Queen Mary University of London

Department of Electrical Engineering & Computer Science

ECS708P Machine Learning

Assignment 1: Part 1 – Linear Regression

Hasan Emre Erdemoglu

5 November 2020

Table of Contents:

1 Linear Regression with One Variable

Task 1 is to modify the method in *calculate_hypothesis.py*. Since there is a single variable only the function $h_\theta(x) = \theta_0 x_0 + \theta_1 x_1$ can be directly written as the hypothesis term. For accommodating more variables automatically, I will be using vector notation and write the hypothesis as $h_\theta(x) = \overline{\theta^T}\,\bar{x}$. The code for this part can be found below, necessary changed as shown with bold. From now on code segments will be boxed as seen below to separate code from the report.

```
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 = np.dot(X[i], theta)
    ########################################
    return hypothesis
```

Please note that this is a generic use which also extends on using multiple number of variables. Therefore, this implementation will also solve the first question of Task 2. Hence, the explanation will be omitted there.

Calculate hypothesis was not used in the gradient descent method. Instead the hypothesis was manually embedded to the method. This was fixed by commenting out parts that manually embeds the hypothesis; and instead the function in *calculate_hypothesis.py* was used. For briefness; the code segments which are unrelated with the changes made are omitted by using "(…)"; the code can be found in the zip file in its entirety.

```
def gradient_descent(X, y, theta, alpha, iterations, do_plot):
    (…)
    # 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):
            # hypothesis = X[i, 0] * theta[0] + X[i, 1] *
theta[1]
            ########################################
            # 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):
            # hypothesis = X[i,0] * theta[0] + X[i,1] * theta[1]
            ########################################
            # 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
(…)
    return theta
```

Upon first run of the *ml_assgn1_1.py* the cost function explodes and not a good fit is achieved. This is due to very high learning rate. The following results are achieved when a learning rate of 1 or 0.1 is used (0.1 has not identical results but the overall shapes are similar):

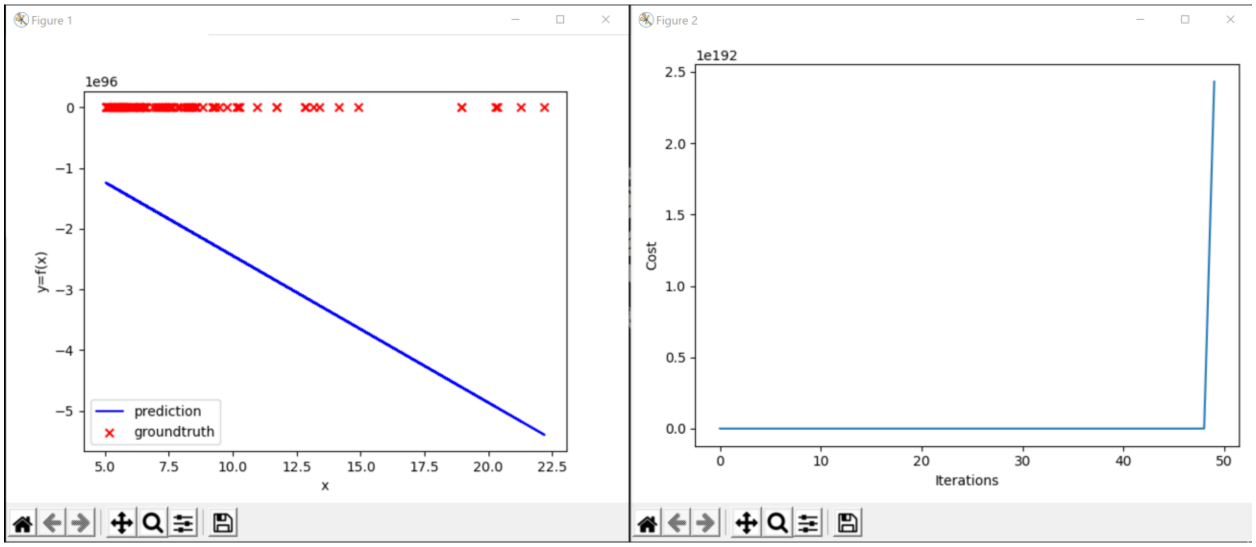

Figure 1. Results with lr=1 (Min loss = 172570.09522 at iteration 1)

In second implementation, the learning rate is reduced to 0.001; this time the function converges but very slowly. A learning rate of 0.01 also gives a similar result but it converges faster and slightly fits better to the training data.

