ECS708 Machine Learning Assignment 1: Part 1 - Linear Regression

1. Linear Regression with One Variable :

After putting values of x0 and x1 as 1 and 5 respectively.

```
hypothesis = theta[0]*1 + theta[1]*5
```

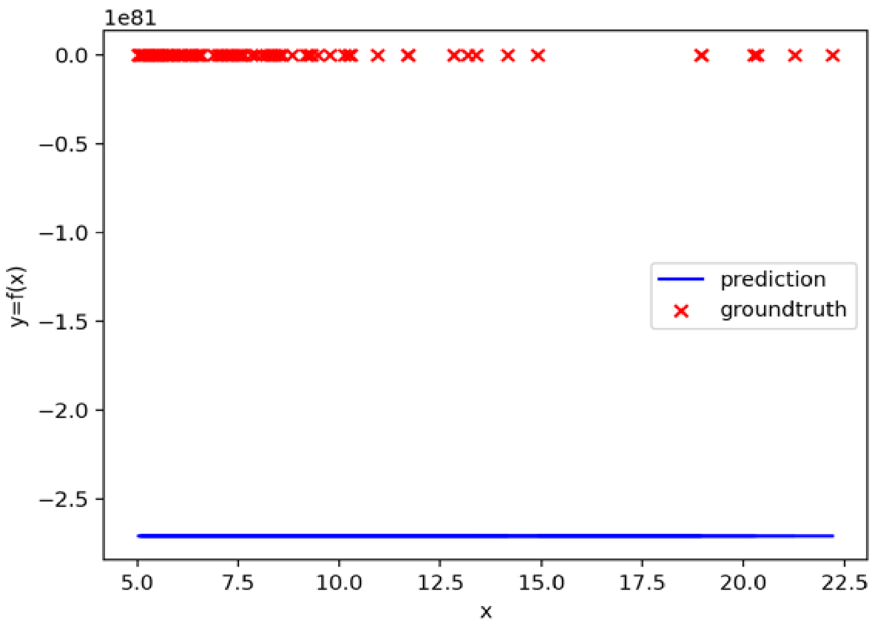


Figure 1.0

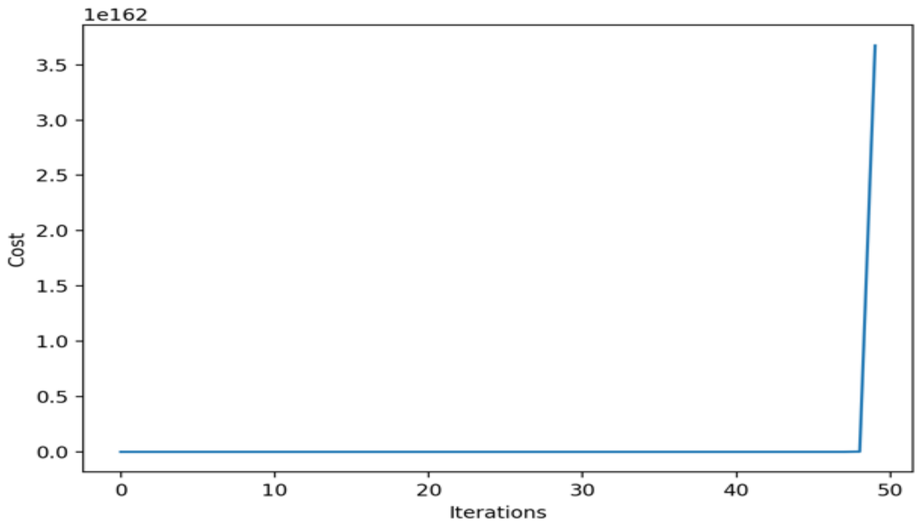


Figure 1.1

Note: In the plots given above we can clearly see that the hypothesis and the cost graph are flat because we haven't calculated our hypothesis.

After changing the calculate_hypothesis.py file we will get the following plots as shown below:

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

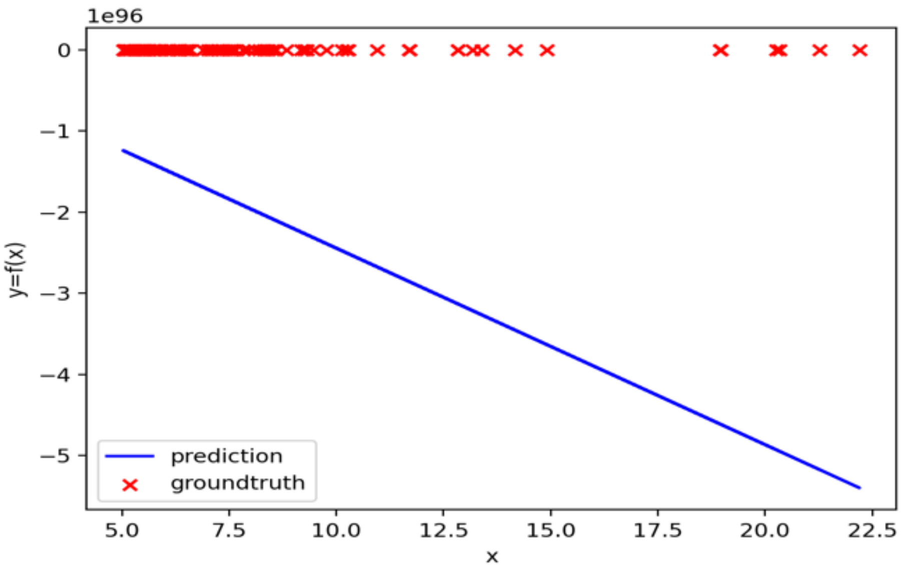


Figure 1.2

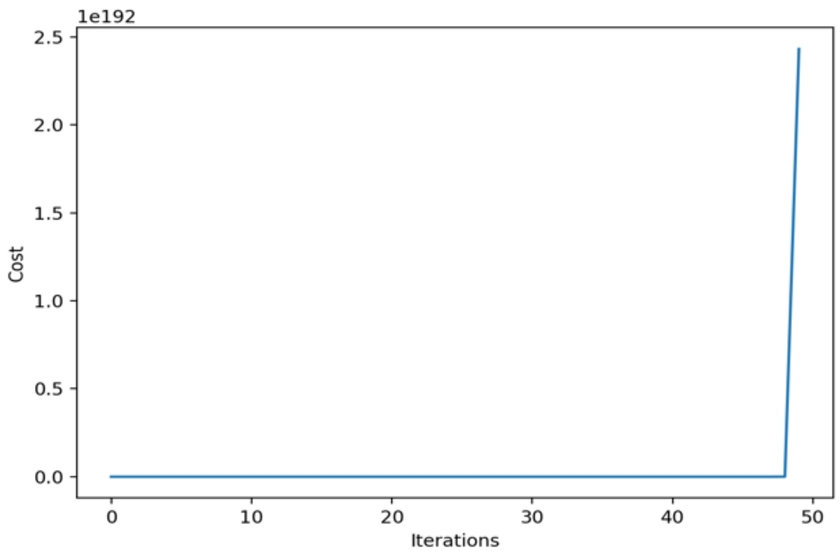


Figure 1.3

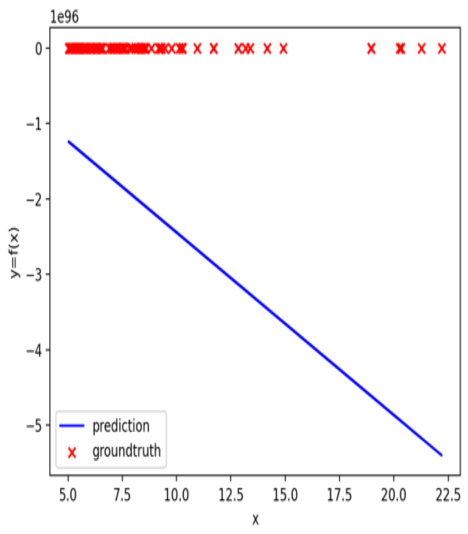
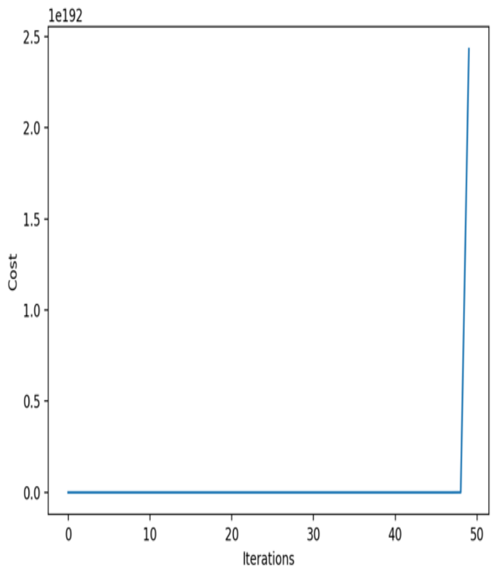
Note: In the plots (Fig 1.2 & Fig 1.3) given above we can see that the gradient of the hypothesis is fitting the model in a better way and cost is going down over time. In this case hypothesis function is not being used in gradient_descent function.

