Linear Regression Report

[**Student Information**](#_qewguqj585he) **1**

[**Introduction**](#_e4nniutc9lu7) **1**

[**Data Generation**](#_ibw5grtks0n8) **1**

[**The Normal Equations Method**](#_8q0a5w8fjhbj) **2**

[**Gradient Descent**](#_7ffcn2fkxzte) **3**

[Learning Rate Too Small](#_yyjnlc5xipu5) 4

[Learning Rate Too Large](#_qyv6q0qu6erm) 6

[Learning Rate Just Right](#_y8l3cy1089ve) 7

[**Code For Generating the Contour Plot**](#_k5l1i5ci61us) **8**

[**Comparison Of Normal Equations and Gradient Descent**](#_v40f2qje3ahl) **9**

[**Conclusion**](#_c6m05kl2cns5) **9**

# Student Information

Name: Mahmoud Hassan Mohamed Daif

ID: 18R8107

Mail. mahmoud.daif.8h@stu.hosei.ac.jp

# Introduction

The purpose of this assignment is to implement gradient descent and the normal equations for fitting a linear regression line and compare the two methods and explore the effect of hyper parameter tuning on linear regression.

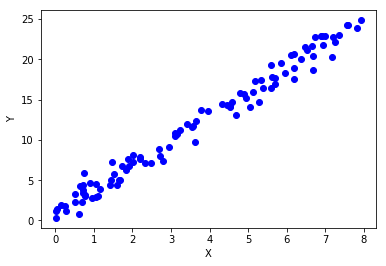
# Data Generation

The goal is to generate a data that follows the equation Y = 1 + 3X.

With the values of X ranging from 0 to 8. We generate a hundred examples then add white Gaussian noise to that so that the relation between X and Y won’t be perfect. Below is the code to do so.

|  |
| --- |
| num\_of\_points = 100  num\_dim = 1  X = 8\*np.random.rand(num\_of\_points,num\_dim)  X = np.c\_[np.ones((num\_of\_points,num\_dim)),X]  print(X.shape)  thetas\_best = np.array([1,3])  print(thetas\_best.shape)  Y = X.dot(thetas\_best)  Y = Y.reshape(num\_of\_points,num\_dim)  Y = Y +np.random.randn(num\_of\_points,num\_dim)  print(y.shape) |

The shape of the Data will be as the figure below



The code used to plot the figure is shown Below.

|  |
| --- |
| plt.plot(X[:,1],Y,'bo') plt.xlabel('X') plt.ylabel("Y") |

# The Normal Equations Method

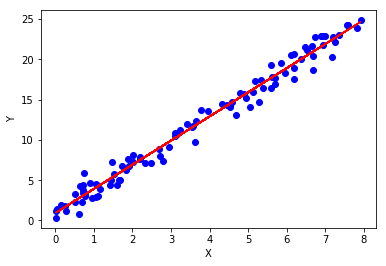
First we try the normal equation method. My implementation of the normal equations method is shown below:

|  |
| --- |
| #Input: #x: features #y: observed values #Output: #theta: linear regression coefficients  def normal\_eguation\_method(x,y):  theta = np.linalg.inv(x.T.dot(x)).dot(x.T).dot(y)  return theta |

|  |
| --- |
| theta = normal\_eguation\_method(X,Y) |

The best values we get for theta 1 and theta 2 are 0.93073497 and 2.99542427 respectively.

Next, those values of theta1 and theta2 are used to fit a linear regression line as shown in the figure below:



The figure is plotted using the code below

|  |
| --- |
| plt.plot(X[:,1], Y, 'bo') plt.plot(X[:,1], X.dot(theta), '-',color='red') plt.xlabel('X') plt.ylabel("Y") |

# Gradient Descent

Second gradient descent is tested against the data. My implementation of gradient descent is as shown below:

|  |
| --- |
| #Objective: the function computes the cost function for linear regression  #using mean square error #Input: #Theta: the weight coefficents for linear regression #x: the feature values for linear regression #y: the real value of predicted output #Output: #The root mean square error value def cost\_function(theta, x, y):  y\_hat = x.dot(theta)  m = x.size  mean\_error = (1/2\*m)\*np.sum(np.square((y\_hat-y)))  return mean\_error |

