ECS708 Machine Learning

Guanzhen Wu

161189284

Assignment 1: Part 1 - Linear Regression

Task 1:

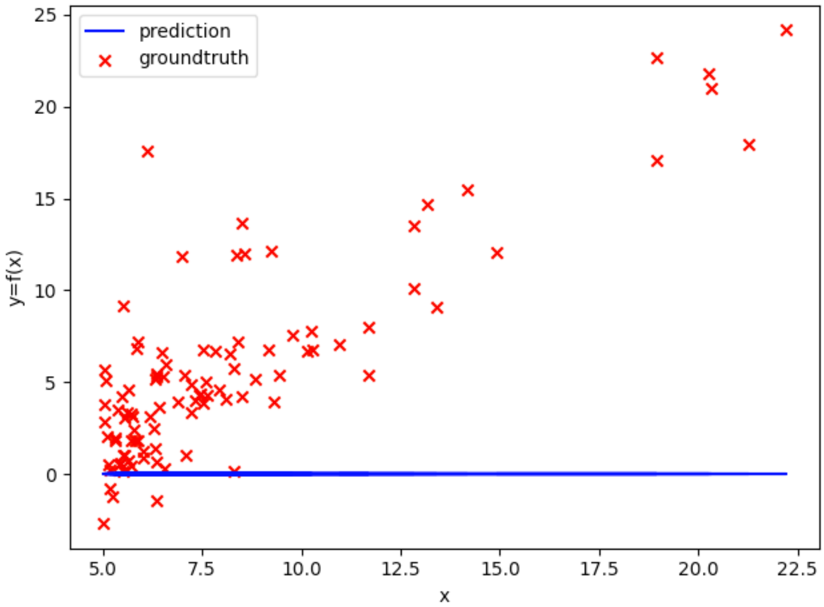
As the formula (1) shows, the hypothesis is equal to the sum of theta multiply X

$$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 \tag{1}$$

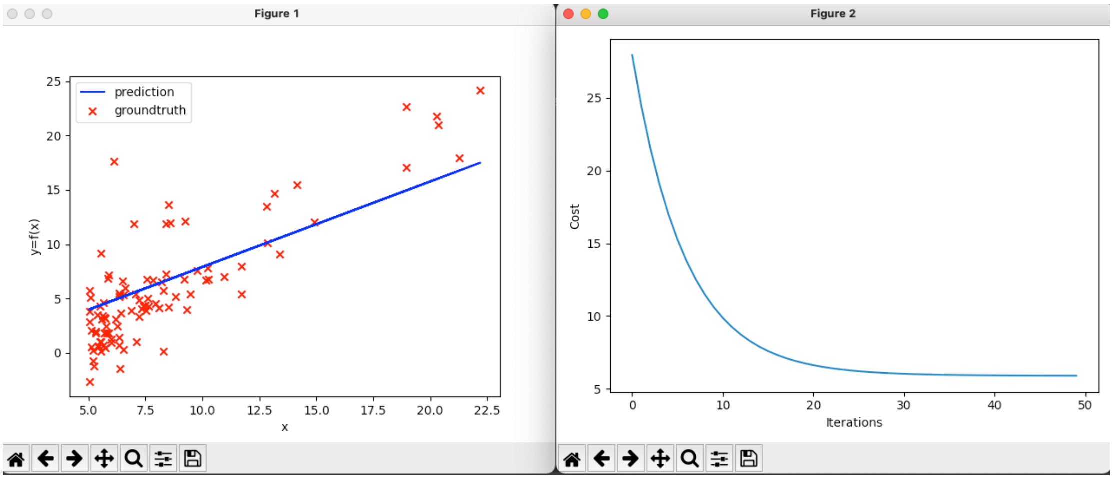
The implemented code is:

```
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 = X[i, 0] * theta[0] + X[i, 1] * theta[1]
###########################################
```

