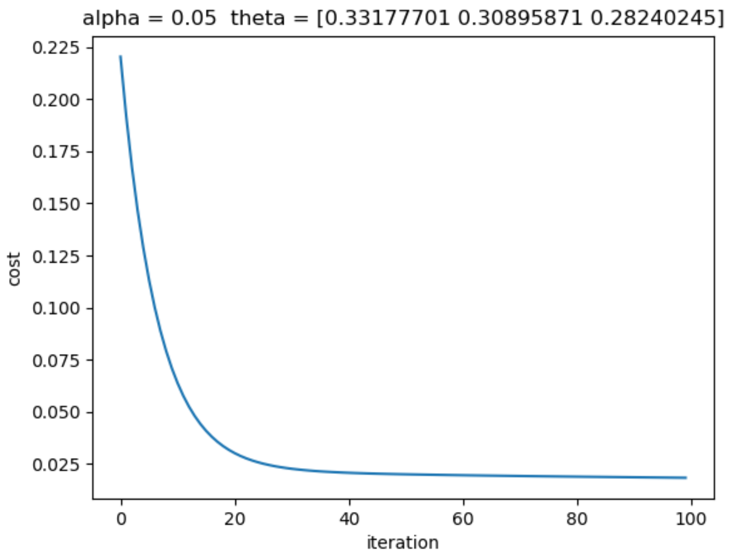


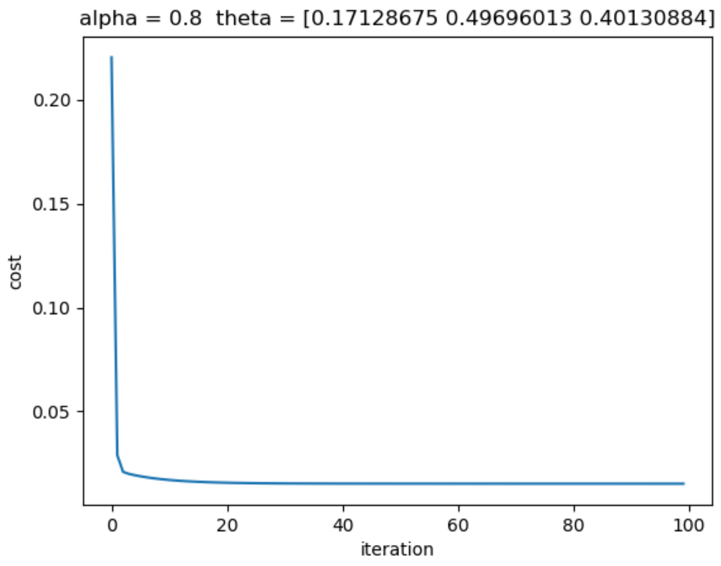


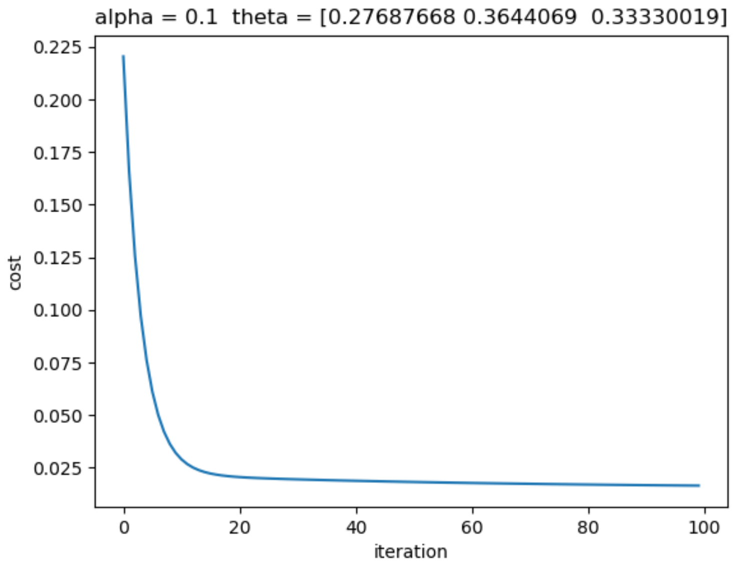


Samples with 100 Iterations:









- We can observe that, whenever the alpha i.e. the learning rate is reduced, the convergence curve appears to be really small, and thus we have a slow convergence.
- If the alpha is too large, the theta may not decrease on every iteration and may not converge like in the figure 5 of the 500 iterations.
- And with the less number of iterations, the convergence curve either requires a bigger alpha to reach the minimum cost or else it would not be able to reach the minimum cost. The more the number of iterations, the better chances of reaching the minimum cost.
- However, the more the number of iterations, the more time it will take to compute the value.

Module 2:

Normalization is a technique used to change the values of the numeric columns in a dataset to a generalized common scale. This is done without distorting any differences in the ranges of values. Normalization is usually only required when the features have different ranges or are too high.
For writing the normalization functions, I first created 2 functions stored in the normData.py file. The 2 of the normalization functions used are Min-Max Normalization and Mean Normalization.

