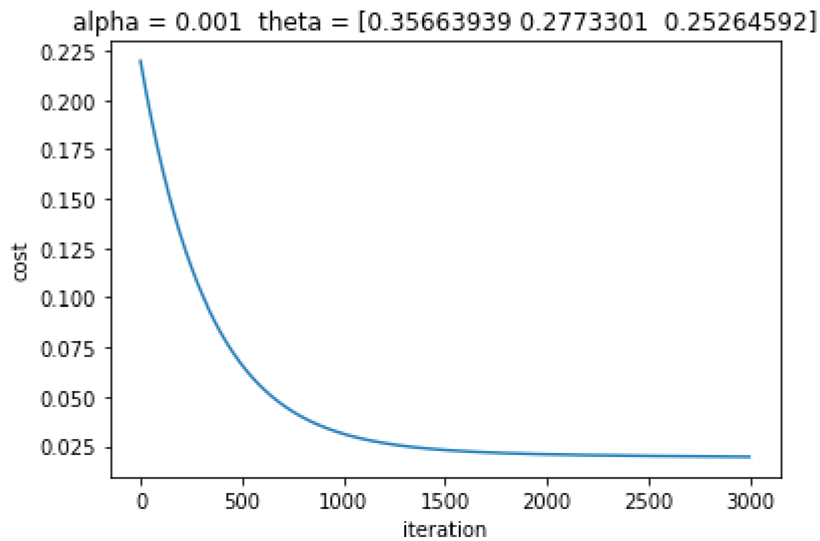
Tom Nguyen
Dr. Xiaobai Liu
CS 596 MW
March 08, 2018

Homework Assignment 3

Placeholder 1

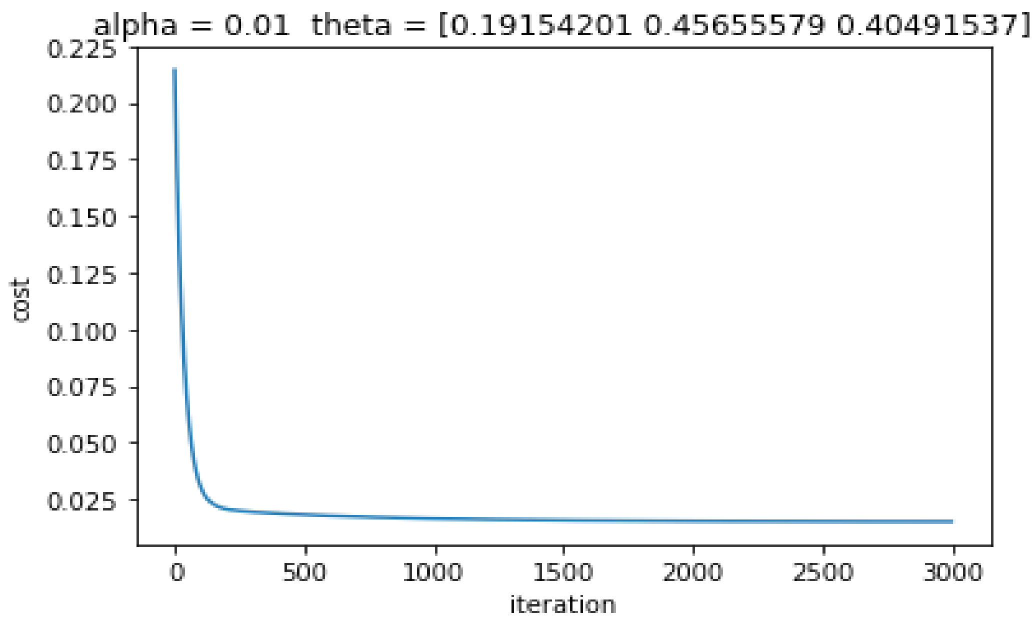
ALPHA = 0.001
MAX_ITER = 3000
Convergences between 1,000 – 1,500
normalization method = rescaleMatrix(sat)




```
results: 0.1441621797645365 (0.1186983906925166З)
```

It will converge slower the lower alpha is because it is our learning rate. When the learning rate is lower our curve will take longer to match up with our data set. On the other hand, when alpha is lower our curve will be more gradual and smooth compare to a higher alpha.

