CSE 151: Programming Assignment #4

Due on Thursday, May 25, 2016

Mangione-Tran,Carmine 9AM

Kyle Lee, Jaehee Park
A01614951, A11287366

Graphics are located at the end.

Homework Code

```
import numpy as np
import csv
import random
import math
import operator
from decimal import *
from numpy import genfromtxt
import matplotlib.pyplot as plt

##### METHODS #####
# Performs z-scaling on matrix
# @param: matrix - The matrix/array to scale
# @return: matrix - The matrix that has been scaled by the mean of each
#                   column and the standard deviation of each column
def zscale(matrix, means,stdev):
    offset = len(means)-2
    matrix[:,0:offset] = (matrix[:,0:offset] - means[0:offset])/(stdev[0:offset])
    return matrix

# Calculate the distance between two instances of data
# @param: data1 - First data to compare to
# @param: data2 - Second data to compare to
# @param: length - Size of data
def calculateDistance(data1, data2, length):
    distance = 0
    for x in range(length):
        distance+=pow((data1[x] - data2[x]),2)
    return math.sqrt(distance)
#method for generating hits
#@param:x => our data
#       :testSize => hit rate
#       :genList => list of # of hits per index
#@return:genList => updated list with hits per index
def count(x, testSize, genList):
    size = len(x)
    expectedDraws = int(round(size*testSize))
    x1 = Decimal(expectedDraws)/Decimal(size)

    j = 0

    for i in range(0, size):
        # Compare with random uniform
        x2 = random.uniform(0,1)
        if x2 < x1:
            genList[i] = genList[i] + 1
            j = j+1
        # Update x1 to new conditional probability
        x1 = Decimal(expectedDraws-j)/Decimal(size-i)
```

```
    return genList

#method for separating input data based on counter
#@param: counter => a vector that stores the indices of test/train sets
#@param: inputData => original data
#@return: trainingSet - Training Set based on data
#@return: testSet - Test Set based on data
def separateSet(counter,inputData):
    trainingSet = []
    testSet = []
    size = len(inputData)
    for i in range(size):
        if counter[i] == 1:
            trainingSet.append(inputData1[i])
        else:
            testSet.append(inputData1[i])
    return trainingSet, testSet

#method to initialize QR Decomposition on a matrix X
#@param: X - the matrix we want to decompose. We note that the
#            X is not an augmented matrix
def QRdecompose(X):

    # Copy
    R = X.copy()
    # Store the shape of A
    [m, n] = X.shape
    # Create identity matrix of size m
    Q = np.identity(m)

    # Applying method 1. Recursively define H
    for i in range(n - (m == n)):

        # Empty householder identity matrix
        H = np.identity(m)

        # Create householder matrix
        H[i:, i:] = householdervk(R[i:, i])

        # Recreate Q,R
        Q = np.dot(Q, H)
        R = np.dot(H, R)
    return [Q,R]

def householdervk(X):

    # Determine shift
    vk = X / (X[0] + np.copysign(np.linalg.norm(X), X[0]))

    # e1
    vk[0] = 1

    # Take the shape of each R[i:,i]
```

```
    H = np.identity(X.shape[0])

    # Create householder matrix
    H = H - (2 / np.dot(vk, vk)) * np.dot(vk[:, None], vk[None, :])
    return H

#method for back substitution a matrix A with b
#@param: R - Upper right triangular matrix already formatted
#@param: b - A nx1 vector
def backsolve(R,b):
    n = np.size(b)
    x = np.zeros((n,1))

    # Start at the end, solve accordingly
    for i in range(n-1,-1,-1):
        x[i] = (b[i] - np.dot(R[i,:],x))/R[i,i]

    return x

#method for adjusting to method 2. Remove (m-n) rows and adjust
#sizes accordingly
#@param: Q - a matrix
#@param: R - an upper right triangular matrix with rows of 0's
def simplifyQR(Q,R):
   [Rm,Rn] = R.shape
   # Remove rows of zeros and adjust Q matrix to match dimensions
   # for np.dot
   if Rm != Rn:
       R = R[0:Rn,:]
       Q = Q[:,0:Rn]
   return Q,R

#method for calculating the root mean squared error
#@param: YActual - the correct vector
#@param: YEstimated - the estimated vector after QR Decomposition
def rmse(Y_actual,Y_estimated):