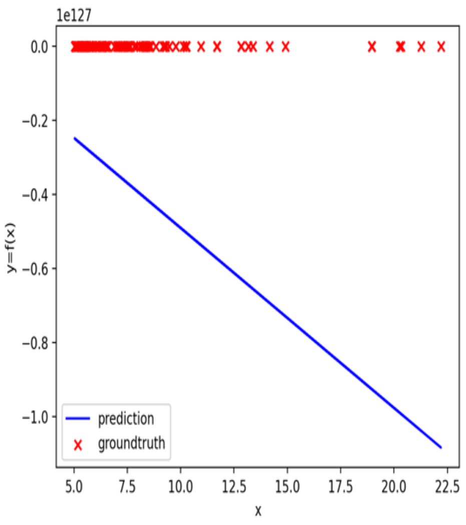
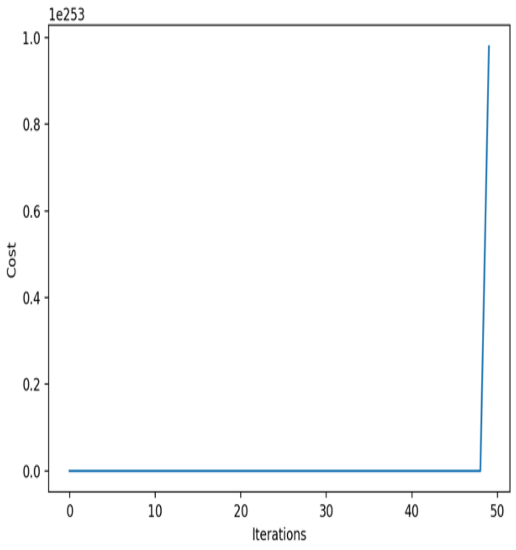
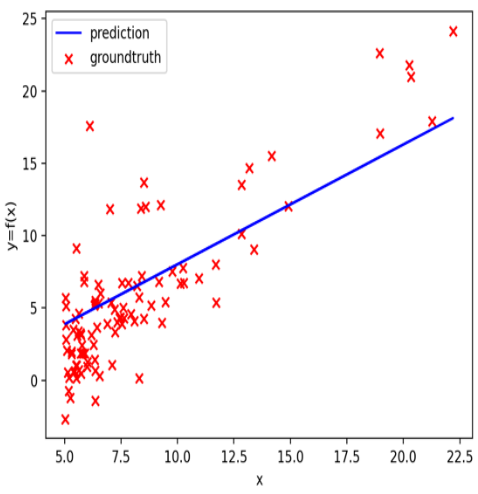
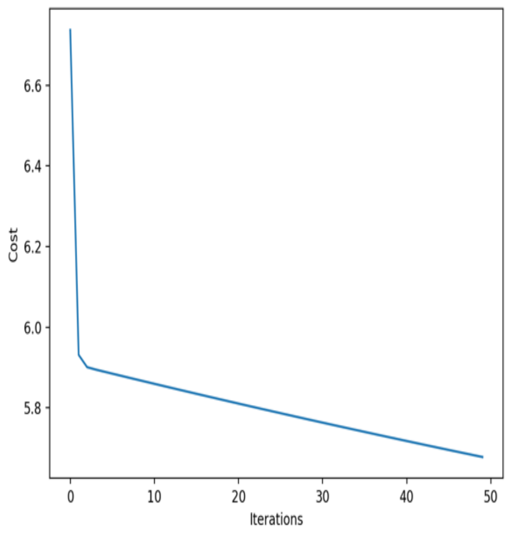
After making the final changes and using the hypothesis function in gradient_descent function we observe the following changes in code below:

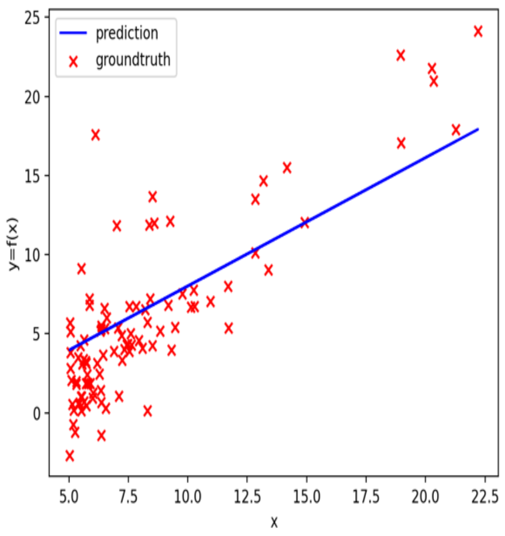
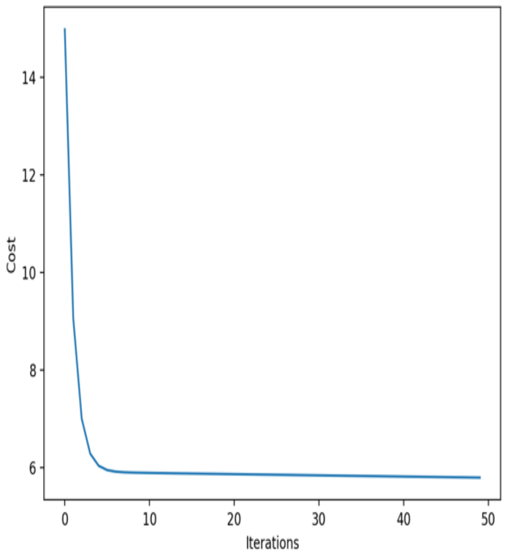
```
hypothesis = calculate_hypothesis(X, theta, i)
```

Note: Changes are done for both the variables for theta_0 and theta_1

Modifying values for learning rate (alpha) in ml_assgn1_1.py :