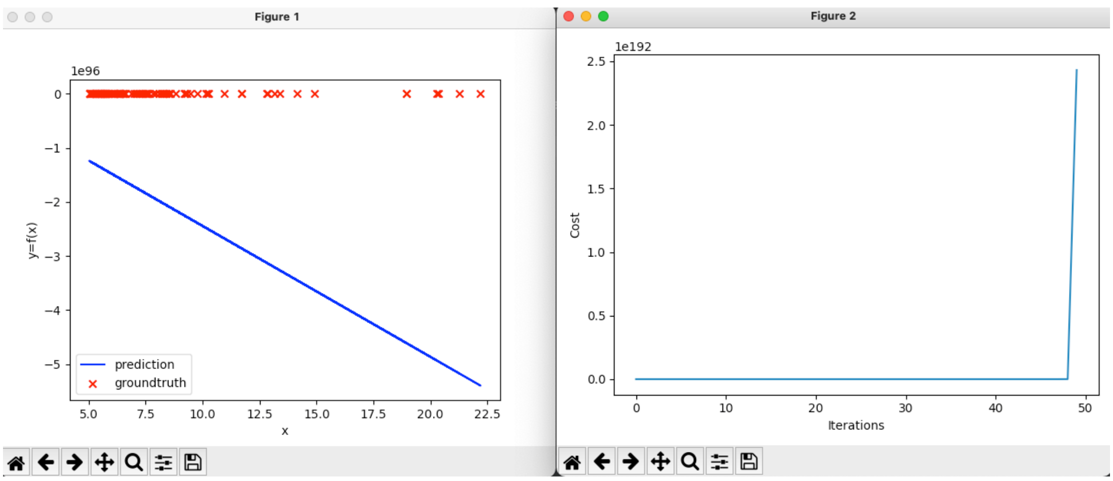
```
sigma = 0.0
for i in range(m):

    ########################################
    # 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)
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
```



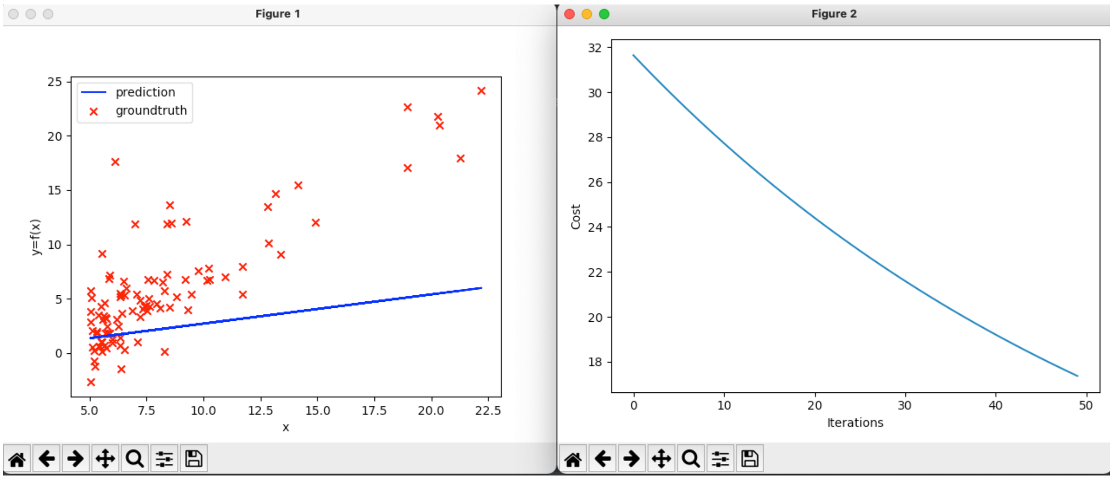
Original graphic



alpha = 0.001



Alpha = 1.0



Alpha = 0.0001

As shown in the figure above, after many tests, a suitable learning rate (alpha) value that can make the cost function converge is 0.001. When the alpha is too high, such as alpha = 1, the cost function cannot converge and grows exponentially with the increase of iteration. When the alpha value is too small, the cost function has not fully converged yet.

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. [5 points]

```
hypothesis = 0
for a in range(len(X[0])):
    hypothesis += X[i, a] * theta[a]
```