    return np.sqrt(np.mean((Y_actual-Y_estimated)**2))

# Method to pick random centroids from data set
def randomCentroid(inputData, k):

    size = len(inputData)
    x1 = Decimal(k)/Decimal(size)

    centroids = [] #stores hit rates on index
    j = 0

    for i in range(0, size):
        # Compare with random uniform
        x2 = random.uniform(0,1)
        if x2 < x1:
            centroids.append(inputData[i])
            j = j+1
```

```
        # Update x1 to new conditional probability
        x1 = Decimal(k-j)/Decimal(size-i)
    return np.array(centroids)

# method to cluster a matrix with the means
#@param: X - the data matrix (already cleaned)
#@param: mu - the collection of means
def cluster(X, mu):
    clusters  = {}
    for x in X:
        mu2 = min([(i[0], np.linalg.norm(x-mu[i[0]])) \
                    for i in enumerate(mu)], key=lambda t:t[1])[0]
        # Attach a matrix that satisfies the argmin into a particular centroid
        try:
            clusters[mu2].append(x)
        except KeyError:
            clusters[mu2] = [x]
    return clusters

# method to update the means after each iteration
#@param: mu - the centroids, or means
#@param: clusters- the clusters belonging to each centroid
def updateMid(mu, clusters):
    mymu = []
    keys = sorted(clusters.keys())
    for k in keys:
        mymu.append(np.mean(clusters[k], axis = 0))
    return mymu

# method to check convergence by checking the entries of the centroids
# and the old centroids to see if they are equal
#@param: mu - centroids
#@param: old - old centroids
def converged(mu, old):
    return (set([tuple(a) for a in mu]) == set([tuple(a) for a in old]))

# method to find the centers given a data matrix and K-means
#@param: X - the data matrix
#@param: K - the number of centroids desired by user
def find_centers(X, K):
    # Initialize to K random centers
    old = randomCentroid(X, K)
    mu = randomCentroid(X, K)
    while not converged(mu, old):
        old = mu
        # Assign all points in X to clusters
        clusters = cluster(X, mu)
        # Reevaluate centers
        mu = updateMid(old, clusters)
    return(mu, clusters)
# method to find the total within sum of clusters
#@param: mu - the centroids
#@param: clusters- the clusters with respect to each centroid
```

```
def calculate_wss(mu,clusters):
    n = len(mu)
    total = 0
    for i in range(n):
        ithsum = 0

        # Check cluster by cluster
        for j in range(len(clusters[i])):
            ithsum = ithsum + sum((clusters[i][j] -mu[i])**2)

        # Sum all the inner cluster total wss
        total = total + ithsum
    return total

# Method to calculate the total rmse of a cluster
#@param: Y_actual - cluster
#@param: Y_estimated - cluster mean
def calculate_total_rmse(rmse_array):
    total = 0

    # Sum all the rmse
    for i in rmse_array:
        total = total + i
    return total

################# ABALONE DATA SET #################
# Read CSV Files
inputFile = open('abalone.data')
inputReader = csv.reader(inputFile)
inputData = list(inputReader)   #inputData = list of our data (which is in lists)


# Initiaize test Size
trainingSize = 0.9

# Length of the data
size = len(inputData)

# Initialize counter
counter = [] #stores hit rates on index

# initialize counter array
for x in range(size):
    counter.append(0)

# Find test size
for i in range(1,2):
    counter = count(inputData, trainingSize, counter)

for i in range(size):
    if inputData[i][0] == 'M':
        inputData[i][0] = 0
    elif inputData[i][0] == 'F':
```

```
        inputData[i][0] = 1
    elif inputData[i][0] == 'I':
        inputData[i][0] = 2

# Add 3 columns for classification of sex
proxiedData = np.zeros((size,len(inputData[0])+3))
proxiedData[:,:-3] = inputData
# Fix data for Euclidean distance
for i in range(size):
    if proxiedData[i][0] == 0:
        proxiedData[i][9] = 1
    elif proxiedData[i][0] == 1:
        proxiedData[i][10] = 1
    elif proxiedData[i][0] == 2:
        proxiedData[i][11] = 1

# Remove first column
proxiedData = proxiedData[:,1:12]