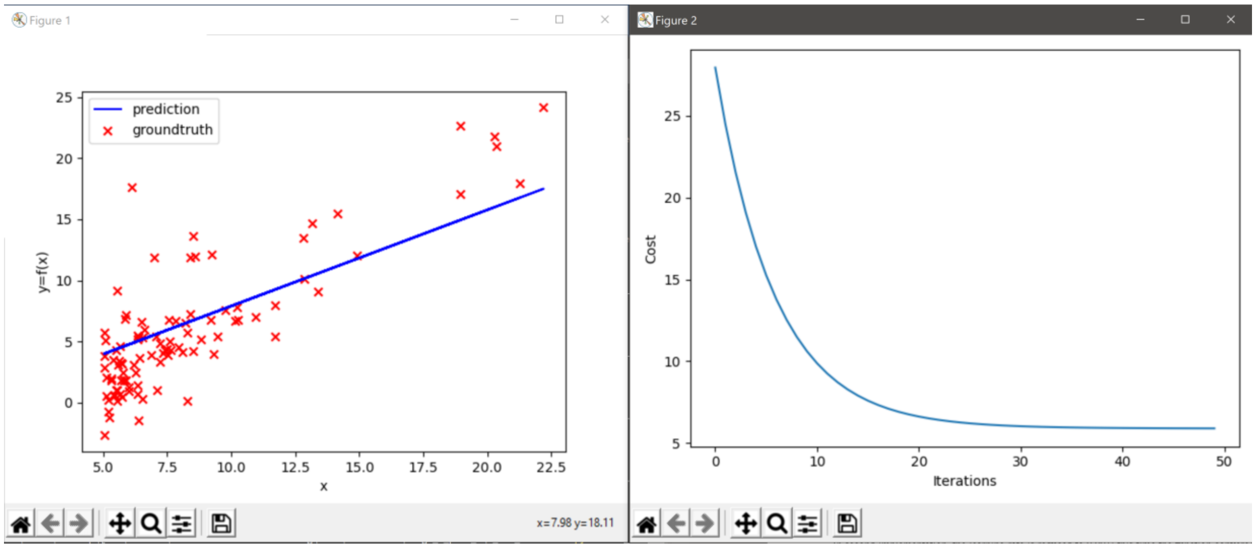

Figure 2. Results with lr=0.001 (Min loss = 5.89 at iteration 50)

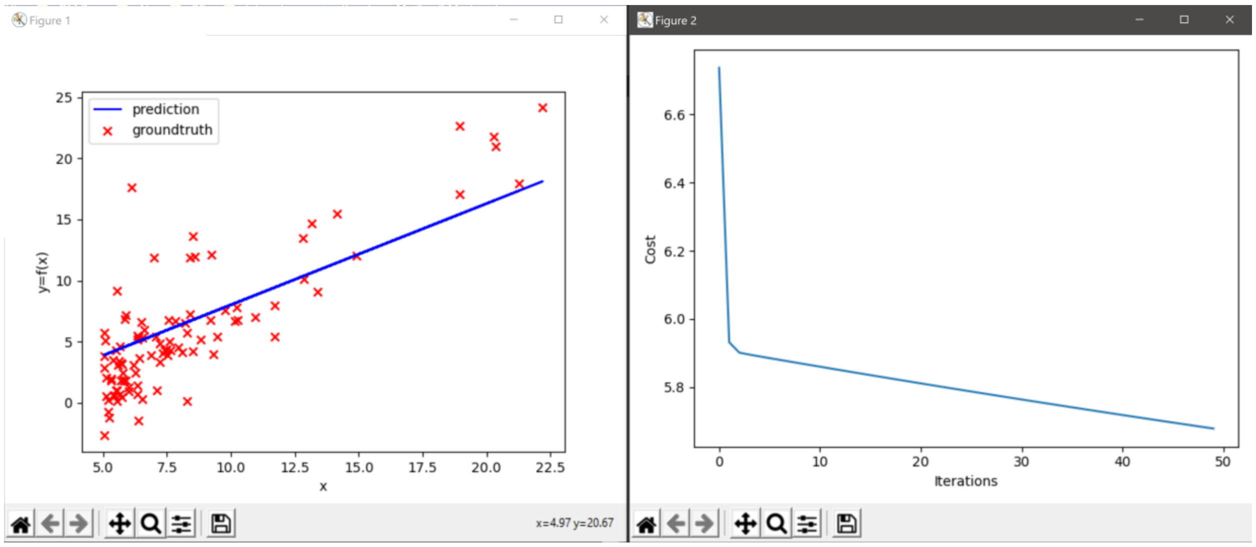

Figure 3. Results with lr=0.01 (Min loss = 5.67 at iteration 50)

Note that these experiments were made both using the given gradient implementation and the altered version that I have prepared. The results are identical, implying the implementation of *calculate_hypothesis.py* is correct.

The best learning rate value can be evaluated by iteratively repeating the experiments with changing the learning rate; however, for the sake of this question it is not necessary. From *ml_assgn1_1.py* only the variable **alpha** has been changed.

2 Linear Regression with Multiple Variables

As in the previous section, the *calculate_hypothesis.py* function was written in vectoral form. Due to this reason it can be directly translated here. I will omit the details as they are equivalent with Task 1.