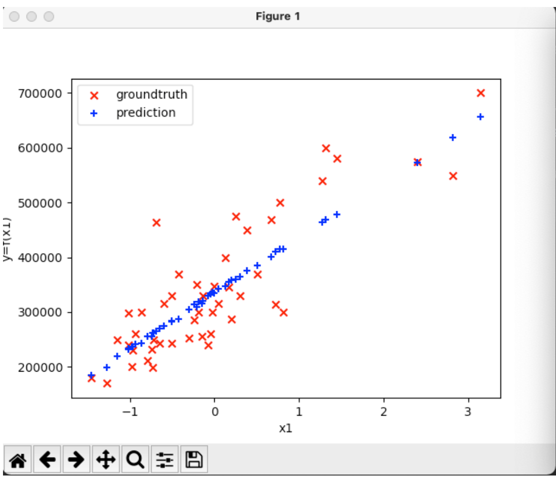
```
test = np.array([[1650, 3], [3000, 4]])

# Normalize
# tn, _, _ = normalize_features(test)

tn = (test - mean_vec) / std_vec


# After normalizing, we append a column of ones to X, as the bias term
column_of_ones = np.ones((tn.shape[0], 1))
# append column to the dimension of columns (i.e., 1)
tn1 = np.append(column_of_ones, tn, axis=1)
# tn2 = np.append(column_of_ones, tn2, axis=1)

print("theta_final is",theta_final)

h1 = calculate_hypothesis(tn1, theta_final, 0)
print("Prediction of [1650,3] is", h1)

h2 = calculate_hypothesis(tn1, theta_final, 1)
print("Prediction of [3000,1] is", h2)
```

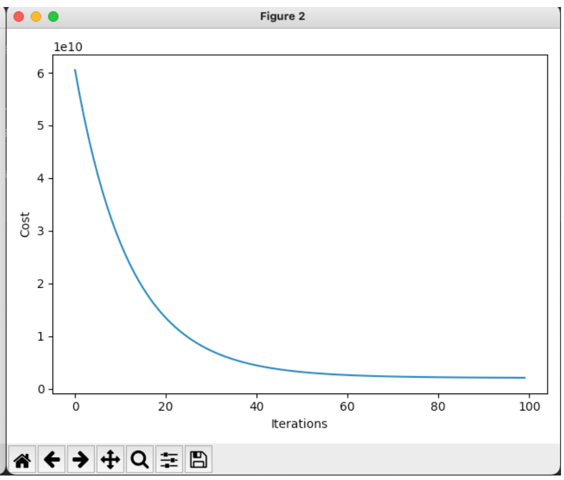
Run ml_assgn1_2.py and see how different values of alpha affect the convergence of the algorithm. Print the theta values found at the end of the optimization. Does anything surprise you? Include the values of theta and your observations in your report. [5 points]

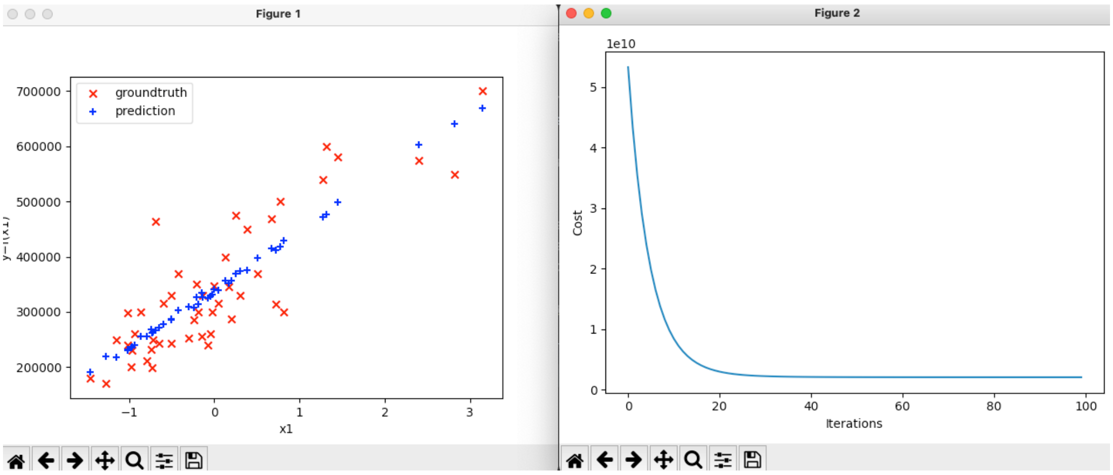
Excluding the learning rate that is too large and too small and adjust it within an appropriate range, it is found that the cost will gradually approach a uniform value after multiple iterations using different alphas which is about 2.04*10^9. The final theta and price prediction shows as below.

