Queen Mary
University of London

Machine Learning

Assignment1

Part1

Spyridon Roumpis

181004877

1. Linear Regression with One Variable

Task1

In this task we had to modify the function *calculate_hypothesis.py* to return the predicted value for a single specified training example. Below is shown the line of coded that has been added for the prediction.

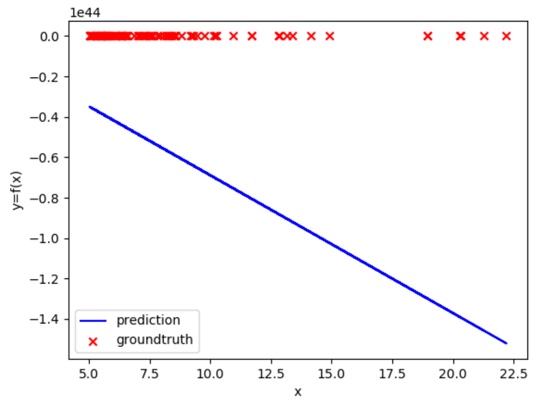
hypothesis = X[i,0]*theta[0] + X[i,1]*theta[1]

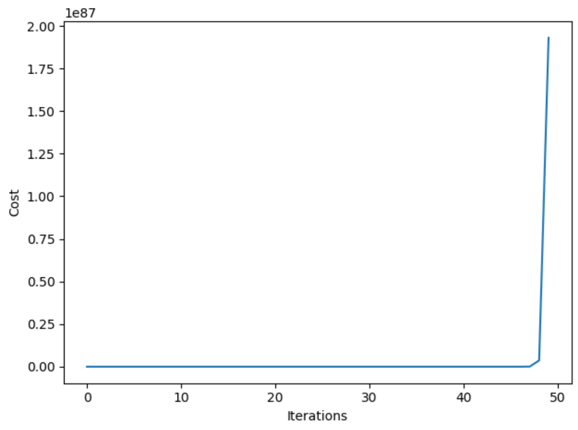
Then we had to modify the gradient_descent function to use the hypothesis function we modified above.
The below line was added in two parts of the gradient_descent function, inside the two for loops.

hypothesis = calculate_hypothesis(X,theta,i)

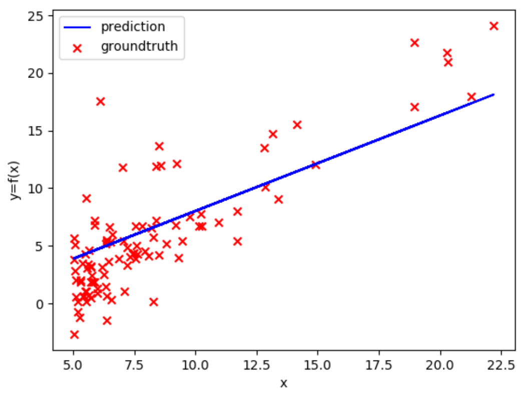
Finally, we will run some experiments on the values for the learning rate, alpha to see for which one the model fit better.

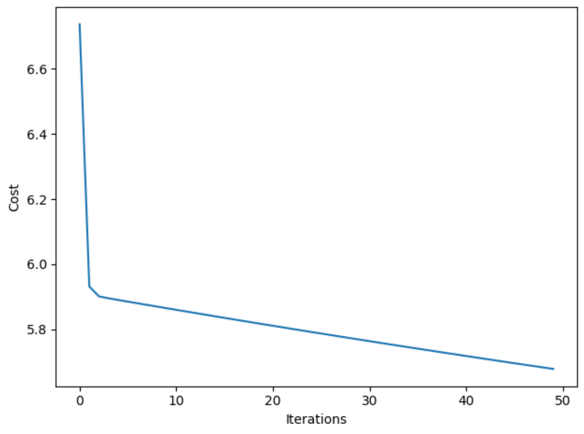
- **For alpha = 0.1 -→** Minimum cost: 1370.27709, on iteration #1