Learning rate (alpha)	Fig 1 (Predictions)	Fig 2 (Cost)
1.0		

4.0	1e127 0.0 −0.2 −0.4 −0.6 −0.8 −1.0 y=f(x) prediction groundtruth 5.0 7.5 10.0 12.5 15.0 17.5 20.0 22.5 x	1e253 1.0 0.8 0.6 0.4 0.2 0.0 Cost 0 10 20 30 40 50 Iterations
0.01	25 20 15 10 5 0 y=f(x) prediction groundtruth 5.0 7.5 10.0 12.5 15.0 17.5 20.0 22.5 x	6.6 6.4 6.2 6.0 5.8 Cost 0 10 20 30 40 50 Iterations

0.005		

Observation: For higher learning rate we can see that it is not converging and the cost function is not correct. In the case of slow learning rate will allow the model to learn more optimal but it may take longer to train.

2. Linear Regression with Multiple Variables

In this case I am making hypothesis function general so that we can have any number of extra variables.

After modifying both the calculate_hypothesis and gradient_descent files the changes are shown below for the code:

calculate_hypothesis :

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

Note: Here I have calculated the hypothesis for the i-th sample of X, given X, theta and i.

gradient_descent :

Here I am iterating for every theta value

```
for t in range(len(theta)):
```

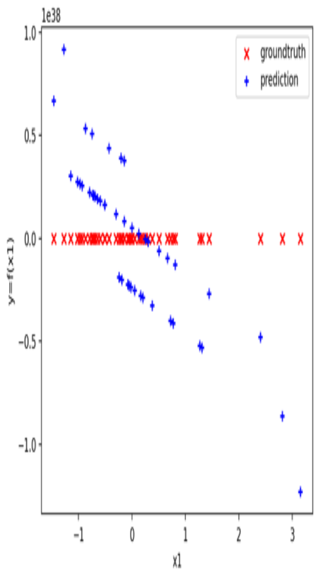
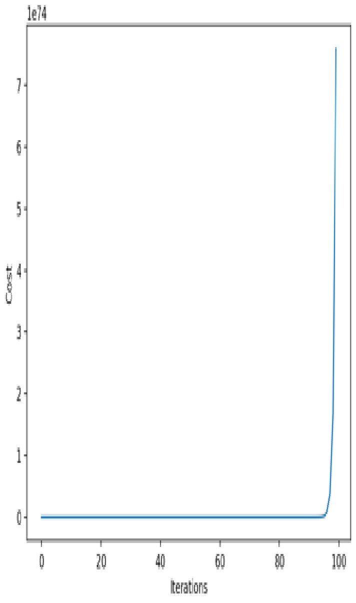
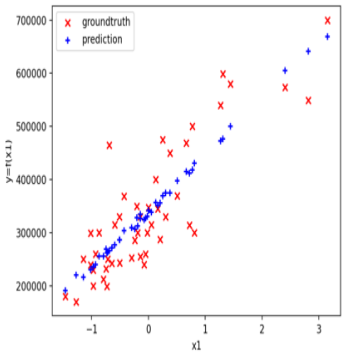
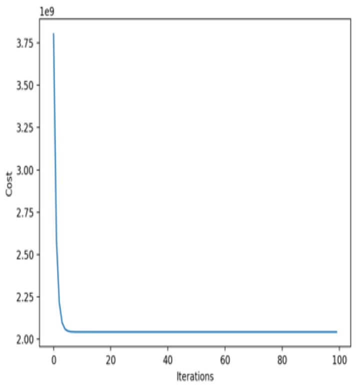
```
hypothesis = calculate_hypothesis(X, theta, i)
```

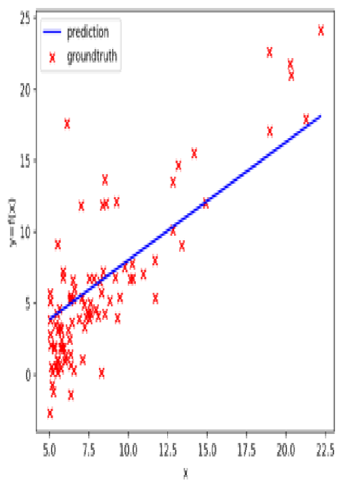
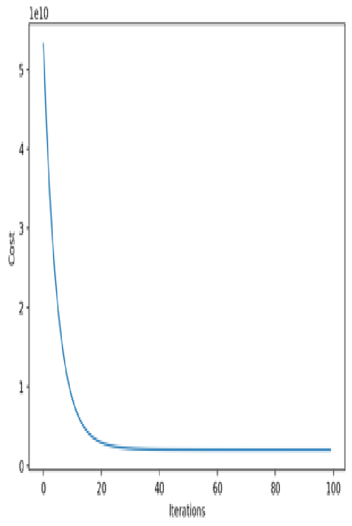
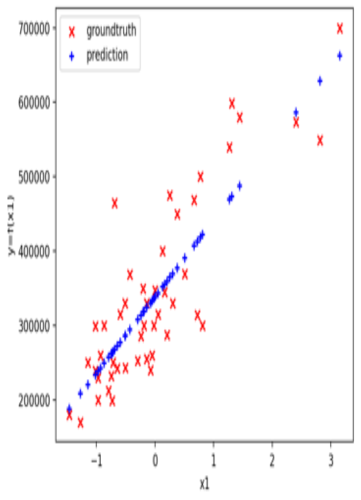
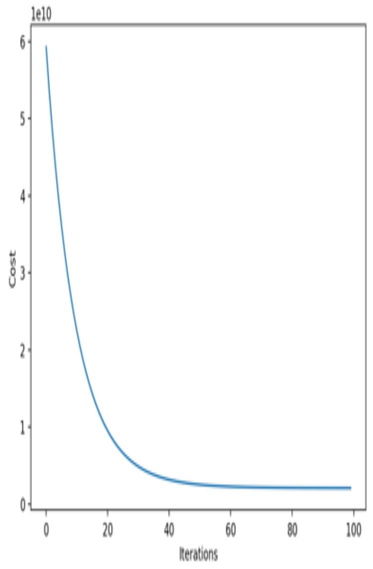
After this step I updated theta_temp using the values of sigma:

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

```
theta_temp[t] = theta_temp[t] - (alpha / m) *
sigma[t]
```

After the following changes and running the ml_assgn1_2.py, it showed how different values of alpha affected the convergence of the algorithm:

Learning Rate (alpha)	**(Predictions)**	**(Cost)**	**theta values**
2			[-1.02882804e+22 - 2.20545795e+37 - 2.20545795e+37]
1			[340412.65957447 109447.79646964 - 6578.35485416]

0.1			[340403.61773803 108803.37852266 - 5933.9413402]
0.005			[3.38397236e+05 1.03161481e+05 - 3.22620198e+02]

Now we will make prediction of house prices and to make a prediction I am normalizing the two variables using the saved values for mean and standard deviation.

The formula used for normalisation is

$$X_{norm} = \frac{X - mean}{standard\ deviation}$$

ml_assgn1_2.py:

```
# Normalize
X_normalized, mean_vec, std_vec =
normalize_features(checkSize)
test_normalised = (X_normalized-
mean_vec)/std_vec
print(test_normalised)

# 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)

hypo_one =
calculate_hypothesis(X_normalized,theta_final,0
)
print (hypo_one)
hypo_two =
calculate_hypothesis(X_normalized,theta_final,1
)
print(hypo_two)
print(theta_final)
```

-The prediction for the following sq. Ft and bedrooms are as following:

1650 sq. ft. and 3 bedrooms = 237543.21795898757

3000 sq. ft. and 4 bedrooms = 443282.1011899486

3. Regularized Linear Regression

In the task 3 it is already mentioned that the punishment for having more terms is not applied to the bias.

In the **compute_cost_regularised.py** file we have compute_cost_regularised function which is shown in the figure below:

```
def compute_cost_regularised(X, y, theta):
```

In the next step I am modifying the gradient_descent to use the compute_cost_regularised method instead of compute_cost.

Its shown in the figure given below:

```
from compute_cost import * {replacing it
with}=>
from compute_cost_regularised import *
```

After replacing compute_cost with compute_cost_regularised, this is the code we are getting shown below for both bias and non-bias term

```
if t == 0:
    iteration_cost = compute_cost(X, y, theta)
else:
    iteration_cost =
compute_cost_regularised(X, y, theta, l)
cost_vector = np.append(cost_vector,
iteration_cost)
```

Note: We don't want to punish the bias term.

```
for t in range(len(theta)):
```