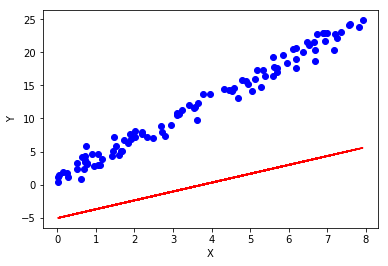
|  |
| --- |
| #Input: #Theta: the weight coefficents for linear regression  #X: the feature values for linear regression #learning\_rate: the learning rate for gradient descent def gradient\_descent(theta, x, y, learning\_rate = 0.01, iterations = 1000):  m = x.size  thetas = np.zeros((iterations, theta.shape[0]))  costs = np.zeros(iterations)  for i in range(iterations):  y\_hat = x.dot(theta)  theta = theta - (learning\_rate/m)\*x.T.dot((y\_hat-y))  thetas[i,:] =theta.T  costs[i] = cost\_function(theta, x, y)  return theta, thetas, costs |

Now we try to reproduce the 3 scenarios with gradient descent

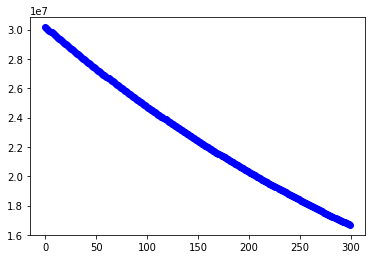
## Learning Rate Too Small

Learning rate = 0.0001

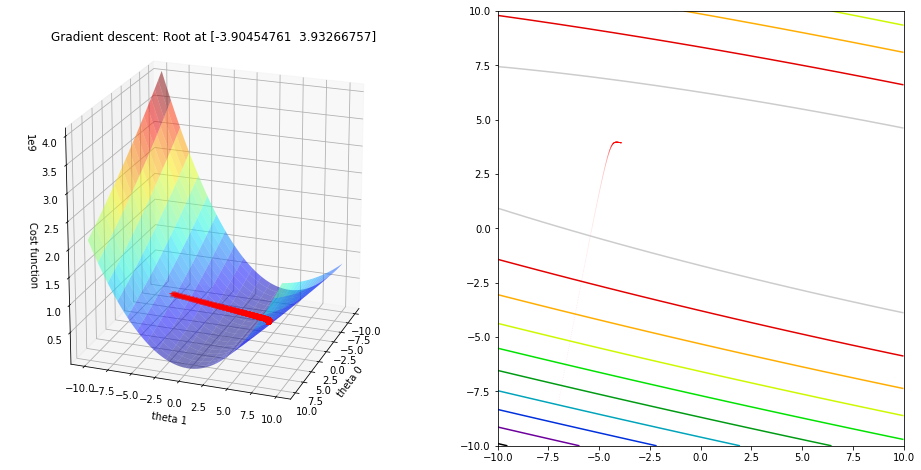
We get the values of theta1 and theta2 -5.06388634 and 1.34061087 respectively after 1500 iterations. Now, let’s try to examine the line produced by such values.



Not so much of a good fit. We can confirm this by examining the change in the cost function



It still doesn’t converge. When taking a look at the contour plot we see that the problem is that gradient descent descends to the local minimum too slowly

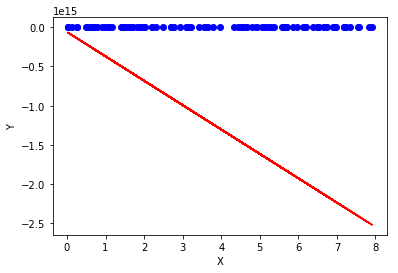


## Learning Rate Too Large

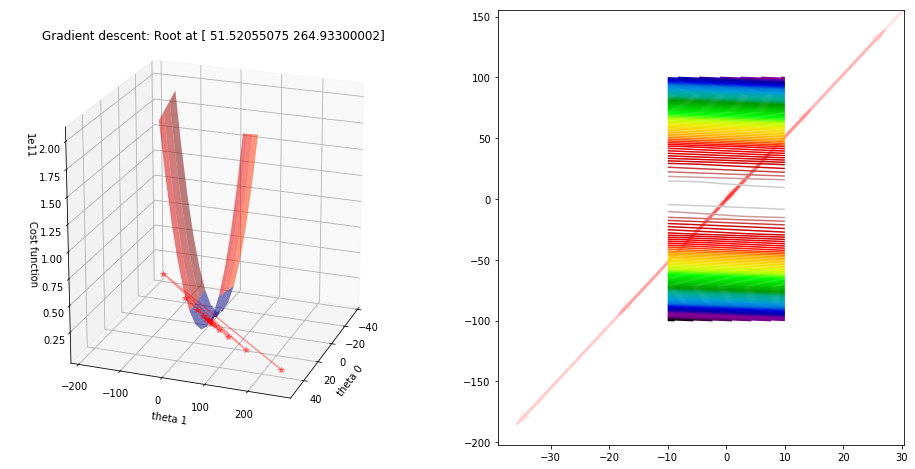
Learning rate = 0.24

We get the following values of theta1 and theta2, -6.02941510e+13 and -3.10870755e+14 respectively.

