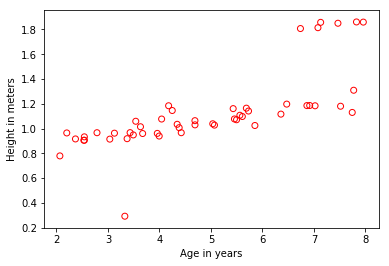
**Programming Assignment 1**

**Part A**

Raw data

****

Converged value of theta: [[0.49421836] [0.12800497]]

Plot of fit line

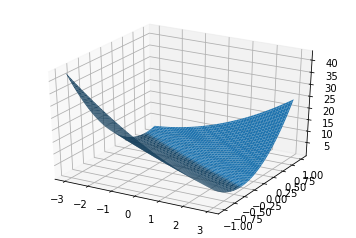


Prediction:

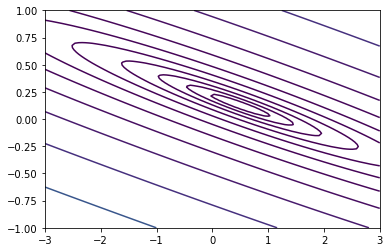
For age 3.5, we predict a height of 0.942236

For age 7, we predict a height of 1.390253

Plot of 3D surface of J:



Plot of contour of J:



What is the relationship between these plots (3D surface and contour) and the value of θ0 and θ1 that your implementation of gradient descent had found?

J\_theta approaches a minimum at theta\_0 and theta\_1

Code:

1. # -\*- coding: utf-8 -\*-
2. """
3. Spyder Editor
5. This is a temporary script file.
6. """
8. **import** numpy as np
9. **from** mpl\_toolkits.mplot3d **import** Axes3D
10. **import** matplotlib.pyplot as plt
12. x = np.loadtxt('ax.dat')
13. y = np.loadtxt('ay.dat')
15. #Plot original data out
16. plt.scatter(x, y, facecolors='none', color='red')
17. plt.xlabel('Age in years')
18. plt.ylabel('Height in meters')
19. plt.show()
21. #Get the number of examples
22. m = x.shape[0]
23. #Reshape x to be a 2D column vector
24. x.shape = (m,1)
25. #Add a column of ones to x
26. X = np.hstack([np.ones((m,1)), x])

29. #initialize theta
30. theta = np.zeros(shape=(2,1))

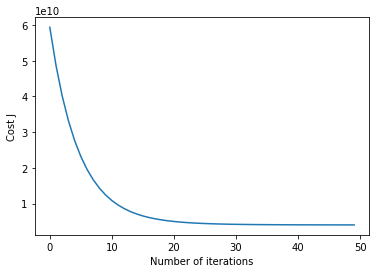
33. #gradient descent
34. alpha = .07
35. loops = 1300
37. **def** computeCost(X, y, theta):
38. m = y.size
39. estimates = X.dot(theta).flatten()
40. squaredErrors = (estimates - y) \*\* 2
41. J = (1.0 / (2 \* m)) \* squaredErrors.sum()
42. **return** J
44. **def** gradientDescent(X, y, theta, alpha, iterations):
45. m=y.size
46. J\_history = np.zeros(shape = (iterations, 1))
48. **for** i **in** range(iterations):
49. estimates = X.dot(theta).flatten()
50. err\_x1 = (estimates - y) \* X[:, 0]
51. err\_x2 = (estimates - y) \* X[:, 1]
53. theta[0][0] = theta[0][0] - alpha \* (1.0/m) \* err\_x1.sum()
54. theta[1][0] = theta[1][0] - alpha \* (1.0/m) \* err\_x2.sum()
56. J\_history[i,0] = computeCost(X, y, theta)
58. **return** theta, J\_history