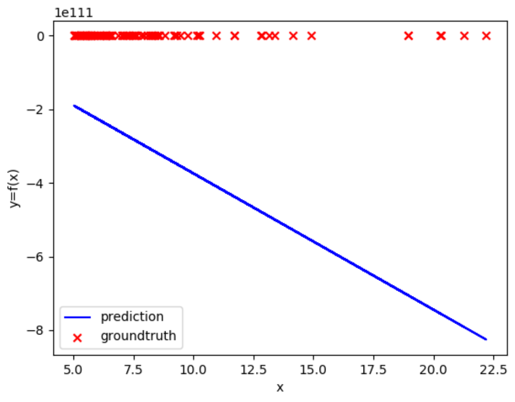


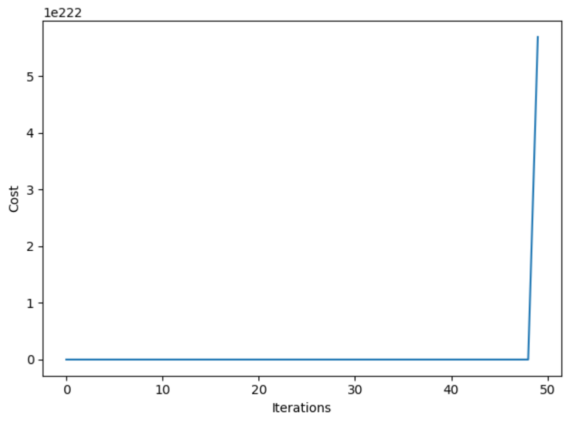
- **For alpha = 0.01 -→** Minimum cost: 5.67829, on iteration #50





- **For alpha=2** -→ Minimum cost: 698788.07091, on iteration #1





It can be seen that for alpha=0.01 the gradient Descent function fits really well while for the other two values of the learning rate is behaving really badly, it keeps overshooting again and again the local minimum. In general as the learning rate gets bigger and bigger the cost function is behaving really badly.

2. Linear Regression with Multiple Variables

Task2

In this task we will now look at linear regression with two variables (three including the bias). We will need to modify the functions calculate_hypothesis and gradient_descent to support the new hypothesis function. Our new hypothesis function's code should be sufficiently general so that we can have any number of extra variables.

Below is shown the line of coded that has been added in calculate_hypothesis.

hypothesis = np.dot(X[i,:],theta)

And for the gradient_descent the bold lines below represent the new lines of code.

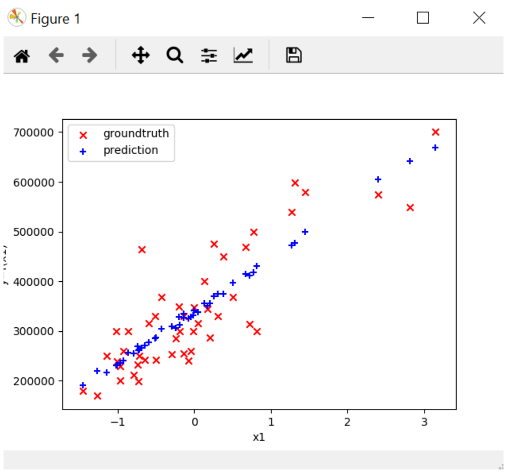
for i in range(m):