Here we are iterating for every theta values

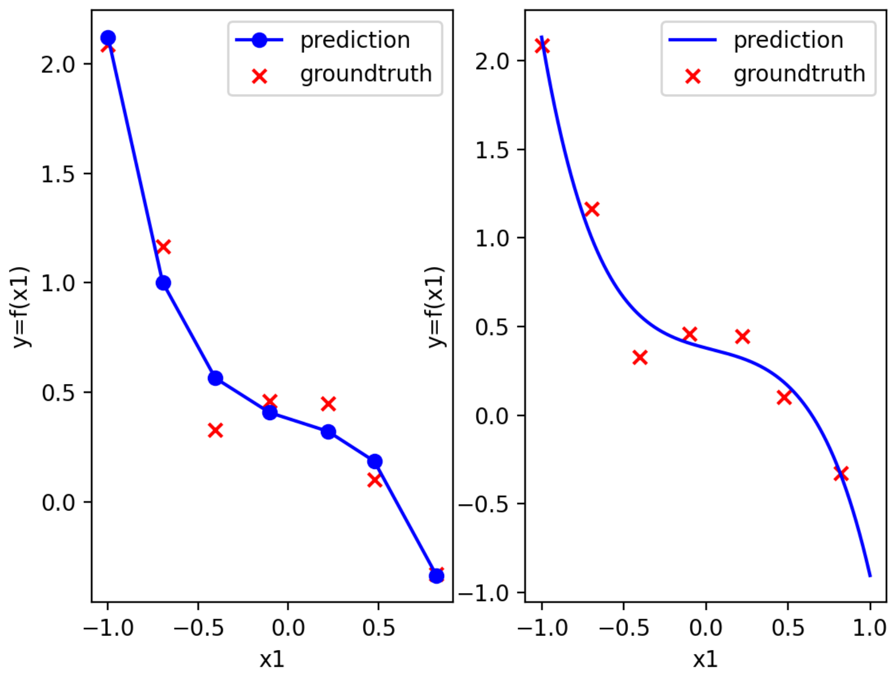
We have calculated the hypothesis for the i-th sample of X

```
for i in range(m):

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

Now updating the theta_temp, using the values of sigma for both bias and non-bias terms

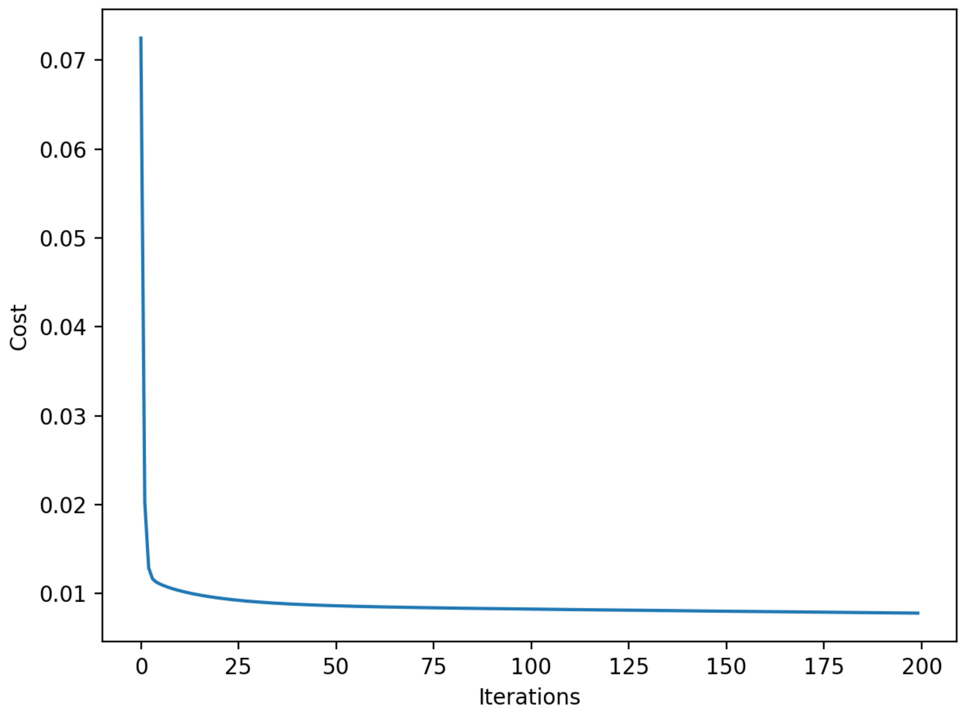
```
if t == 0:
    theta_temp[t] = theta_temp[t] - (alpha / m) * sigma[t]
else:
    theta_temp[t] = theta_temp[t] * (1 - alpha * (l / m)) - (alpha / m) * sigma[t]
```

After updating the gradient function and running the ml_assgn1_3.py we get the following plots for the cost function with the default data given:



prediction
groundtruth
y=f(x1)
x1
prediction
groundtruth
y=f(x1)
x1

Fig 3.1

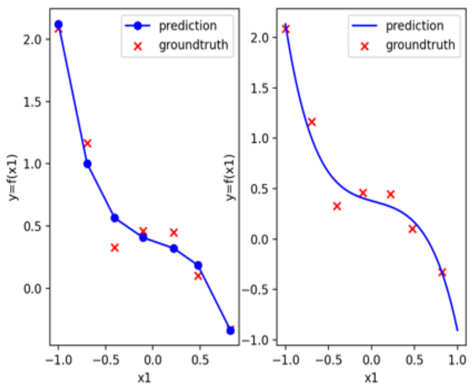
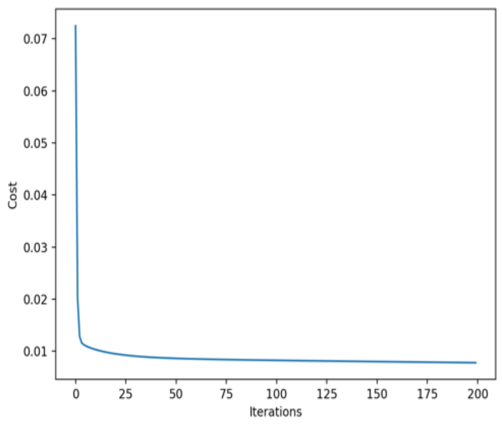


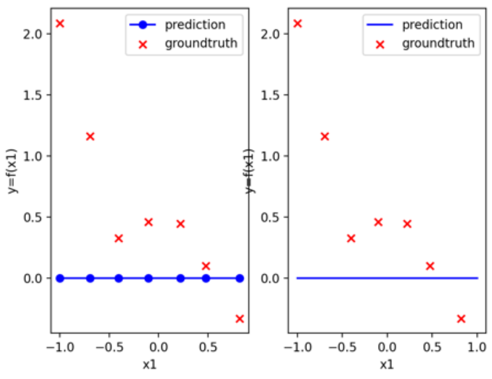