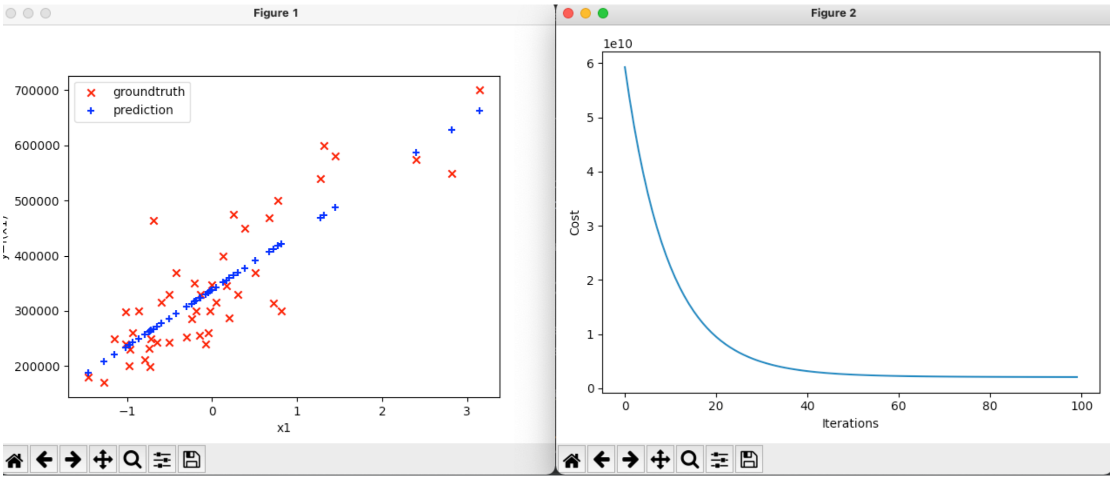
What surprises me is that prediction has become a discrete point. At the same time, I found that a larger alpha will make the prediction points quickly approach the final distribution area, which proves that the learning rate is faster. But the subsequent dozens of iterations only fluctuate slightly.

```
➜  2_multiple_variables python3.8 ml_assgn1_2.py
Dataset normalization complete.
alpha is  0.1
Gradient descent finished.
Minimum cost: 2043462824.61817, on iteration #100
theta_final is [340403.61773803 108803.37852266  -5933.9413402 ]
Prediction of [1650,3] is 293214.1635457116
Prediction of [3000,1] is 472159.98841419816
➜  2_multiple_variables python3.8 ml_assgn1_2.py
Dataset normalization complete.
alpha is  0.4
Gradient descent finished.
Minimum cost: 2043280050.60283, on iteration #100
theta_final is [340412.65957447 109447.79624289  -6578.35462741]
Prediction of [1650,3] is 293081.46438477084
Prediction of [3000,1] is 472277.8551080717
➜  2_multiple_variables python3.8 ml_assgn1_2.py
Dataset normalization complete.
alpha is  0.04
Gradient descent finished.
Minimum cost: 2102251697.27196, on iteration #100
theta_final is [334669.78929365  99541.59787385   3164.12175216]
Prediction of [1650,3] is 289554.4837724504
Prediction of [3000,1] is 464681.71368288004
```









Task 3 Note that the punishment for having more terms is not applied to the bias. This cost function has been implemented already in the function *compute_cost_regularised*. 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. [5 points]

For hypothesis:

$$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^3 + \theta_4 x_1^4 + \theta_5 x_1^5$$

```
##########################################
# Write your code here
# You must calculate the hypothesis for the i-th sample of X, given X, theta and i.
hypothesis = 0
for a in range(len(X[0])):
    hypothesis += X[i, a] ** a * theta[a]
##########################################/
```

```python
# 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)
```

```python
# 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)
```