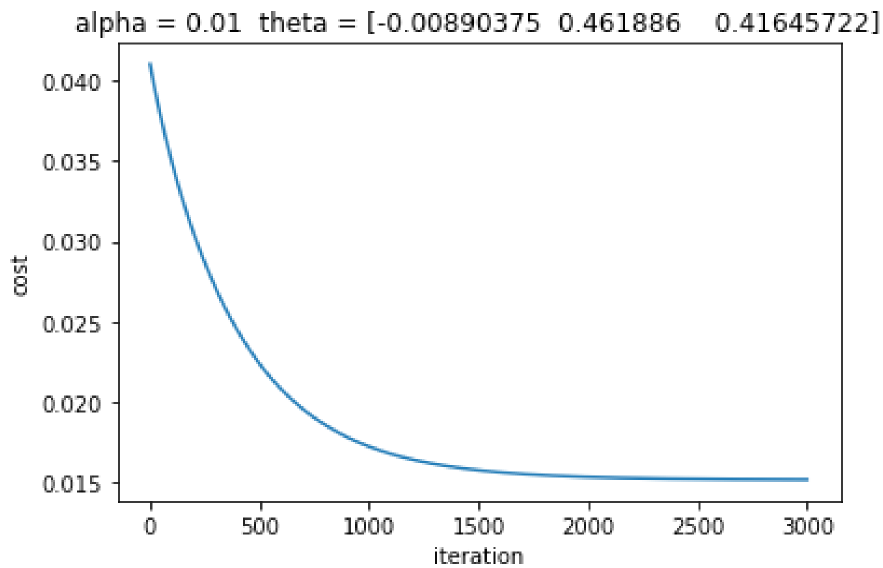
ALPHA = 0.01
MAX_ITER = 3000
Convergences between 0 – 500
normalization method = rescaleMatrix(sat)




```
results: 0.1695064002866059 (0.12500719127595206)
```

Since, we have a higher alpha here notices how the curve is a bit sharper. The data convergences at a faster rate then the graph above. Changing the iterations will be the same curve but longer. We will need at least 500 iterations to get that nice curve.

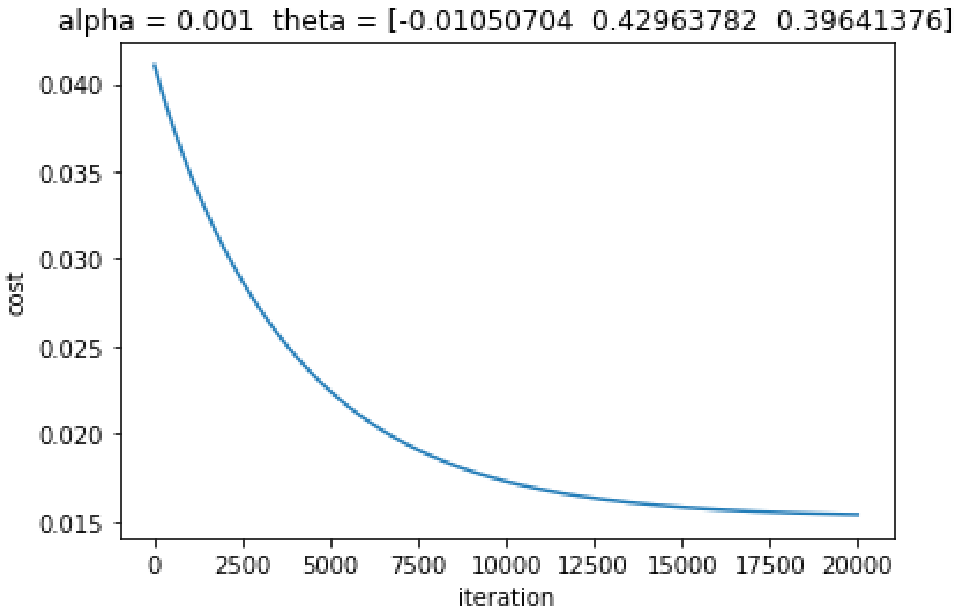
ALPHA = 0.01
MAX_ITER = 3000
Convergences between 1,000 – 1,500
normalization method = meanMatrix(sat)




```
results: 0.17116649903614495 (0.1261600727061402)
```

This is very similar to the rescale normalization method but we need a little bit higher alpha, about 10 higher in order to get the same curve. So the curve above when using the rescaling normalization method, would look similar to our mean normalization method if we were to change our alpha to 0.1. Note that the theta is different but we still reach the same conclusion. Next instead of changing my alpha to 0.1 I will change it to alpha 0.001.

ALPHA = 0.001
MAX_ITER = 20,000
Convergences between 7,500 – 10,000
normalization method = meanMatrix(sat)



results: 0.166000129298324 (0.1235794402035036 6)

Just changing alpha from 0.01 to 0.001 will take about 6,000 more iterations. So, it will not take 10 times longer as one would guess but instead it would take about 5 to 7 times longer. Since we needed more iterations here, the max iterations was changed to 20,000 so that we could see that nice curve.