For testing effect of different alpha values, I will use alpha values of 10, 1, 0.1, 0.01 and 0.001; the same approach I used in Task 1. The only parameter that will be changed in *ml_assgn1_2.py* will be variable named by **alpha**. The table below will show the final theta values achieved by using the corresponding learning rates. For completeness, this table will also predict the prices of houses with areas 1650 and 3000 & with bedrooms 3 and 4, respectively. The following code segment is appended to the end of *ml_assgn1_2.py* in order to add new samples. These samples are normalized using z distribution; whose mean and standard deviation is taken from the initial dataset. After normalizing these data with respect to the dataset; hypothesis is calculated again to produce predictions on these data.

```
########################################

# Write your code here

# Create two new samples: (1650, 3) and (3000, 4)

print('The final theta values found:')

print(theta_final, '\n')


new_samples = [[1650,3], [3000,4]] # new ndarray


# normalize and add bias:

# Make sure to apply the same preprocessing that was applied to the training data

new_samples = (new_samples - mean_vec) / std_vec

column_of_ones = np.ones((new_samples.shape[0], 1))

new_samples = np.append(column_of_ones, new_samples, axis=1)


print('New (Normalized) Samples:')

print(new_samples, '\n')


# Calculate the hypothesis for each sample, using the trained parameters theta_final

hyp = []

for i in range(len(new_samples)):

    hyp.append(calculate_hypothesis(new_samples, theta_final, i))


# Print the predicted prices for the two samples

print('Area of 1650, 3 bedroom house price: ', hyp[0])

print('Area of 3000, 4 bedroom house price: ', hyp[1])

########################################
```

The table and the outputs are as follows:

Table 1. Experiment Results (red result diverged), green not converged yet, after 100 iterations

alpha	theta $(\theta_0, \theta_1, \theta_2)$	(1650,3) prediction	(3000,4) prediction	Min Loss (value, iteration)
10	**(-7.19e105, -0.44e121, -0.22e121)**	**9.40e120**	**-3.32e121**	**5.59e12, 1**
1	**(3.40e5, 1.09e5, -6.59e3)**	**293081**	**472277**	**2.04e9, 48**
0.1	**(3.40e5, 1.09e5, -5.93e3)**	**293214**	**472159**	**2.43e9,100**
0.01	**(2,16e5, 6.14e4, 2.00e4)**	**183865**	**36034**	**1.06e10, 100**
0.001	**(3.24e4, 9,93e3, 4,93e3)**	**26863**	**50475**	**5.38e10, 100**

Note that the 'e' terms stand to exponential, implying the 2 significant figure is multiplied by 10 to the power x, where 'x' is the factor to the right of 'e'. Also note that the loss is MSE loss and the magnitude of the loss is due to the scale of the house prices.

From Table 1, it is visible that learning rate of 10 was too much; theta values came to be very large and negative and the predictions are very large and there are negative house prices. The loss also overshoots just like the case seen in Figure 1. As seen below; predictions have nothing to do with the ground truth.

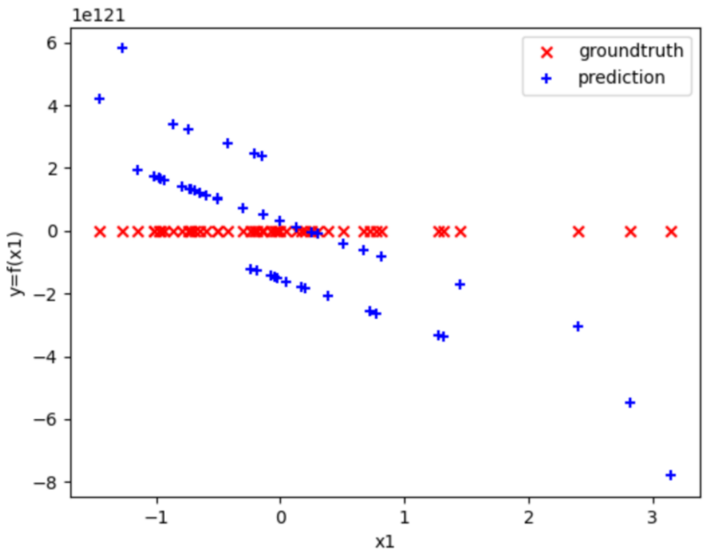

Figure 4.Predictions when lr=10

Using a learning rate of 1; minimum loss is achieved at 48th iteration and the loss stabilized (remain constant) after that. The predictions were mostly on par with the ground truth. A learning rate of 0.1 also yielded very similar results; but the minimum loss is achieved at iteration 100 where the max iteration is limited by 100. This means although the training loss is stabilized, and correct results are mostly reached; this learning rate requires more iterations to be perfectly stabilize. Learning rate of 1 is more robust and faster converging than a learning rate of 0.1.

As learning rate is further reduced, such as to 0.01; minimum loss is achieved at the final iteration, but the predictions worsen compared to learning rates 1 and 0.1. This is because of slowness caused by the small learning rate; 100 iterations are simply not sufficient to fit to the dataset perfectly.