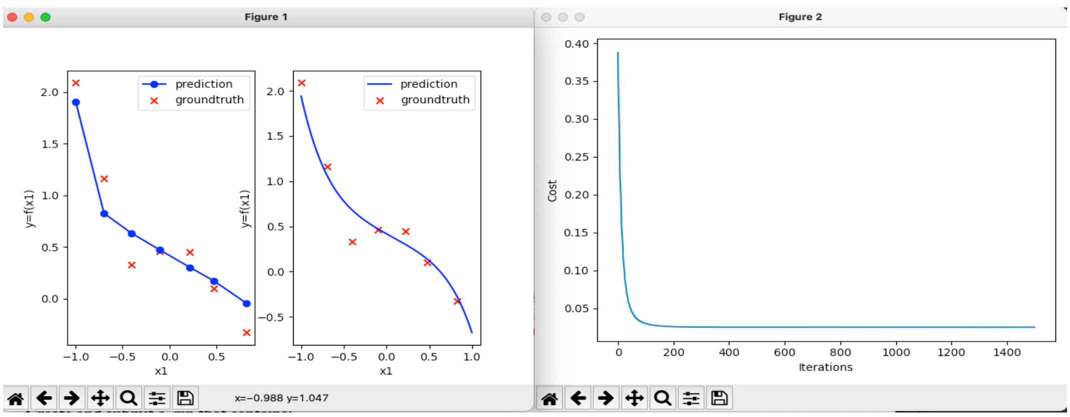
In *gradient_descent.py*, add the theta calculation as the formula below.

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

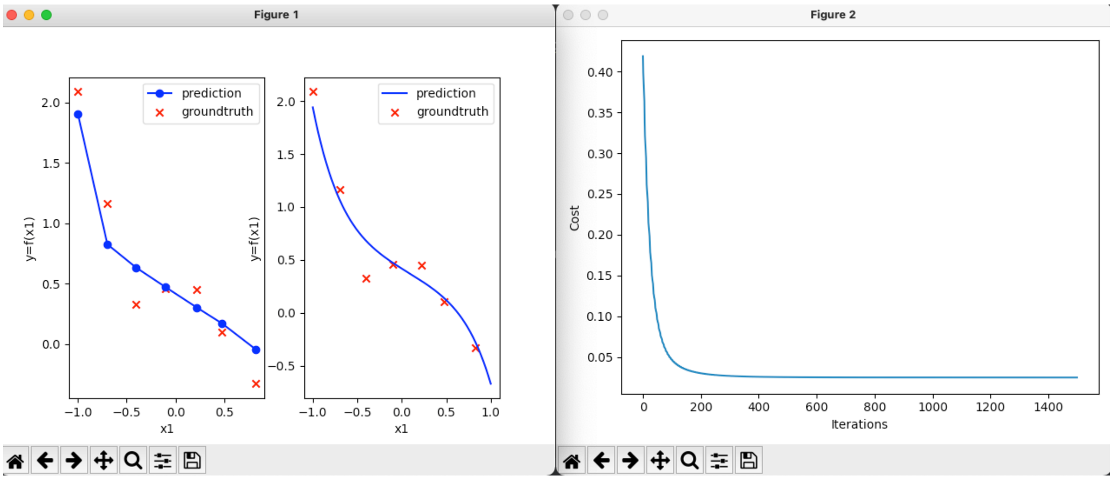
$$\theta_j = \theta_j\left(1 - a\frac{\lambda}{m}\right) - a\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)}$$

```python
    sigma = np.zeros((len(theta)))
    for j in range(len(theta)):
        for i in range(m):
            hypothesis = calculate_hypothesis(X, theta, i)
            ########################################
            # Write your code here
            # Calculate the hypothesis for the i-th sample of X, with a call to the "calculate_hypothesis" function

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

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

        if j == 0:
            theta_temp[j] = theta_temp[j] - (alpha * 1.0 / m) * sigma[j]
        else:
            theta_temp[j] = theta_temp[j]*(1 - alpha * l / m) - (alpha * 1.0 / m) * sigma[j]
        # 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_regularised(X, y, theta, l)
        cost_vector = np.append(cost_vector, iteration_cost)
```