Min Max Normalization:

In the min max normalization, the minimum and the maximum values from each of the X columns is gathered and the respective minimum value is subtracted from each element of the X columns. This element is then divided by the range i.e. the max value – min value. These elements are then stored in the final matrix and returned.
This steps are performed on all the columns except from the last column i.e. the prediction column or the Y. The function returns a new updated matrix.

```
def minMaxNorm(origMat):

    noRow = origMat.shape[0]
    noCol = origMat.shape[1]
    finalMat = np.zeros(origMat.shape)

    for i in range(0, noCol):
        minVal = np.min(origMat[:, i])
        maxVal = np.max(origMat[:, i])
        rangeVal = maxVal - minVal
        for j in range(0, noRow):
            newTemp = (origMat[j, i] - minVal) / rangeVal
            finalMat[j, i] = newTemp
    return finalMat
```

Mean Normalization:

In the mean normalization technique, the mean value of a column is subtracted from each of the elements in the respective column and then the result is divided by the range i.e. the max value – min value. These elements are then stored in the final matrix and returned.
This steps are performed on all the columns except from the last column i.e. the prediction column or the Y. The function returns a new updated matrix.

```
def meanNorm(origMat):

    noRow = origMat.shape[0]
    noCol = origMat.shape[1]
    finalMat = np.zeros(origMat.shape)

    finalMat[:, noCol-1] = origMat[:, noCol-1]

      for i in range(0, noCol-1):
            minVal = np.min(origMat[:, i])
            maxVal = np.max(origMat[:, i])
            rangeVal = maxVal - minVal
            mean = np.mean(origMat[:, i])
            print(minVal, maxVal, rangeVal, "Min, Max, Ran")
            for j in range(0, noRow):
                newTemp = (origMat[j, i] - mean) / rangeVal
                finalMat[j, i] = newTemp
                # print(origMat)
        return finalMat
```

Module 3 & Module 4:

In the GD.py, the gradientDescent() function is used to calculate the gradient descent from the input matrix. The functions takes the X columns or the input features, a Y column or the ground truth, theta and alpha as the cost and the learning rate and the number of iterations.

To calculate the cost function (a type of a score associated with the current set of weights which can be minimized to improve the accuracy of a model):

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(\theta^T\left(x^{(i)}\right) - y^{(i)}\right)^2$$

To calculate the gradient descent (used to find the minimum of a function):

Repeat {

$$\theta_J = \theta_J - \propto \frac{\partial J(\theta)}{\partial \theta_J}$$

$$\theta_J = \theta_J - \propto \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right).x_j^{(i)}$$

}

Gradient Descent:

- So in the function gradientDescent(), the entire procedure to calculate the cost and the gradient descent is put in a loop ranging from 0 to the number of iterations, given as an input with the parameter of the function.
- Firstly, our prediction (y0) is made with having a dot product of the X and the theta.
- We then compute the difference (gdDiff) between our prediction (y0) and the ground truth values (Y).
- Applying the gradient descent formula to the gathered values, we compute the dot product of the difference (gdDiff) and the transposed matrix of X and then divide the result with m which is the number of values in the ground truth column (Y).
- This is then multiplied with the learning rate or alpha and then subtracted from the original cost.
- The entire process is repeated until the number of iterations times.

Cost Function:

- To derive the cost value, we simply use the values gathered during the gradient descent and use it in the cost function formula.
- I.e. We compute the square of the difference (gdDiff) between our prediction (y0) and the ground truth values (Y).
- We divide the result with (2*m), take the sum of the result and then append it to a separate list (arrCost).
- This separate list contains all the computed values of the cost functions at each iteration and then this is used to plot the graph.

The graph is plotted using the final arrCost list and the convergence curve is displayed having the total cost at each iteration.

```
def gradientDescent(X, y, theta, alpha, numIterations):
    residualError = 0
    m = len(y)
    arrCost = [];

    transposedX = np.transpose(X) # transpose X into a vector

    for iteration in range(0, numIterations):
        y0 = np.dot(X, theta)
        # print("y0 ", y0)

        # Error: Our prediction - the ground truth
        gdDiff = y0 - y
        gradient = (1/m) * np.dot(transposedX, gdDiff)

        change = [alpha * x for x in gradient]
        theta = np.subtract(theta, change)  # or theta = theta - alpha * gradient

        atmp = np.sum((gdDiff)**2/(2*m))
        arrCost.append(atmp)

    return theta, arrCost
```

The average error and the standard deviation is computed,
One of the gathered error rate is 0.1727749787556687 ≈ 0.17
The standard deviation is 0.1268844229789746 ≈ 0.13
The average error is the average of the precision of the estimated values whereas the standard deviation is simply the difference between the regression line and the data at each value of the independent variable.