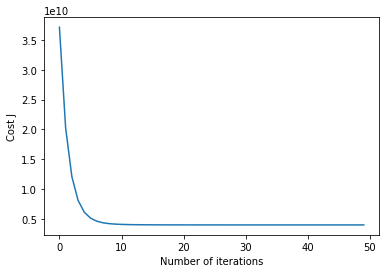
61. theta, J\_history = gradientDescent(X, y, theta, alpha, 1300)
63. **print** theta
65. predict1 = np.array([1, 3.5]).dot(theta).flatten()
66. **print** 'For age 3.5, we predict a height of %f' % (predict1)
67. predict2 = np.array([1, 7]).dot(theta).flatten()
68. **print** 'For age 7, we predict a height of %f' % (predict2)

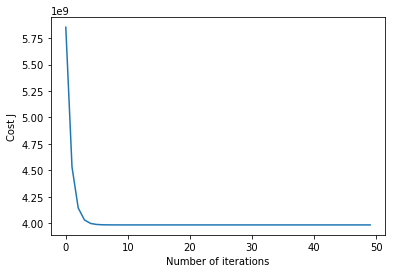
71. plt.plot(X[:,1],np.dot(X,theta))
72. plt.legend(['Linear Regression', 'Training Data'])
73. plt.scatter(x, y, facecolors='none', color='red')
74. plt.xlabel('Age in years')
75. plt.ylabel('Height in meters')
76. plt.show()


80. #Display Surface Plot of J
81. t0 = np.linspace(-3,3,100)
82. t1 = np.linspace(-1,1,100)
83. t0.shape = (len(t0),1)
84. t1.shape = (len(t1),1)
85. T0, T1 = np.meshgrid(t0,t1)
86. J\_vals = np.zeros((len(t0),len(t1)))
87. **for** i **in** range(len(t0)):
88. **for** j **in** range(len(t1)):
89. t = np.hstack([t0[i], t1[j]])
90. J\_vals[i,j] = computeCost(X, y, t)
92. #Because of the way meshgrids work with plotting surfaces
93. #we need to transpose J to show it correctly
94. J\_vals = J\_vals.T
95. fig = plt.figure()
96. ax = fig.gca(projection='3d')
97. ax.plot\_surface(T0,T1,J\_vals)
98. plt.show()
99. plt.close()
100. #Display Contour Plot of J
101. 5
102. plt.contour(T0,T1,J\_vals, np.logspace(-2,2,15))
103. plt.show()

**Part B**

Learning rate of .1

Learning rate of .3

Learning rate of 1.0

Converged value of theta for 1.0 learning rate: [[353178.61702128] [114973.94461206] [-3702.28591238]]

Prediction of price of house with 3 bedrooms that is 1650 sqft: 303613.809912

Code:

1. #!/usr/bin/env python2
2. # -\*- coding: utf-8 -\*-
3. """
4. Created on Mon Sep 11 20:23:00 2017
6. @author: rditljtd
7. """
9. **import** numpy as np
10. **from** mpl\_toolkits.mplot3d **import** Axes3D
11. **import** matplotlib.pyplot as plt
13. x = np.loadtxt('bx.dat')
14. y = np.loadtxt('by.dat')

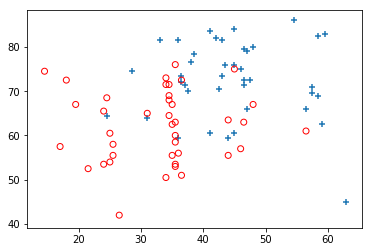
17. test = [1650, 3]
19. #Get the number of examples
20. m = x.shape[0]
22. #Reshape x to be a 2D column vector
23. x.shape = (m,2)
25. sigma = np.std(x,axis=0) #std
26. mu = np.mean(x,axis=0) #mean
27. x = (x-mu) / sigma #adjustment
29. test = (test-mu) / sigma
31. #Add a column of ones to x
32. X = np.hstack([np.ones((m,1)), x])
34. #initialize theta
35. theta = np.zeros(shape=(3,1)) #Initialize theta
36. #print theta
37. alpha = 1.#Your learning rate#
38. #J = []
39. iterations = 50
40. J\_history = np.zeros(shape = (iterations, 1))
42. #Closed-form solution
44. theta2 = np.linalg.inv(X.transpose().dot(X)).dot(X.transpose().dot(y))
45. **print** theta2
47. predict2 = np.array([1, test[0], test[1]]).dot(theta2).flatten()
48. **print** 'For a house with 3 bedrooms and 1650 sqft, we predict a price of %f' % (predict2)