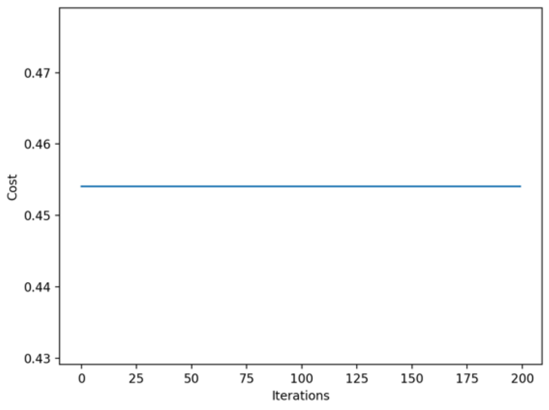
Cost
Iterations

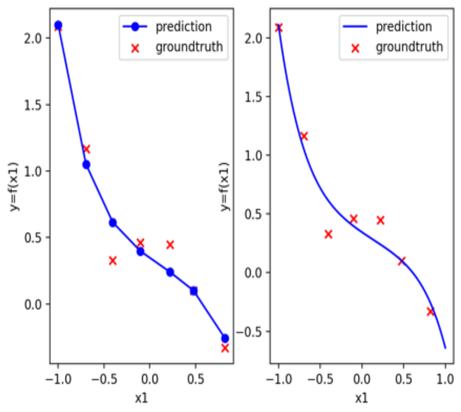
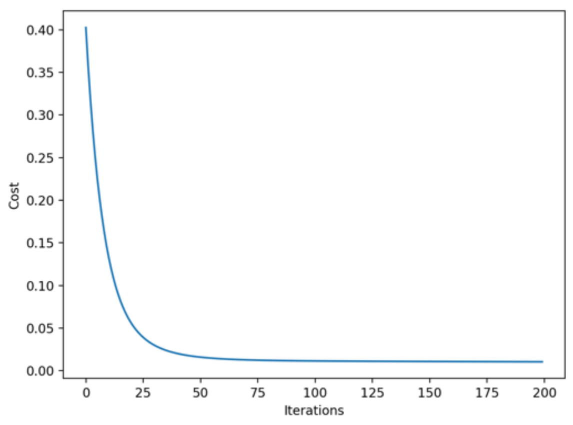
Fig 3.2

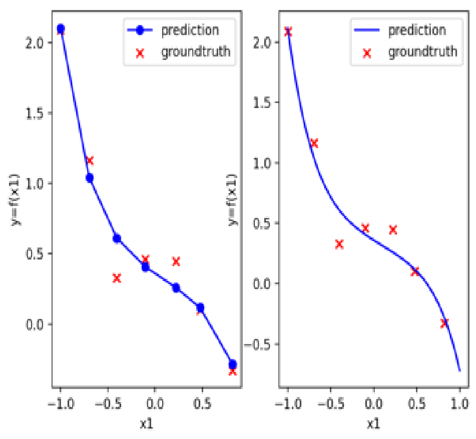
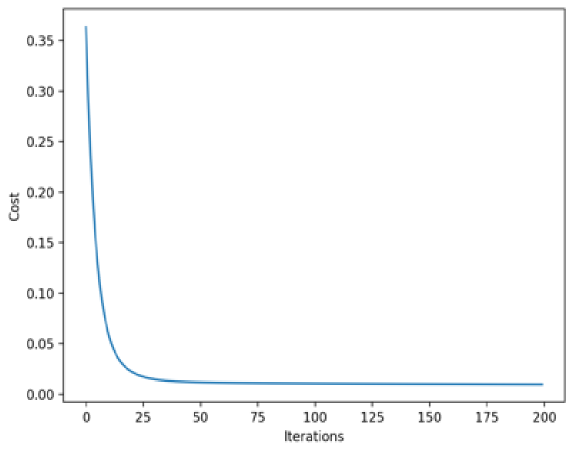
Here the figures shown below are for different alpha values :

λ and alpha values	Prediction	Cost Function

Alpha = 1
 	
λ = 0
		

Alpha = 0
 	
λ = 0
		

Alpha = 0.05		

Alpha = 0.09
 	
λ = 0
		

Note : I am modifying gradient_descent file to take an extra parameter *l* (which represents λ, used for regularization)

We are modifying the gradient_descent function to accept an additional argument lambda (l)

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

current iteration’s cost to cost_vector

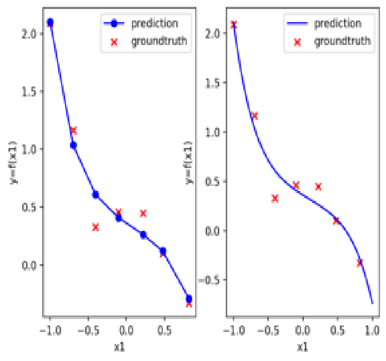
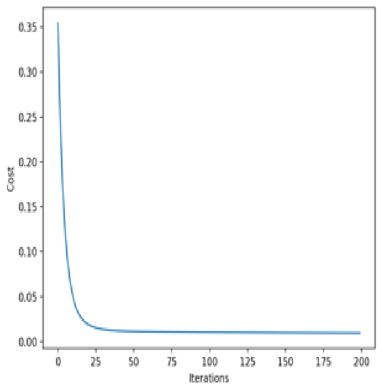
```
iteration_cost = compute_cost_regularised(X, y,
theta, l)