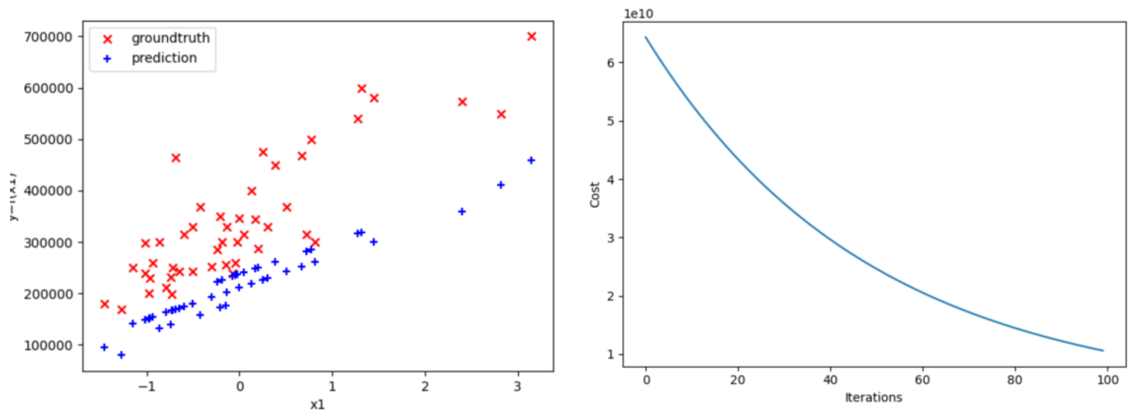

Figure 5. Predictions and loss when lr = 0.01

The decrease in loss indicates that the training is in right direction, the loss has not plateaued yet. In predictions vs ground truth graph the effect of this is seen, as predictions follow a similar fashion to ground truth however undershoots as there have not been enough passes. Due to this reason the predictions on new house samples are coming to be lower than they are supposed to be. A learning rate of 0.001 further proves this point as the predictions undershoot more; minimum loss is at the final iteration. For these cases, the learning rate is not enough to reach to minima in 100 iterations.

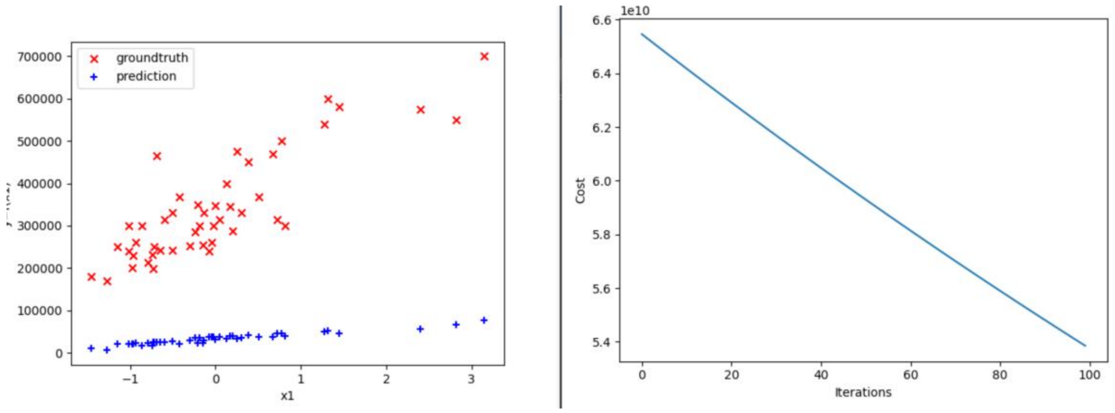

Figure 6. Predictions and loss when lr = 0.001

For completeness, the code for the *gradient_descent.py* can be found below. The hypothesis is calculated, then elementwise error between hypothesis and output is accumulated and theta is updated accordingly. As gradient descent is based on derivate of the cost with respect to theta it must satisfy the following set of equations:

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

$$\sigma = \frac{dJ_\theta}{d\theta} = \frac{1}{m}\left[\sum_{i=1}^{m}\left(h_\theta\left(x^i\right) - y^i\right) \times \frac{dh_\theta\left(x^i\right)}{d\theta}\right]$$

The derivative term $\frac{dh_\theta(x^i)}{d\theta}$ will yield to 1 for the bias term; and x^i for all the other terms. In the code the summation is kept in sigma as the model passes through all the samples; where np.dot(…) operation handles vector multiplication and summation. When a pass through all samples are complete "theta_temp" is updated by multiplying the sigma by alpha over m and subtracting the value from theta_temp itself. In equation form:

$$\theta_i = \theta_i - \alpha\,\sigma$$

```
(…) # imports
def gradient_descent(X, y, theta, alpha, iterations, do_plot):
(…)
    # 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):
            #hypothesis = X[i, 0] * theta[0] + X[i, 1] * theta[1]
            ########################################
            # 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 +=  np.dot((hypothesis - output),X[i])
            ########################################/

        # update theta_temp
        ########################################
        # Write your code here
        # Update theta_temp, using the values of sigma
        theta_temp -= alpha/m * sigma
        ########################################/
        # copy theta_temp to theta
        theta = theta_temp.copy()
        (…)
    return theta
```