51. #gradient descent solution
53. **def** computeCost(X, y, theta):
54. m = y.size
55. estimates = X.dot(theta).flatten()
56. squaredErrors = (estimates - y) \*\* 2
57. J = (1.0 / (2 \* m)) \* squaredErrors.sum()
58. **return** J
60. **def** gradientDescent(X, y, theta, alpha, iterations):
61. m=y.size
63. estimates = X.dot(theta).flatten()
64. err\_x1 = (estimates - y) \* X[:, 0]
65. err\_x2 = (estimates - y) \* X[:, 1]
66. err\_x3 = (estimates - y) \* X[:, 2]
67. #print "err\_x1: " + `err\_x1` + " err\_x2: " + `err\_x2` + " err\_x3: " + `err\_x3`
69. theta[0][0] = theta[0][0] - alpha \* (1.0/m) \* err\_x1.sum()
70. theta[1][0] = theta[1][0] - alpha \* (1.0/m) \* err\_x2.sum()
71. theta[2][0] = theta[2][0] - alpha \* (1.0/m) \* err\_x3.sum()


75. **return** theta
77. **for** i **in** range(iterations):
78. theta = gradientDescent(X, y, theta, alpha, iterations)#might need to take i out
79. J\_history[i] = computeCost(X, y, theta)
81. #Now plot J
82. plt.plot(range(iterations), J\_history)
83. plt.xlabel('Number of iterations')
84. plt.ylabel('Cost J')
86. **print** theta
88. predict1 = np.array([1, test[0], test[1]]).dot(theta).flatten()
89. **print** 'For a house with 3 bedrooms and 1650 sqft, we predict a price of %f' % (predict1)

**Part C**

Plot of raw data:



I spent a good 4 hours on Part C last night and could not figure out how Newton’s method worked. I could not get the math programmed correctly. I tried tweaking it for hours, and nothing I did seemed to work.

Here is the code I had when I went to sleep last night.

Code:

1. #!/usr/bin/env python2
2. # -\*- coding: utf-8 -\*-
3. """
4. Created on Mon Sep 11 20:23:00 2017
6. @author: rditljtd
7. """
9. **import** numpy as np
10. **from** mpl\_toolkits.mplot3d **import** Axes3D
11. **import** matplotlib.pyplot as plt
12. **from** scipy.special **import** expit
14. x = np.loadtxt('cx.dat')
15. y = np.loadtxt('cy.dat')

18. #Get the number of examples
19. m = x.shape[0]
21. #Reshape x to be a 2D column vector
22. x.shape = (m,2)
24. #Add a column of ones to x
25. X = np.hstack([np.ones((m,1)), x])
27. #initialize theta
28. theta = np.zeros(shape=(3,1)) #Initialize theta
29. #print theta
30. #alpha = .3#Your learning rate#
31. #J = []
32. iterations = 15
33. J\_history = np.zeros(shape = (iterations, 1))
35. pos = np.nonzero(y)
36. neg = np.where(y==0)[0]
38. plt.scatter(x[pos,0],x[pos,1],marker='+')
39. plt.scatter(x[neg,0],x[neg,1],facecolors='none',marker='o', color='r')
40. plt.show()

43. #Newtons method solution
45. **def** sigmoid(z):
46. **print** "z: " + `z`
47. toreturn = expit(z)
48. **print** "sigmoid: " + `toreturn`
49. **return** toreturn
51. **def** hypothesis(X, theta):
52. **print** "here: " + `theta`
53. **print** "there: " + `X`
54. toreturn = (sigmoid(np.transpose(theta))\*(X))
55. **print** "hypothesis: " + `toreturn`
56. **return** toreturn