# Swap the actual to the predictions
proxiedData[:,[7,8,9,10]] = proxiedData[:,[8,9,10,7]]

# Keep track of counter and create training and testing sets
trainIndex = []
testIndex = []
trainingSet = []
testSet = []

# Create training and testing sets
for i in range(size):
    if counter[i] == 1:
        trainIndex.append(i)
        trainingSet.append(proxiedData[i])
    else:
        testIndex.append(i)
        testSet.append(proxiedData[i])


# Change to numeric arrays
trainingSet = np.array(trainingSet)
testSet = np.array(testSet)

X_train = trainingSet[:,:-1]
Y_train= trainingSet[:,-1]
X_test = testSet[:,:-1]
Y_test = testSet[:,-1]

#ZSCALING
means = X_train.mean(axis=0)
stdevs = X_train.std(axis=0)
X_train_scaled = zscale(X_train, means, stdevs)
X_test_scaled = zscale(X_test,means,stdevs)
```

```
train_means = trainingSet.mean(axis=0)
train_stdevs = trainingSet.std(axis=0)

train_set_scaled = zscale(trainingSet,train_means,train_stdevs)
test_set_scaled = zscale(testSet,train_means,train_stdevs)

fmean = []
fstd = []
fmean1 = []
fstd1 = []

wcssArray = []
rmseArray = []

# K = 1
test1,test2 = find_centers(train_set_scaled,1)
error1 = calculate_wss(test1,test2)

# Fix dimension
Y_test = Y_test[:,np.newaxis]

print ("K = 1")
print ("centroids = ", test1)
print ("WCSS = ", error1)
for k in range(0, len(test2)):
    for j in range(0,len(test2[k][0])):
        a = np.array(test2[k])
        b = a[:,j]
        a2 = np.array(b)
        fmean.append(np.mean(a2))
        fstd.append(np.std(a2))
        fmean1.append(fmean[0])
        fstd1.append(fstd[0])
        fmean = []
        fstd = []
    print ("cluster #",k+1)
    print ("MEAN =", fmean1)
    fmean1 = []
    print ("STDEV =", fstd1)
    fstd1 = []

totalRMSE = 0
for k in range(0, len(test2)):
    a = np.array(test2[k])

    # Perform QR Decomposition and backsolving
    Q,R = QRdecompose(a[:,:-1])
    Y_train = a[:,-1]
    # Check if QR decomposition was successful
    np.dot(Q,R)

    # Fixing and backsolving
    Q,R = simplifyQR(Q,R)
```

```
    Z = np.dot(Q.T,Y_train)
    beta = backsolve(R,Z)

    RMSE = rmse(np.dot(X_test,beta), Y_test)
    print ("RMSE IS ", RMSE)
    totalRMSE = totalRMSE + RMSE

print ("RMSE = ", totalRMSE)
wcssArray.append(error1)
rmseArray.append(totalRMSE)
print ("--------------------------------------------------------------")

# K = 2
test1,test2 = find_centers(train_set_scaled,2)
error1 = calculate_wss(test1,test2)
print ("K = 2")
print ("centroids = ", test1)
print ("WCSS = ", error1)
for k in range(0, len(test2)):
    for j in range(0,len(test2[k][0])):
        a = np.array(test2[k])
        b = a[:,j]
        a2 = np.array(b)
        fmean.append(np.mean(a2))
        fstd.append(np.std(a2))
        fmean1.append(fmean[0])
        fstd1.append(fstd[0])
        fmean = []
        fstd = []
    print ("cluster #",k+1)
    print ("MEAN =", fmean1)
    fmean1 = []
    print ("STDEV =", fstd1)
    fstd1 = []

totalRMSE = 0
for k in range(0, len(test2)):
    a = np.array(test2[k])

    # Perform QR Decomposition and backsolving
    Q,R = QRdecompose(a[:,:-1])
    Y_train = a[:,-1]
    # Check if QR decomposition was successful
    np.dot(Q,R)

    # Fixing and backsolving
    Q,R = simplifyQR(Q,R)
    Z = np.dot(Q.T,Y_train)
    beta = backsolve(R,Z)

    RMSE = rmse(np.dot(X_test,beta), Y_test)
    print ("RMSE IS ", RMSE)
    totalRMSE = totalRMSE + RMSE
```

```
print ("RMSE = ", totalRMSE)
wcssArray.append(error1)
rmseArray.append(totalRMSE)
print ("------------------------------------------------------------")

# K = 4
test1,test2 = find_centers(train_set_scaled,4)
error1 = calculate_wss(test1,test2)
print ("K = 4")
print ("centroids = ", test1)
print ("WCSS = ", error1)
for k in range(0, len(test2)):
    for j in range(0,len(test2[k][0])):
        a = np.array(test2[k])
        b = a[:,j]
        a2 = np.array(b)
        fmean.append(np.mean(a2))
        fstd.append(np.std(a2))
        fmean1.append(fmean[0])
        fstd1.append(fstd[0])
        fmean = []
        fstd = []
    print ("cluster #",k+1)
    print ("MEAN =", fmean1)
    fmean1 = []
    print ("STDEV =", fstd1)
    fstd1 = []

totalRMSE = 0