3 Regularized Linear Regression

Again *calculate_hypothesis.py* will follow the same approach in Tasks 1 & 2. The L2 regularized cost function is given by the question and has already been implemented. In *gradient_descent.py* the following updates (in bold) are made:

As it is not clearly visible in the code, the parameter "l' is added to the list of parameters defining gradient descent function. This term is the regularization parameter and supplied to the function in *ml_assgn1_3.py*. The details of how the updates are made will be explained below:

```
(…) # imports

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

(…) # definitions, documentation

    # 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, i)

            ########################################/

            output = y[i]

            #########################################

            # Write your code here

            # Adapt the code, to compute the values of sigma for all the elements of theta

           # both does the update over sample and summation

            sigma += np.dot((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] -= alpha/m * sigma[0]

        theta_temp[1:] = theta_temp[1:] * (1 - alpha*l/m) - alpha/m * sigma[1:]

        ########################################/

        # 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)

    (…)

    return theta
```

The regularized cost function is given below:

$$J_\theta = \frac{1}{2m}\left[\sum_{i=1}^{m}\left(h_\theta\left(x^i\right) - y^i\right)^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right]$$

The derivation of the derivative will follow from derivative 2, with the exception that the derivative of the regularization term must also be considered.

$$\frac{dJ_\theta}{d\theta} = \frac{1}{m}\left[\sum_{i=1}^{m}\left(h_\theta\left(x^i\right) - y^i\right) \times \frac{dh_\theta\left(x^i\right)}{d\theta}\right] + \frac{\lambda}{m}\left[\sum_{j=1}^{n}\theta_j\frac{d\theta_j}{d\theta}\right]$$

In the equation form this would give the expression stated in the *Assignment_1_Part_1.pdf* document. Normally, the equation above could be used for all theta value; however, the bias term must not be regularized as well. Therefore, the update function for first theta value will follow from Task 2, whereas remaining theta values will be updated by the equation above.

$$\theta_0 = \theta_0 - a\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_0^{(i)}$$

$$\theta_j = \theta_j\left(1 - a\frac{\lambda}{m}\right) - a\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)}$$

Figure 7. Iterative Update Equations for Task 3 (From Assignment_1_Part_1.pdf)

After replacing *compute_cost* with *compute_cost_regularized* as instruted, the script *ml_assgn1_3.py* can be run. The following output is generated:

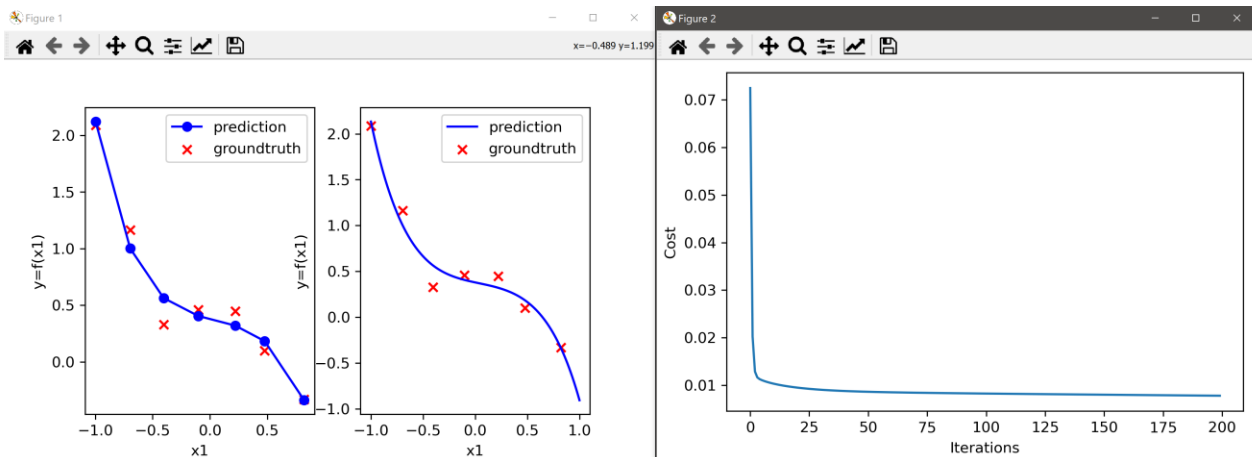

Figure 8. Outputs of ml_assgn1_3.py with alpha=1 l=0 (Min cost = 0.00780 @ iteration 200)

For finding the best alpha and regularization an experiment can be conducted. For each pair (**alpha, l)** the experiment can be run to understand how minimum cost changes after 5000 iterations. Note that, I have changed the initial number of 200 iteration limit to 5000 as my preliminary analysis showed that when alpha is increased above 1, the algorithm diverges. For values less or equal to one; 200 iterations are not enough for the model to fully stabilize. I understand this as the function of fifth order should be able to fit to all of the points (2 degree of freedom less than number of samples); if allowed to overfit to the datapoints. See Figures 9-11 for comparison between iterations. Following the same convention with Task 2, lets use 1, 0.1 and 0.01 for the **alpha** values. For **l** values lets work with 100, 1, 0,01 and 0.