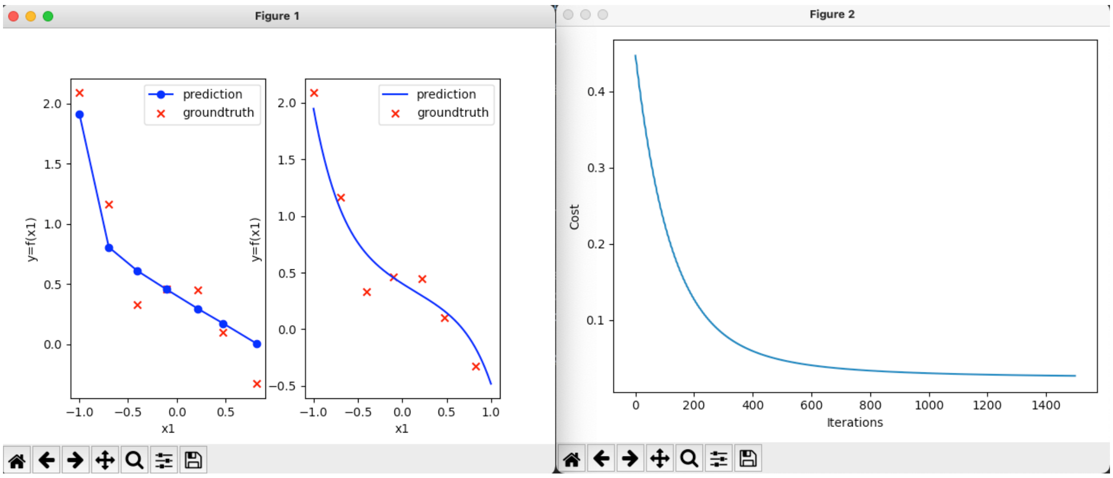
To find the best alpha, set the λ default to 1:



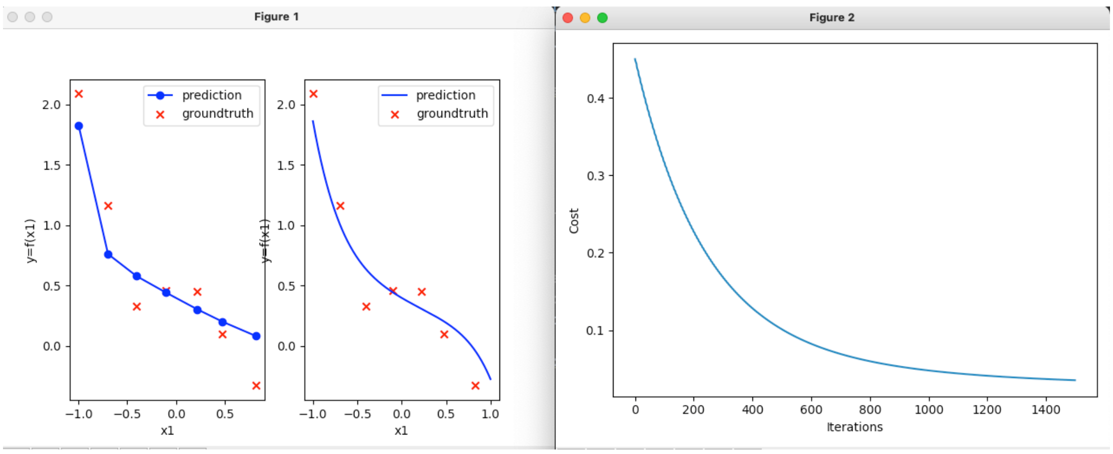
Alpha = 0.2 Minimum cost: 0.02705, on iteration #1498



Alpha = 0.1 Minimum cost: 0.02490, on iteration #1500



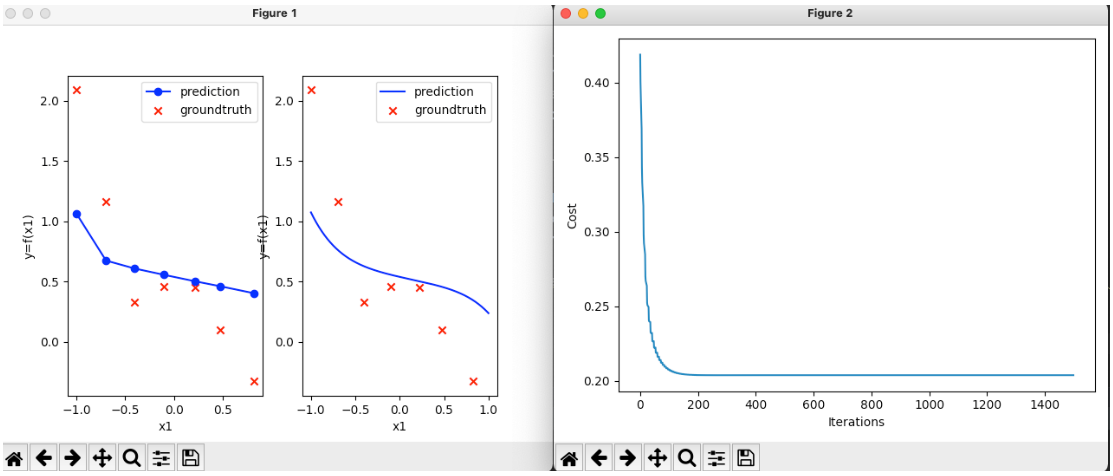
Alpha = 0.02 Minimum cost: 0.03522, on iteration #1500