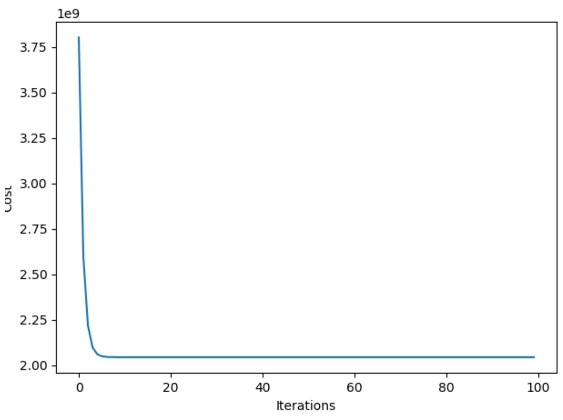
```
            hypothesis = calculate_hypothesis(X,theta,i)

            output = y[i]
            sigma = sigma + (hypothesis - output)*X[i,:]

        # update theta_temp
        theta_temp=theta_temp-(alpha/m)*sigma

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

Then we want to see see how different values of alpha affect the convergence of the algorithm.

- **alpha =1** → Minimum cost: 2043280050.60283, on iteration #48

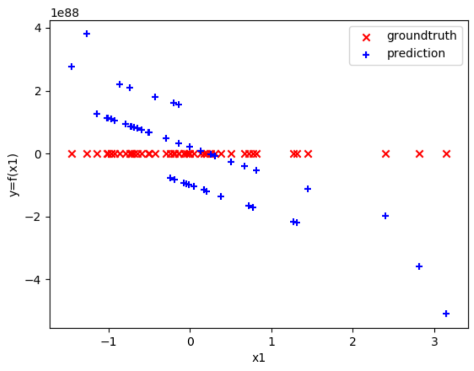
  Theta = [340412.65957447 109447.79646964 -6578.35485416]

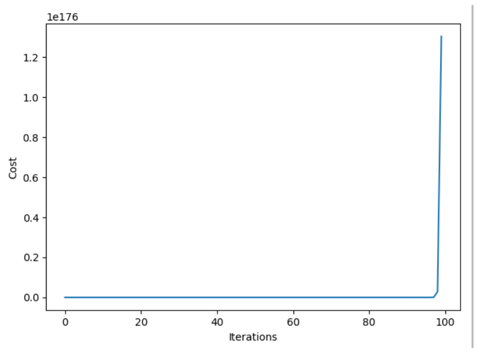




- **alpha=5** →Minimum cost: 1122043184133.05811, on iteration #1

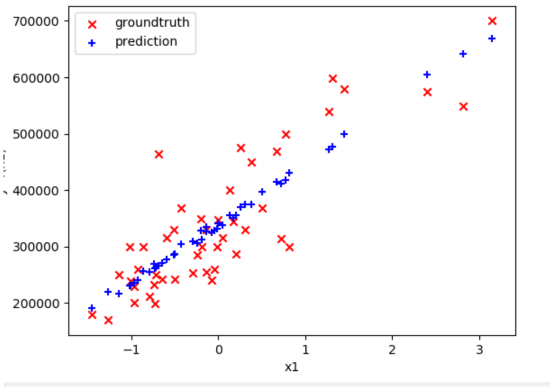
  theta = [-4.48411088e+72 -9.14324521e+87 -9.14324521e+87]

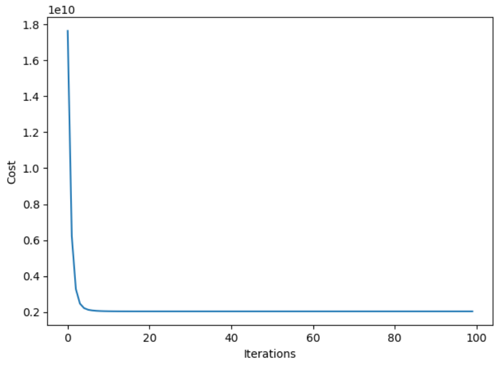




- **alpha=0.5 →**Minimum cost: 2043280050.60283, on iteration #78

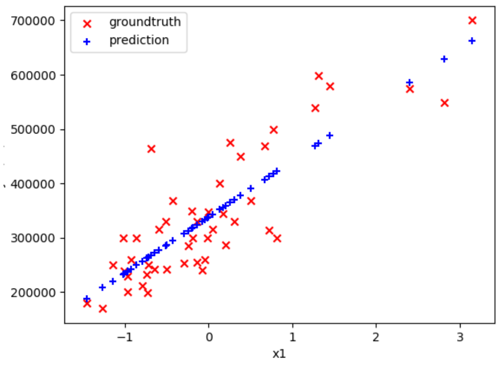
  theta = [340412.65957447 109447.7964687   -6578.35485322]

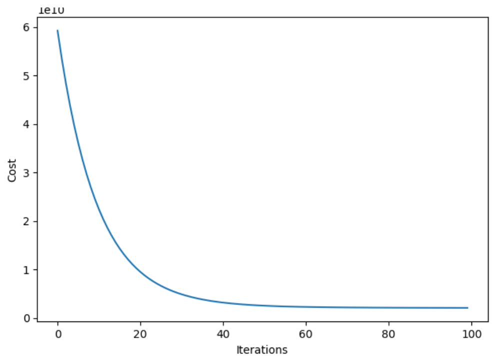




- **alpha=0.05 →**Minimum cost: 2062616003.82372, on iteration #100

  theta = [3.38397236e+05 1.03161481e+05 -3.22620198e+02]





We ran ml_assgn1_2.py to see how different values of alpha affect the convergence of the algorithm. From the above graphs it can be seen that for learning rate, a = 0.5 our cost function performs best.

Finally, we would like to use our trained theta values to make a prediction for the cost of a house with 1650 sq. ft. and 3 bedrooms and for a house with 3000 sq. ft. and 4 bedrooms.

We added the below lines in our code :

```
thetatrain=[340412.65957447,109447.7964687,-6578.35485322]

X1=[1650,3]
X2=[3000,4]