cost_vector = np.append(cost_vector, iteration_cost)
```

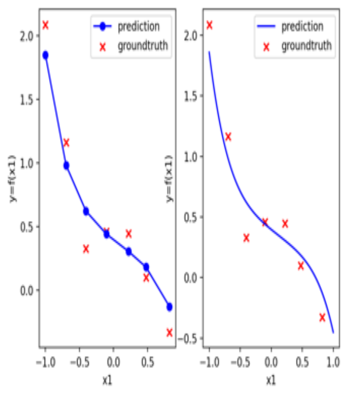
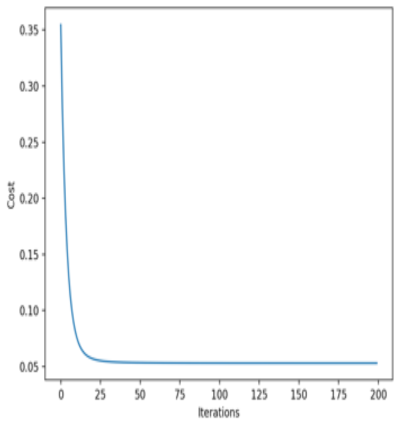
we also did minor change in ml_assgn1_3.py to pass an additional argument lambda (l)

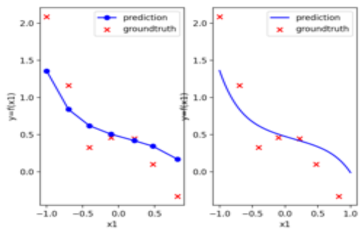
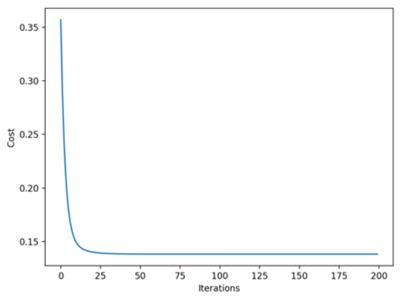
```
theta_final = gradient_descent(X, y, theta, alpha,
iterations, do_plot, l)
```

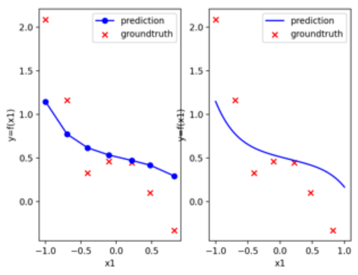
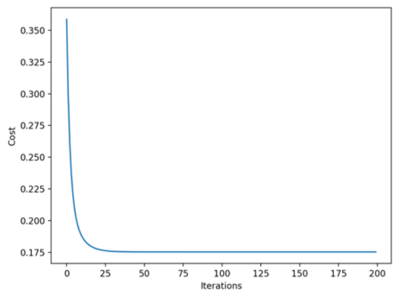
Here the figures shown below are for different lambda and alpha values :

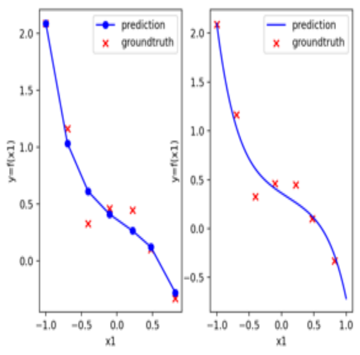
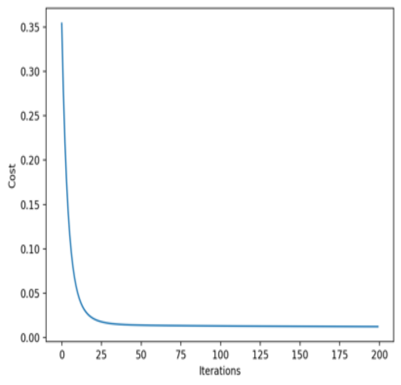
λ and alpha values	Prediction	Cost Function

Alpha = 0.01
λ = 0
		

Alpha = 0.01
λ = 1
		

Alpha = 0.01
λ = 5
		

Alpha = 0.01
 	
λ = 9
		

Alpha = 0.01
 	
λ = 0.05
		

Note: From the following figures we can observe that if the value of lambda is higher then it is underfitting and if the value of lambda is lower then its overfitting. For lambda value 0 it is optimised.