Placeholder 2

My version of mean Matrix (python code):

```
def meanMatrix(dataMatrix):
    colCount = len(dataMatrix[0])#3
    rowCount = len(dataMatrix)#105
    new_Value_0 = []
    new_Value_1 = []
    new_Value_2 = []
    new_Value = []
    newMatrix = np.zeros(dataMatrix.shape)
    for row in range(rowCount):
        new_Value_0.append(dataMatrix[row][0])
        new_Value_1.append(dataMatrix[row][1])
        new_Value_2.append(dataMatrix[row][2])
    new_Value_0 = meanNormalization(new_Value_0)
    new_Value_1 = meanNormalization(new_Value_1)
    new_Value_2 = meanNormalization(new_Value_2)
    for row in range(rowCount):
        col=0
        newMatrix[row][col] = new_Value_0[row]
        col+=1
        newMatrix[row][col] = new_Value_1[row]
        col+=1
        newMatrix[row][col] = new_Value_2[row]
    return newMatrix
```

What I did here was use the function meanNormalization by Dr. Xiaobai Liu and incorporated it into my code. Since meanNormalization takes in an argument of a single array and returns a new array with the updates in it, I stripped our size (x, 3) and turn it into three (x,1) array and passed it in. I could have done this method more efficiently by having count the column arbitrary and have the code do all the work of guessing how many columns there were. Instead, I hard coded the mainly because I was afraid of making an array containing column size and each array slot I would hold a large amount of data would cause some kind of error. I suspect it would work just fine. After passing it into meanNormalization, normalizing each column. I recombined it back to a single matrix and pass it back out.

Placeholder 3
Updating theta

```
residualError=np.dot(X,theta)-y
gradient = 1/m * np.dot(transposedX, residualError)
change = [alpha * x for x in gradient]
theta = np.subtract(theta, change)
```

Placeholder 4
Cost

```
residualError = np.dot(X,theta)-y
atmp = (1/(2*m))*np.sum(np.square(np.dot(X,theta)-y))
```

Sidenote:

At the time of writing the code, I didn't know how to import the file properly into jupyter so I copied all the code into a single file.