#Normalize
Xnorm1=(X1-mean_vec)/std_vec
Xnorm2=(X2-mean_vec)/std_vec

# After normalizing, we append a column of ones to X, as the bias term
column_of_ones = np.ones((Xnorm1.shape[0], 1))
column_of_ones = np.ones((Xnorm2.shape[0], 1))

# append column to the dimension of columns (i.e., 1)
Xnorm1 = np.append(column_of_ones, Xnorm1, axis=1)
Xnorm2 = np.append(column_of_ones, Xnorm2, axis=1)

hypothesis1=calculate_hypothesis(Xnorm1,thetatrain,0)
hypothesis2=calculate_hypothesis(Xnorm2,thetatrain,0)

print(hypothesis1,hypothesis2)
```

The result of it is for hypothesis1, for the house with 1650 sq. ft. and 3 bedrooms, the cost is 293081.4643351053, while for hypotehsis2, for the house with 3000 sq. ft. and 4 bedrooms, is 472277.85514620505.

3. Regularized Linear Regression

Task3

We need to modify the gradient_descent to use the compute_cost_regularised method instead of compute_cost. Next, modifying the gradient_descent to incorporate the new cost function. Again, we do not want to punish the bias term. This means that we use a different update technique for the partial derivative of θc, and add the regularization to all of the others.

We first modified the compute_cost_regularised to have the parameter lamda,l, as seen below.

```
J = (total_squared_error + l*total_regularised_error)/(2*m)
```

We also updated the calculate_hypothesis function using the same exactly line of code we used in task 2.

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

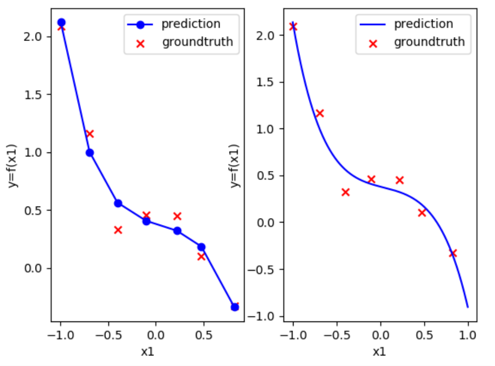
Finally, we updated the gradient_descent to use the compute_cost_regularised method instead of compute_cost and to calculate the hypothesis, the sigma and the thetas without punishing the bias term.

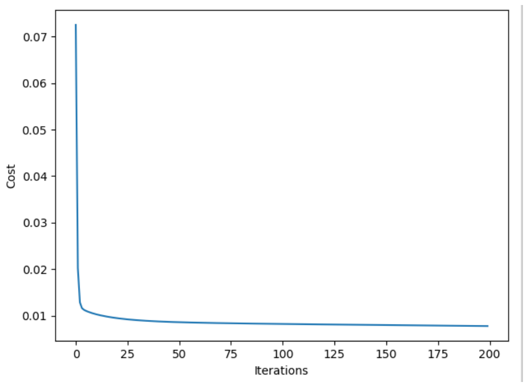
```
# Calculate the hypothesis for the i-th sample of X, with a call to the "calculate_hypothesis" function
        hypothesis = calculate_hypothesis(X,theta,i)

        output = y[i]

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

        sigma = sigma + (hypothesis - output)*X[i,:]