59. **def** computeCost(X, y, theta):
60. m = y.size
61. toreturn = ((1.0/m) \* (-y.dot(np.log(hypothesis(X, theta))) - (1-y).dot(np.log(1-hypothesis(X, theta)))).sum())
62. **print** "computeCost: " + `toreturn`
63. **return** toreturn
65. **def** gradientDescent(X, y, theta):
66. m = y.size
67. minus = hypothesis(X, theta).subtract(hypothesis(X,theta), y)
68. toreturn = (1.0/m)\*(minus.dot(X))
69. **print** "gradientDescent: " + `toreturn`
70. **return** toreturn
72. **def** hessian(X, y, theta):
73. m = y.size
74. minus = 1-hypothesis(X,theta)\*(hypothesis(X, theta))
75. xTrans = (X.transpose()).dot(X)
76. result = minus.dot(xTrans)
77. **print** np.sum(result)
78. toreturn = (1.0/m)\*((minus).dot(xTrans)).sum()
79. **print** "hessian: " + `toreturn`
80. **return** toreturn
82. **def** newtonsMethod(X, y, theta, iterations):
83. m=y.size
85. **for** i **in** range(iterations):
86. J\_history[i] = computeCost(X, y, theta)
87. theta = (theta - (1/(hessian(X, y, theta))) \* gradientDescent(X, y, theta))
89. **print** theta, J\_history
90. **return** theta, J\_history
92. theta, J\_history = newtonsMethod(X, y, theta, 15)

**Corrected Rework**

Part B theta on prediction – theta = [[ 353178.61702128]
[ 114973.94461206]
[ -3702.28591238]]

Part B predicted value = [ 303613.80991151]

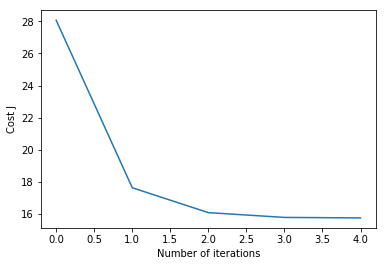
Part C was very much incomplete.

Since then, I have fixed the math logic in the code.

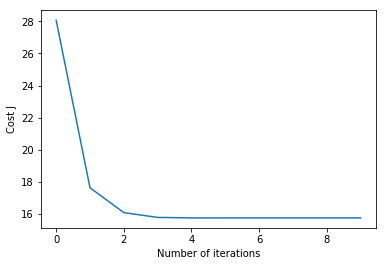
My values for theta are as follows

theta = [[-15.76822471]
[ 0.14087783]
[ 0.15219571]]

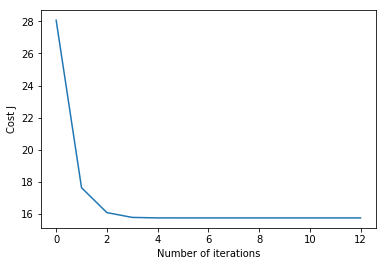
With 5 iterations, it has begun to converge



With 10 iterations it converges more.

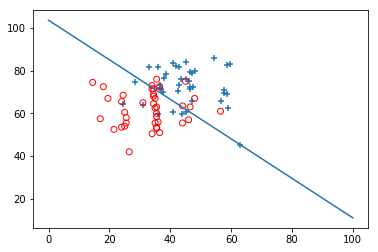


With 13 iterations it converges agreeably.



If someone makes a 20 on exam1 and an 80 on exam2, they have a 0.77501118 or (77.501118%) chance of being admitted.

Plot of final decision boundary:



Corrected code: (Highlighted by comment line of dashes “-“ followed by a comment stating what has changed. Mostly it is just the understanding of the math library behind the functions that has changed.)

1. #!/usr/bin/env python2
2. # -\*- coding: utf-8 -\*-
3. """
4. Created on Mon Sep 11 20:23:00 2017
6. @author: rditljtd
7. """
9. import numpy as np
10. from mpl\_toolkits.mplot3d import Axes3D
11. import matplotlib.pyplot as plt
12. from scipy.special import expit
14. x = np.loadtxt('cx.dat')
15. y = np.loadtxt('cy.dat')
17. #--------------------------------------------------------------------------------------------
18. #Changed: added test values
19. test = [2.0e+01, 8.0e+01]
20. #--------------------------------------------------------------------------------------------