How to properly load and write to an imported file. Instead of using from download_data.py import download_data:
load command : %load path
example: %load GP.py
save file : %%writefile path
example: %%writefile GP.py

```
from pandas import read_table
import numpy as np
import matplotlib.pyplot as plt
# Starting codes for ha3 of CS596.

#NOTICE: Fill in the codes between "%PLACEHOLDER#start" and "PLACEHOLDER#end"

# There are two PLACEHODERS IN THIS SCRIPT

# parameters


def download_data(fileLocation, fields):

    frame = read_table(
        fileLocation,

        # Specify the file encoding
        # Latin-1 is common for data from US sources
        encoding='latin-1',
        #encoding='utf-8',  # UTF-8 is also common

        # Specify the separator in the data
        sep=',',            # comma separated values

        # Ignore spaces after the separator
        skipinitialspace=True,

        # Generate row labels from each row number
        index_col=None,

        # Generate column headers row from each column number
        header=0,          # use the first line as headers

        usecols=fields
    )

    # Return the entire frame
    return frame



# X          - single array/vector
# y          - single array/vector
# theta      - single array/vector
# alpha      - scalar
# iterations - scarlar

def gradientDescent(X, y, theta, alpha, numIterations):
```

```
    '''
    # This function returns a tuple (theta, Cost array)
    '''
    m = len(y)#len(y) =60
    #print(X)#[[1.         0.62871287 0.43253968]
    arrCost =[];
    transposedX = np.transpose(X) # transpose X into a vector  -> XColCount X m matrix
    for interation in range(0, numIterations):
        ################PLACEHOLDER3 #start##########################
        #: write your codes to update theta, i.e., the parameters to estimate.
        #ground_truth - product_of_x
        #residualError = np.dot(X,theta) - y
        #residualError = np.subtract(np.dot(X,theta), y) #error
        residualError=np.dot(X,theta)-y
        #print(residualError)#[0.83236994 0.47976879 0.1849711  0.80346821 0.80346821 0.16763006
        #gradient =  (1/(2*numIterations))*np.square(np.sum(y-theta*x))#cost function error
        #gradient = np.dot(residualError,X)
        gradient = 1/m * np.dot(transposedX, residualError)
        change = [alpha * x for x in gradient]
        theta = np.subtract(theta, change)  # theta = theta - alpha * gradient
        ################PLACEHOLDER3 #end##########################

        ################PLACEHOLDER4 #start##########################
        # calculate the current cost with the new theta;
        #print(y)#[1,...,60]
        #print(theta*X)#[[1,2,3],...,[60,60,60]]
        #apple=np.sum(theta*X)
        #print(apple)
        #residualError = y - np.dot(X,theta)
        #residualError = np.subtract(np.dot(X,theta), y)
        residualError = np.dot(X,theta)-y
        #print(residualError)
        #atmp =  (1/(2*m))*np.square(residualError)
        atmp = (1/(2*m))*np.sum(np.square(np.dot(X,theta)-y))
        #atmp = (1 / (2*m)) * np.sum(residualError**2)
        #print(atmp)
        arrCost.append(atmp)
        # cost = (1 / m) * np.sum(residualError ** 2)
        ################PLACEHOLDER4 #start##########################

    return theta, arrCost

def rescaleNormalization(dataArray):
    min = dataArray.min()
    denom = dataArray.max() - min
    newValues = []
    for x in dataArray:
        newX = (x - min) / denom
        newValues.append(newX)
```

```
    return newValues

def rescaleMatrix(dataMatrix):
    colCount = len(dataMatrix[0])
    rowCount = len(dataMatrix)
    newMatrix = np.zeros(dataMatrix.shape)
    for i in range(0, colCount):
        min = dataMatrix[:,i].min()
        denom = dataMatrix[:,i].max() - min
        for k in range(0, rowCount):
            newX = (dataMatrix[k,i] - min) / denom
            newMatrix[k,i] = newX
    return newMatrix


def meanNormalization(dataArray):
    mean = np.mean(dataArray)
    denom = np.max(dataArray) - np.min(dataArray)
    newValues = []
    for x in dataArray:
        newX = (x - mean) / denom
        newValues.append(newX)
    return newValues

def meanMatrix(dataMatrix):
    colCount = len(dataMatrix[0])#3
    rowCount = len(dataMatrix)#105
    new_Value_0 = []
    new_Value_1 = []
    new_Value_2 = []
    new_Value = []
    newMatrix = np.zeros(dataMatrix.shape)
    for row in range(rowCount):
        new_Value_0.append(dataMatrix[row][0])
        new_Value_1.append(dataMatrix[row][1])
        new_Value_2.append(dataMatrix[row][2])
    new_Value_0 = meanNormalization(new_Value_0)
    new_Value_1 = meanNormalization(new_Value_1)
    new_Value_2 = meanNormalization(new_Value_2)
    for row in range(rowCount):
        col=0
        newMatrix[row][col] = new_Value_0[row]
        col+=1
        newMatrix[row][col] = new_Value_1[row]
        col+=1
        newMatrix[row][col] = new_Value_2[row]
    return newMatrix
```

```
################PLACEHOLDER1 #start###########################
# test multiple learning rates and report their convergence curves.
ALPHA = 0.001
MAX_ITER = 20000
################PLACEHOLDER1 #end###########################

#% step-1: load data and divide it into two subsets, used for training and testing
sat = download_data('sat.csv', [1, 2, 4]).values # three columns: MATH SAT, VERB SAT, UNI. GPA  #
convert frame to matrix
#print(sat)#[643.   589.     3.52]
#(math_SAT. verb_SAT. univ_GPA)
################PLACEHOLDER2 #start###########################
# Normalize data
#sat = meanNormalization(sat) #doesn't work
#sat = rescaleMatrix(sat) # please replace this code with your own codes
#print(sat)#[0.62871287 0.43253968 0.83236994]

sat = meanMatrix(sat)
#print(sat)
################PLACEHOLDER2 #end###########################


# training data;
satTrain = sat[0:60, :]
# testing data;
satTest = sat[60:len(sat),:]

#% step-2: train a linear regression model using the Gradient Descent (GD) method
# ** theta and xValues have 3 columns since have 2 features: y = (theta * x^0) + (theta * x^1) + (theta
* x^2)
theta = np.zeros(3)

xValues = np.ones((60, 3)) #print(xValues)#[1. 1. 1.]
xValues[:, 1:3] = satTrain[:, 0:2]#print(xValues)#[1.         0.62871287 0.43253968]
yValues = satTrain[:, 2]#print(yValues)#[0.83236994 0.47976879 0.1849711  0.80346821 0.80346821
0.16763006
# call the GD algorithm, placeholders in the function gradientDescent()
[theta, arrCost] = gradientDescent(xValues, yValues, theta, ALPHA, MAX_ITER)

#visualize the convergence curve
plt.plot(range(0,len(arrCost)),arrCost);
plt.xlabel('iteration')
plt.ylabel('cost')
plt.title('alpha = {}  theta = {}'.format(ALPHA, theta))
plt.show()

#% step-3: testing
testXValues = np.ones((len(satTest), 3))
testXValues[:, 1:3] = satTest[:, 0:2]
```

```
tVal =  testXValues.dot(theta)


#% step-4: evaluation
# calculate average error and standard deviation
tError = np.sqrt([x**2 for x in np.subtract(tVal, satTest[:, 2])])
print('results: {} ({})'.format(np.mean(tError), np.std(tError)))
```