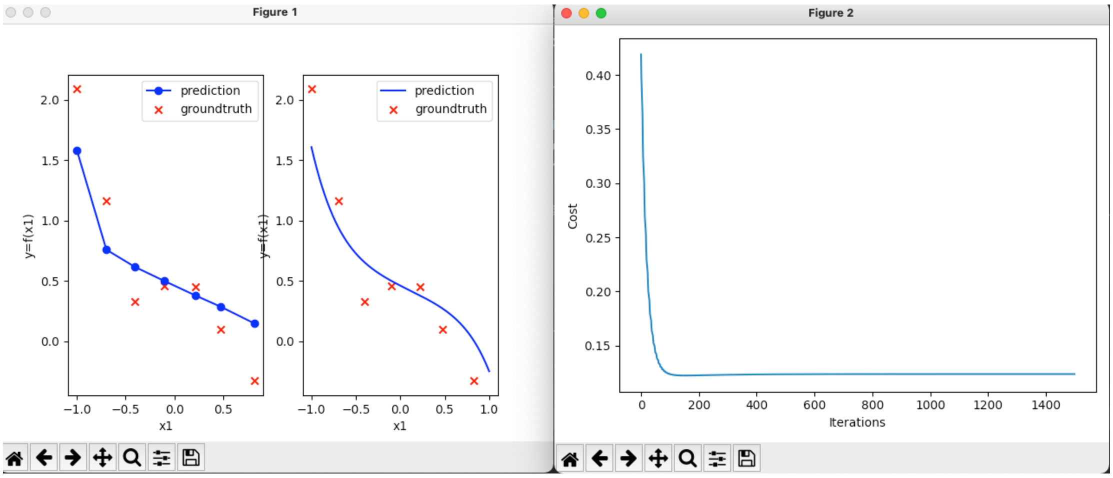
Alpha = 0.01 Minimum cost: 0.06728, on iteration #1413

After many tests, the best alpha is 0.1 and the minimum cost is 0.0249.

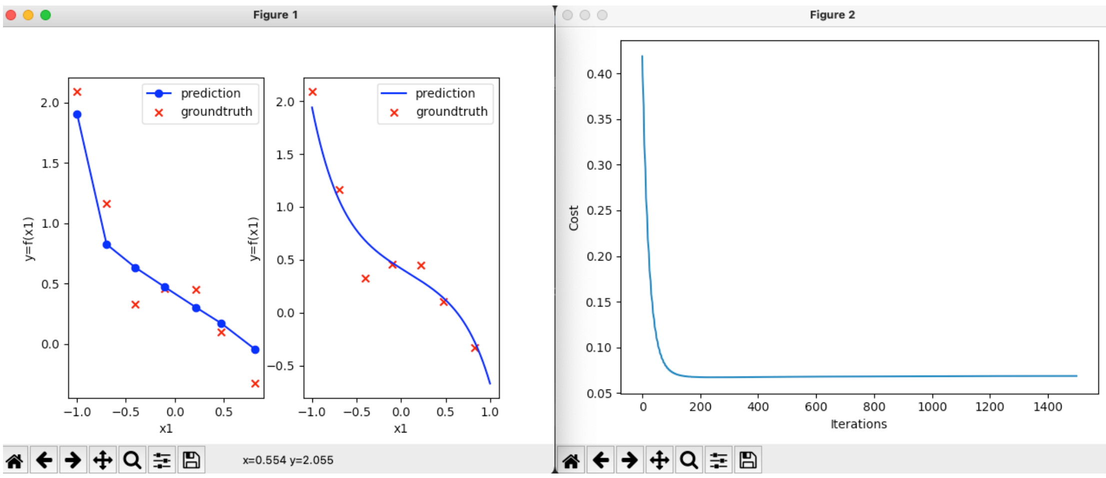
To find the best λ, the alpha = 0.1:



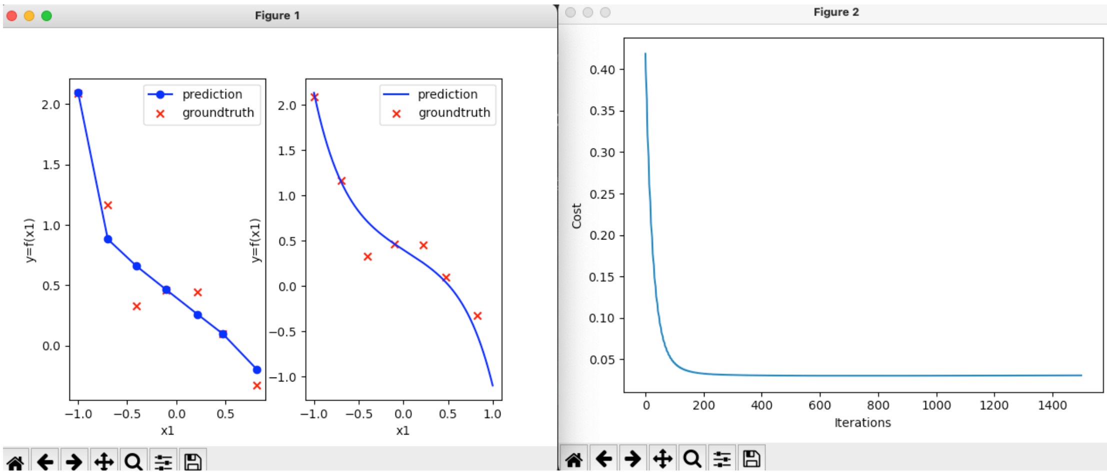
λ=12 Minimum cost: 0.22510, on iteration #37



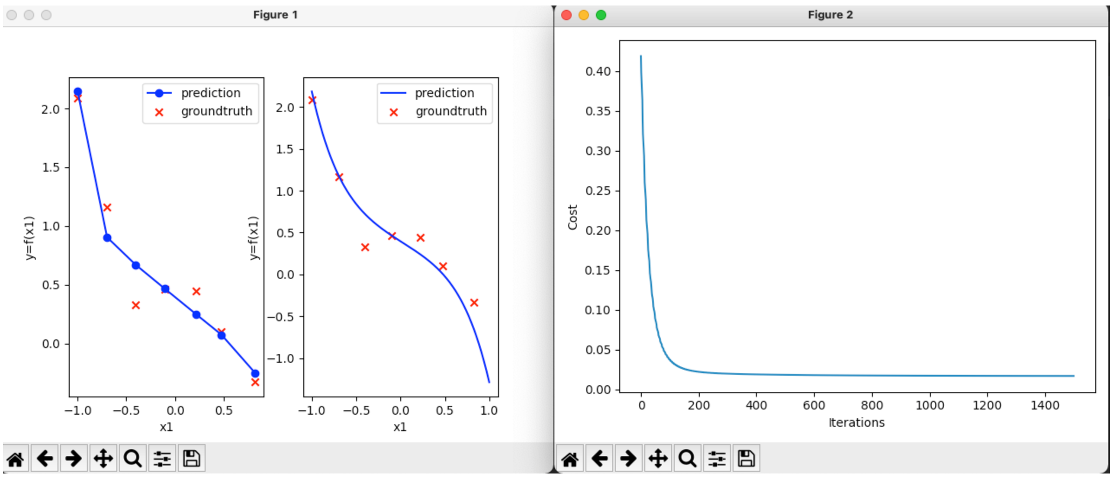
λ=3 Minimum cost: 0.12230, on iteration #153



λ=1 Minimum cost: 0.08542, on iteration #723



λ=0.2 Minimum cost: 0.02988, on iteration #741



λ=0 Minimum cost: 0.01683, on iteration #1499

The lamda is used to correct the over-fitting of the model. When the lamda is too large, the hypothesis prediction result is poor, the convergence is not reached, and under-fitting occurs. When lamda is too small or equal to 0, there is no correction for the effect of overfitting and the hypothesis is greatly affected by the last data point show as above. After experimentation, it is found that the degree of fit is best when lamda = 0.2 and the minimum cost: 0.02988, on iteration #741.