# update theta_temp, using the values of sigma

      theta_temp[0] = 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)
```

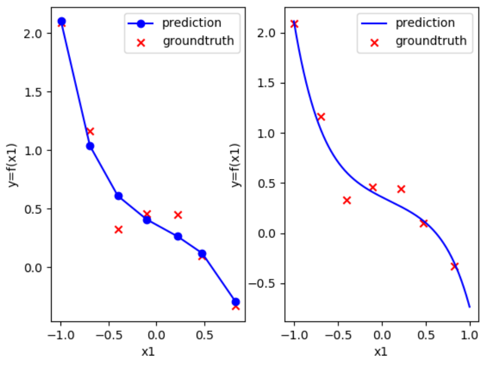
Experiments:

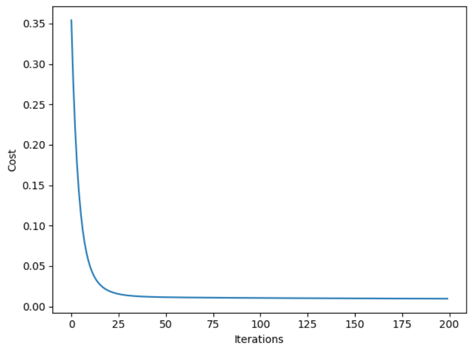
- **alpha=1, l=0 →** Minimum cost: 0.00780, on iteration #200



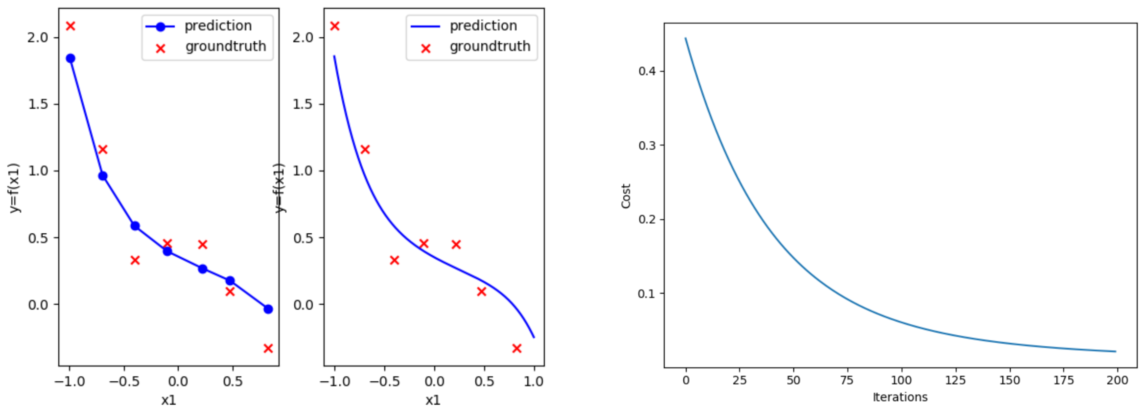


- **alpha=0.1, l=0 →** Minimum cost: 0.00957, on iteration #200



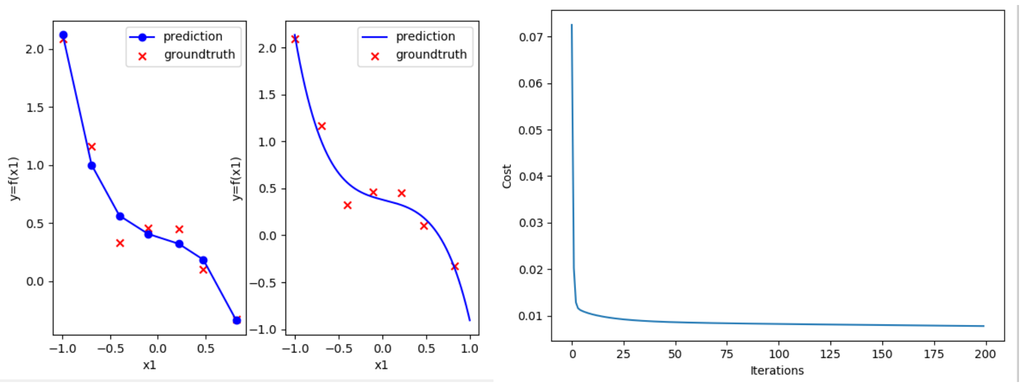


- **alpha=0.01, l=0 →** Minimum cost: 0.02119, on iteration #200

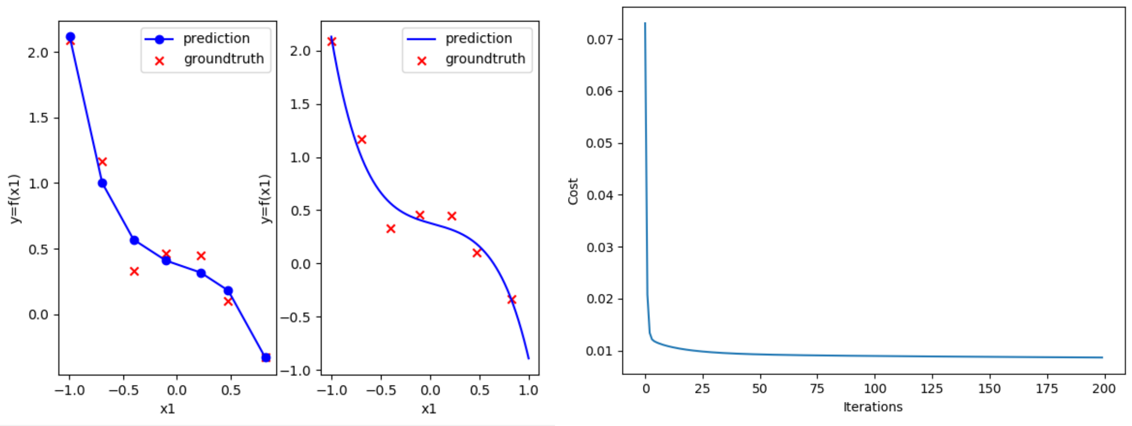


It can be seen that for alpha=1 we get the best minimum cost, so in order to have the best optimized function we will use a=1.