Note that when **l** is 0, we can assume that we are only optimizing over **alpha**. Working with pairs of experiments give additional information about how **l** might affect the best **alpha** value. The table below shows the minimum cost and the iteration when this cost is achieved for different values of **alpha** and **l**.

Table 2. Experiment Results (red result alpha only), best pair (green) after 5000 iterations

Alpha \ l	**100**	**1**	**0.01**	**0**
1	Overflow	0.05295, 192	0.00803, 5000	0.00151, 5000
0.1	0.2563, 225	0.05295,1624	0.00824, 5000	0.00676, 5000
0.01	0.2563, 1882	0.5295, 5000	0.00926, 5000	0.00865, 5000

When **lambda** term is zero, even 5000 iterations were not enough to fit perfectly to the data. As alpha increases, the speed of the gradient descent increases; therefore, minimum cost decreases. For (1,100) case, the calculations diverged and lead to an overflow.

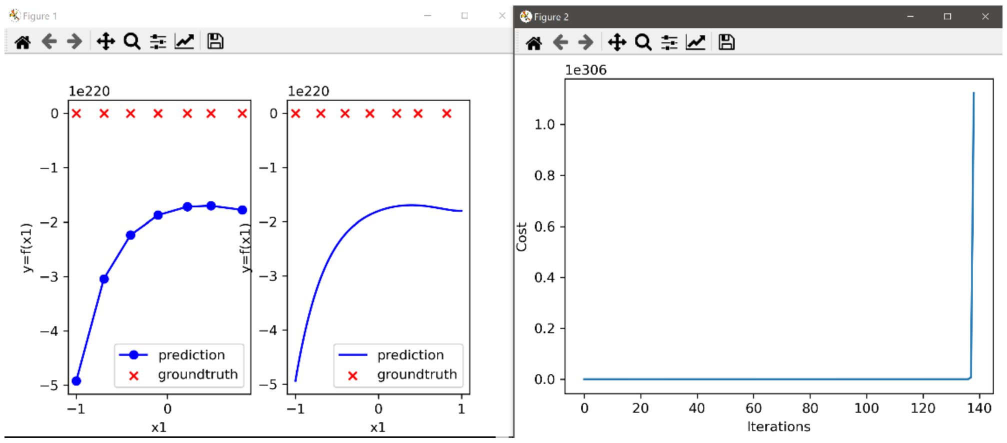

Figure 9. Results at iteration 200, alpha=10, l=0, model fails to converge.

As seen in Figure 9, an **alpha** value of 10 with no regularization fails to converge. In fact, any value which is above 1 fails to converge.