We try to fit a line using those values as shown in the figure below it’s a very bad fit.



Now we want to examine the reason for that by looking into the contour plot below:

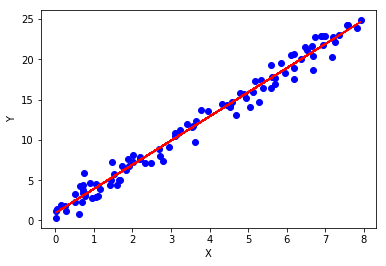


It’s clear that the learning rate is too high and gradient descent keeps overshooting unable to reach a local minmum.

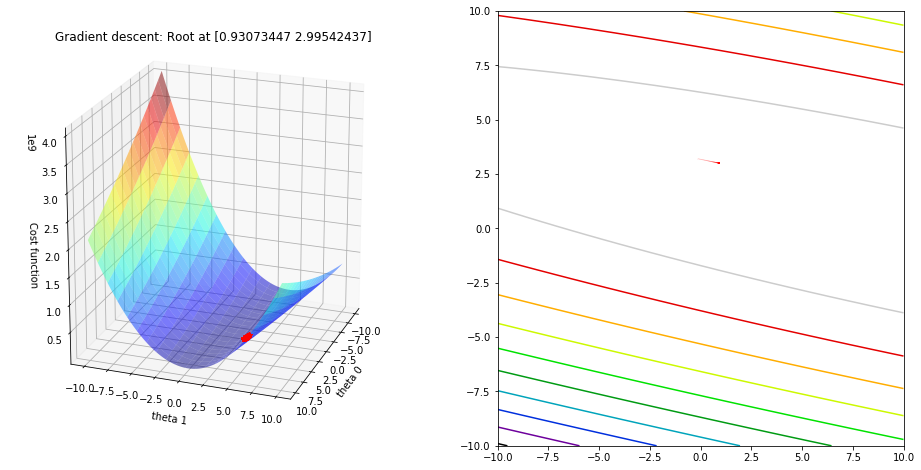
## Learning Rate Just Right

With a learning rate of 0.1 gradient descent reaches the exact value of theta1 and theta2 optained by the normal equations. The values of theta1 and theta2 are 0.93073497 and 2.99542427 respectively.

Then we fit a line using those values:



Next thing is, we examine the contour plot:



We can see that gradient descent doesn’t take to much effort to reach a local minimum.

# Code For Generating the Contour Plot

|  |
| --- |
| theta\_initial = np.array([-7, -8]).reshape(2,1) theta,thetas,costs = gradient\_descent(theta\_initial, X,Y, 0.001,1000) theta\_0 = thetas[:,0] theta\_1 = thetas[:,1] J\_history = costs fig = plt.figure(figsize = (16,8))  #Surface plot ax = fig.add\_subplot(1, 2, 1, projection='3d') ax.plot\_surface(T0, T1, Z, rstride = 5, cstride = 5, cmap = 'jet', alpha=0.5) ax.plot(theta\_0,theta\_1,J\_history, marker = '\*', color = 'r', alpha = .4, label = 'Gradient descent')  ax.set\_xlabel('theta 0') ax.set\_ylabel('theta 1') ax.set\_zlabel('Cost function') ax.set\_title('Gradient descent: Root at {}'.format(theta.ravel())) ax.view\_init(20,20)   #Contour plot ax = fig.add\_subplot(1, 2, 2) ax.contour(T0, T1, Z, 10, cmap = 'nipy\_spectral\_r', origin = 'lower') ax.quiver(theta\_0[:-1], theta\_1[:-1], anglesx, anglesy, scale\_units = 'xy', angles = 'xy', scale = 1, color = 'r', alpha = .1)  plt.show() |

# Comparison Of Normal Equations and Gradient Descent

In terms of speed in our example the normal equations method takes 0.71 second to excute while the gradient descent method takes 0.03 seconds to excute. In the normal equations method we don’t need to choose the learning rate or define the number of iterations nor keep a track of the lurning curve, however computationally it’s much more expensive. When having millions of features and thosands or hundred thousands of training examples. The normal equation method will be computationally very expensive or not feasible.

# Conclusion

It’s better to use the normal equation method when we have relatively few training examples and few features. However as the number of features and training examples increases using gradiet descent becomes more feasible!