Now that we found the best value of a, so we can minimize our cost to we will run experiments on lamda, in order to find the best value.

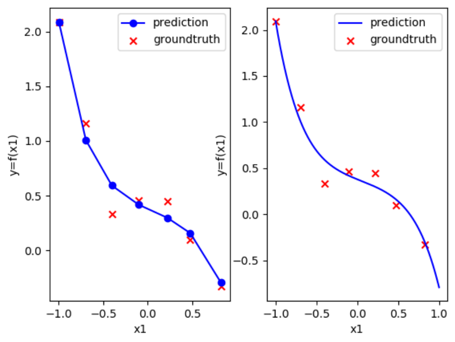
- **alpha=1, l=0 →** Minimum cost: 0.00780, on iteration #200

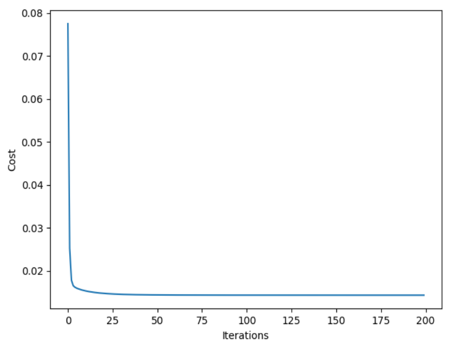


- **alpha=1, l=0.01 →** 0.00865, on iteration #200

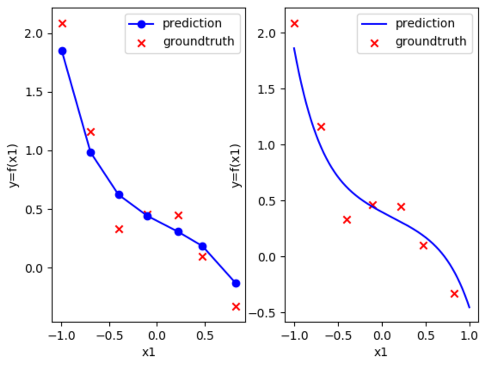


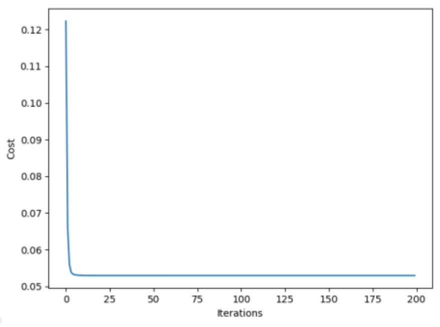
- **alpha=1, l=0.1 →** 0.01437, on iteration #200



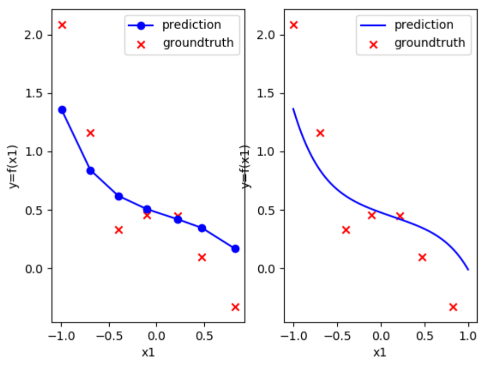


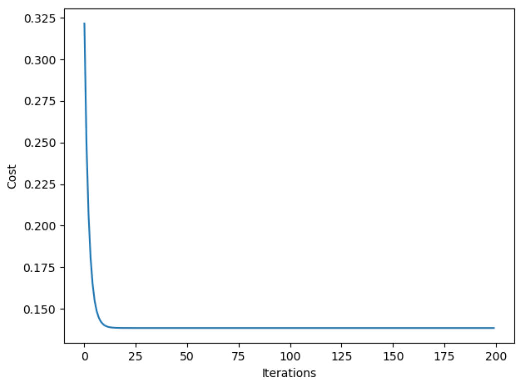
- **alpha=1, l=1 →** 0.05295, on iteration #192





- **alpha=1, l=5 →** 0.13838, on iteration #91





It can be seen that the bigger the value of lamda is the worse our function is behaving,