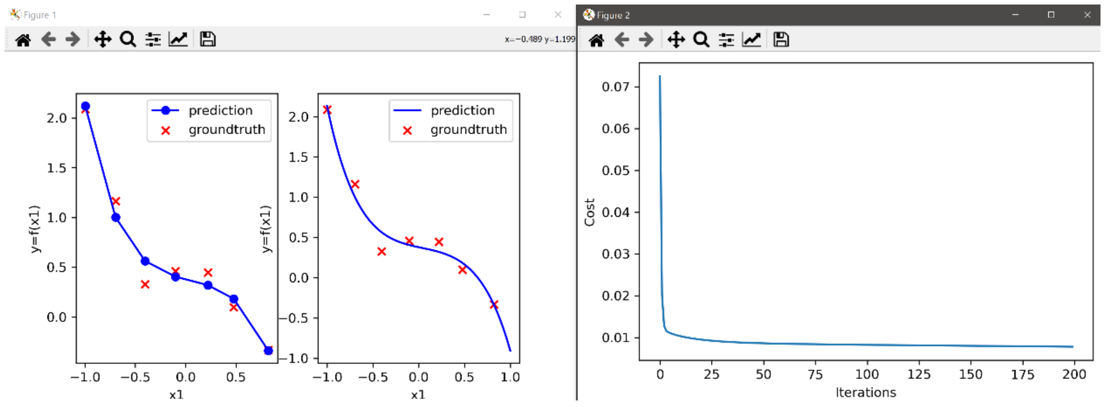

Figure 10. Results at iteration 200, alpha=1, l=0

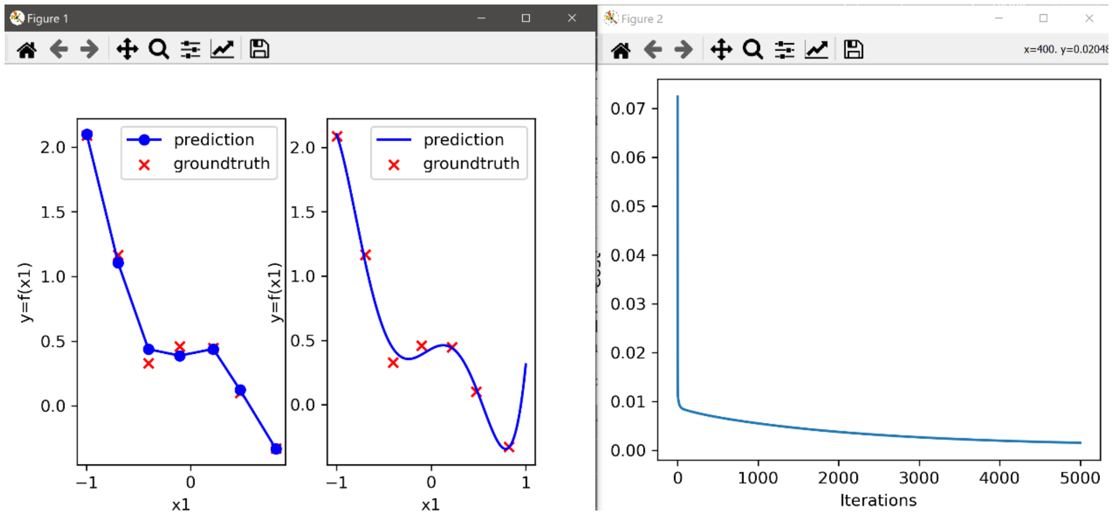

Figure 11. Results at iteration 5000, alpha=1, l=0

As seen in Figures 10 and 11, given enough time; without any regularization, the vanilla model overfits to the given data.

When **lambda** increases and **alpha** is low the following pattern happens. As **lambda** decreases, the fit gets punished less and may extend upon y direction fitting more to the data. It is not ideal to increase **lambda** too much.

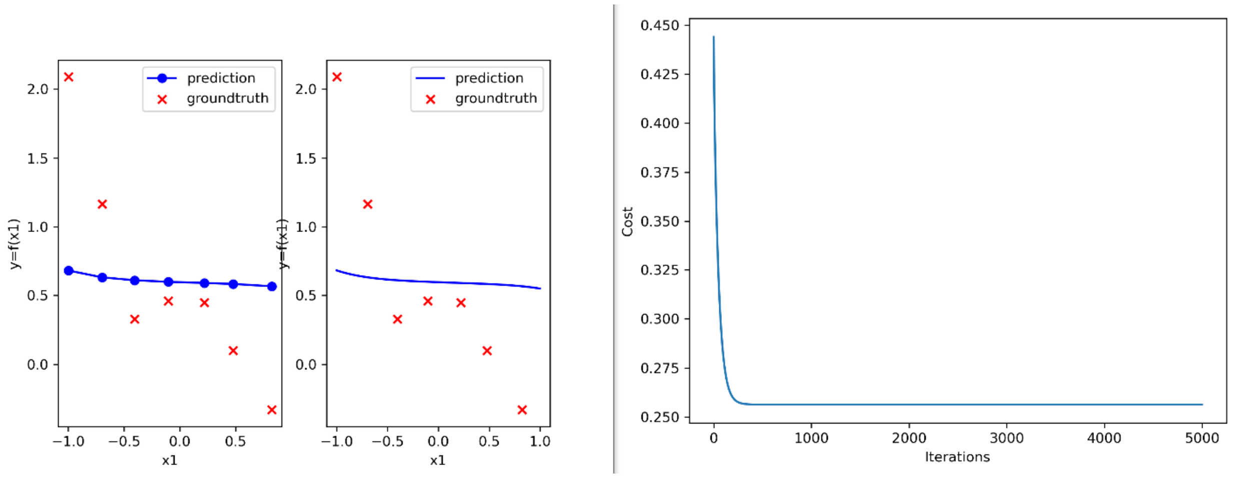

Figure 12. Results at iteration 5000, alpha=0.01, lambda=100

When **lambda** is small, even a small amount can significantly help with the overfitting. Comparing Figure 11 with Figure 13, the regularization effect is very visible:

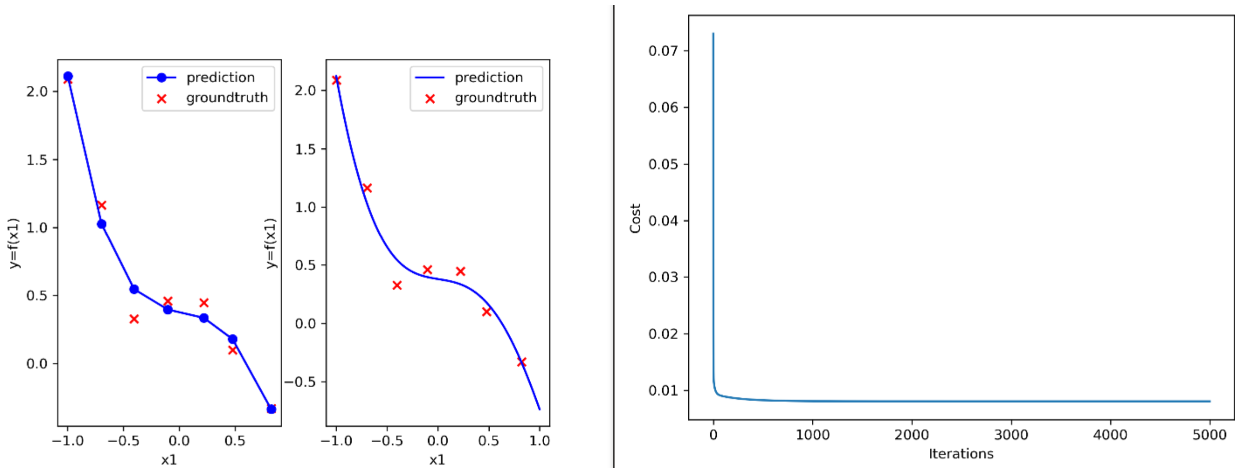

Figure 13. Results at iteration 5000, alpha=1, lambda=0.01

Going back to Table 2, a **lambda** value of 0.01, does not produce a local minimum before iteration 5000, which is the final iteration for all tested values of **alpha**. Increasing the **lambda** value to 1, increases the minimum cost to 0.05295, but the function achieves this minimum cost in less than 200 iterations. For example, when (**alpha**,**l**)=(1, 1) the minimum cost is 0.05295 which is achieved in 192 iterations. After that, the cost starts to increase slowly, therefore the function will not start continuing fitting the datapoints, overfitting in later iterations. While lower **lambda** values provided better minimum cost, the minimum cost is seen at the last iteration and it is certain when they will provide a local minimum (if there were infinite iterations) and whether the regularization would be enough to avoid overfitting.

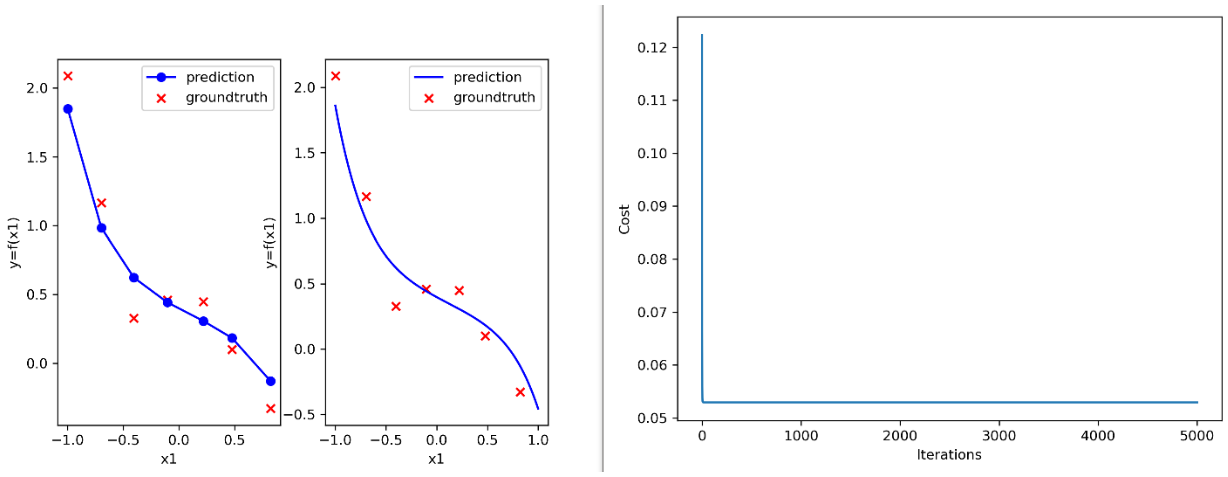

Figure 14. Results at iteration 5000, alpha=1, lambda=1

Therefore, the middle value on regularization term provides optimal performance and fast convergence at the expense of having a higher minimum cost associated with it. With a stable learning rate, both the training speed of the model and likeliness of overfitting can be optimized. Table 2 shows the minimum costs achieved within 5000 iterations with a testbed of 9 learning rate and regularization factor pairs. Figures 9 to 14 show various behavior when **alpha** and **lambda** are changed together also analyzing how changing one affect the behavior of the other term.

For creating this analysis **alpha** and **l** variables of *ml_assgn1_3.py* document has been changed 9 times, the results and plots are recorded every time.