23. #Get the number of examples
24. m = x.shape[0]
26. #Reshape x to be a 2D column vector
27. x.shape = (m,2)
29. #Add a column of ones to x
30. X = np.hstack([np.ones((m,1)), x])
32. #initialize theta
33. theta = np.zeros(shape=(3,1)) #Initialize theta
35. iterations = 13
36. J\_history = np.zeros(shape = (iterations, 1))
38. pos = np.nonzero(y)
39. neg = np.where(y==0)[0]
41. plt.scatter(x[pos,0],x[pos,1],marker='+')
42. plt.scatter(x[neg,0],x[neg,1],facecolors='none',marker='o', color='r')
43. plt.show()

46. #Newtons method solution
48. def sigmoid(z):
49. toreturn = expit(z)
50. **return** toreturn
52. def hypothesis(X, theta):
54. #--------------------------------------------------------------------------------------------
55. #Changed: math logic - np.transpose to np.dot
56. toreturn = (sigmoid(np.dot(X, theta)))
57. #--------------------------------------------------------------------------------------------
59. **return** toreturn
61. def computeCost(X, y, theta):
62. m = y.size
64. #--------------------------------------------------------------------------------------------
65. #Changed: math logic - correct use of \* instead of np.dot
66. toreturn = ((1.0/m) \* (-y\*np.log(hypothesis(X, theta))) - (1-y)\*(np.log(1-hypothesis(X, theta)))).sum()
67. #--------------------------------------------------------------------------------------------
69. #print "computeCost: " + `toreturn`
70. **return** toreturn
72. def gradientDescent(X, y, theta):
73. m = y.size
75. #--------------------------------------------------------------------------------------------
76. #Changed: math logic - correct use of "-" instead of np.subtract
77. minus = hypothesis(X, theta) - y
78. #--------------------------------------------------------------------------------------------
80. #--------------------------------------------------------------------------------------------
81. #Changed: math logic - use of X.T.dot instea of just np.dot
82. toreturn = (1.0/m)\*(X.T.dot(minus))
83. #--------------------------------------------------------------------------------------------
85. **return** toreturn
87. def hessian(X, y, theta):
88. m = y.size
89. minus = (1-hypothesis(X,theta))\*(hypothesis(X, theta))
91. #--------------------------------------------------------------------------------------------
92. #Changed: math logic on multiplying minus and X and transposing
93. Xm = minus\*X
94. result = (1.0/m)\*(Xm.T.dot(X))
95. #--------------------------------------------------------------------------------------------
97. toreturn = result
98. # print "hessian: " + `toreturn`
99. **return** toreturn
101. def newtonsMethod(X, y, theta, iterations):
102. m=y.size
103. y.shape = (m,1)
104. **for** i in range(iterations):
105. J\_history[i] = computeCost(X, y, theta)
107. #--------------------------------------------------------------------------------------------
108. #Changed: used np.linalg.inv instead of 1/x and used np.dot instead of \*
109. theta = (theta - np.linalg.inv((hessian(X, y, theta))).dot(gradientDescent(X, y, theta)))
110. #--------------------------------------------------------------------------------------------
112. print theta, J\_history
113. **return** theta, J\_history
115. theta, J\_history = newtonsMethod(X, y, theta, iterations)
116. print 'Final theta: ', theta
118. plt.plot(range(iterations), J\_history)
119. plt.xlabel('Number of iterations')
120. plt.ylabel('Cost J')
121. plt.show()
123. print np.array([1, test[0], test[1]]).dot(theta)
124. predict1 = np.array([1, test[0], test[1]]).dot(theta).flatten()
125. print predict1
127. #--------------------------------------------------------------------------------------------
128. #Changed: Added code to plot decision boundary
129. theta = theta[:,0]  # Make theta a 1-d array.
130. x2 = np.linspace(0,100, 80)
131. y2 = -(theta[0] + theta[1]\*x2)/theta[2]
132. plt.scatter(x[pos,0],x[pos,1],marker='+')
133. plt.scatter(x[neg,0],x[neg,1],facecolors='none',marker='o', color='r')
134. plt.plot(x2, y2)
135. plt.show
136. #--------------------------------------------------------------------------------------------