for k in range(0, len(test2)):
    a = np.array(test2[k])

    # Perform QR Decomposition and backsolving
    Q,R = QRdecompose(a[:,:-1])
    Y_train = a[:,-1]
    # Check if QR decomposition was successful
    np.dot(Q,R)

    # Fixing and backsolving
    Q,R = simplifyQR(Q,R)
    Z = np.dot(Q.T,Y_train)
    beta = backsolve(R,Z)

    RMSE = rmse(np.dot(X_test,beta), Y_test)
    print ("RMSE IS ", RMSE)
    totalRMSE = totalRMSE + RMSE

print ("RMSE = ", totalRMSE)
wcssArray.append(error1)
rmseArray.append(totalRMSE)
print ("------------------------------------------------------------")
```

```
# K = 8
test1,test2 = find_centers(train_set_scaled,8)
error1 = calculate_wss(test1,test2)
print ("K = 8")
print ("centroids = ", test1)
print ("WCSS = ", error1)
for k in range(0, len(test2)):
    for j in range(0,len(test2[k][0])):
        a = np.array(test2[k])
        b = a[:,j]
        a2 = np.array(b)
        fmean.append(np.mean(a2))
        fstd.append(np.std(a2))
        fmean1.append(fmean[0])
        fstd1.append(fstd[0])
        fmean = []
        fstd = []
    print ("cluster #",k+1)
    print ("MEAN =", fmean1)
    fmean1 = []
    print ("STDEV =", fstd1)
    fstd1 = []

totalRMSE = 0
for k in range(0, len(test2)):
    a = np.array(test2[k])

    # Perform QR Decomposition and backsolving
    Q,R = QRdecompose(a[:,:-1])
    Y_train = a[:,-1]
    # Check if QR decomposition was successful
    np.dot(Q,R)

    # Fixing and backsolving
    Q,R = simplifyQR(Q,R)
    Z = np.dot(Q.T,Y_train)
    beta = backsolve(R,Z)

    RMSE = rmse(np.dot(X_test,beta), Y_test)
    print ("RMSE IS ", RMSE)
    totalRMSE = totalRMSE + RMSE

print ("RMSE = ", totalRMSE)
wcssArray.append(error1)
rmseArray.append(totalRMSE)
print ("-------------------------------------------------------------")

# K = 16
test1,test2 = find_centers(train_set_scaled,16)
error1 = calculate_wss(test1,test2)
print ("K = 16")
print ("centroids = ", test1)
print ("WCSS = ", error1)
```

```
for k in range(0, len(test2)):
    for j in range(0,len(test2[k][0])):
        a = np.array(test2[k])
        b = a[:,j]
        a2 = np.array(b)
        fmean.append(np.mean(a2))
        fstd.append(np.std(a2))
        fmean1.append(fmean[0])
        fstd1.append(fstd[0])
        fmean = []
        fstd = []
    print ("cluster #",k+1)
    print ("MEAN =", fmean1)
    fmean1 = []
    print ("STDEV =", fstd1)
    fstd1 = []

totalRMSE = 0
for k in range(0, len(test2)):
    a = np.array(test2[k])

    # Perform QR Decomposition and backsolving
    Q,R = QRdecompose(a[:,:-1])
    Y_train = a[:,-1]
    # Check if QR decomposition was successful
    np.dot(Q,R)

    # Fixing and backsolving
    Q,R = simplifyQR(Q,R)
    Z = np.dot(Q.T,Y_train)
    beta = backsolve(R,Z)

    RMSE = rmse(np.dot(X_test,beta), Y_test)
    print ("RMSE IS ", RMSE)
    totalRMSE = totalRMSE + RMSE

print ("RMSE = ", totalRMSE)
wcssArray.append(error1)
rmseArray.append(totalRMSE)
print ("-------------------------------------------------------------")

print ("WCSS array = ", wcssArray)
print ("RMSE array = ", rmseArray)

plt.figure(0)
plt.plot([1,2,4,8,16], wcssArray,'ro')
plt.title('K vs WCSS')
plt.xlabel('K')
plt.ylabel('WCSS')
plt.show()

plt.figure(1)
plt.plot([1,2,4,8,16], rmseArray,'ro')
```

```
plt.title('K vs RMSE')
plt.xlabel('K')
plt.ylabel('RMSE')
plt.show()
# End
```

Console Output

```
K = 1
     Centroid = [  0,    0,   0, 0,   0,   0, 0,    0,   0, 0.321447578,    9.93560404]
     WCSS = 73636.107504
     Cluster #1
o    Mean = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0.32144757849920169, 9.9356040447046308]
o    Stdev = [1.0000000000000031, 1.000000000000002, 0.99999999999999856, 1.0000000000000002, 1.00000
     RMSE = 2.16441629043
K = 2
     Centroids
o     #1 =  [-0.71696688, -0.72698104, -0.67580905, -0.74062862, -0.67616611, -0.72601942, -0.754260
0.57410468, 7.68264463]
o    #2 = [  0.66973489,    0.67908934,    0.63128844,    0.69183785, 0.63162197,    0.67819107,
0.70457142,    0.20999688, 0.29094812,    0.08543489,   12.04014411]
     WCSS = 42434.6263608
     Cluster #1
o    Mean = [-0.71696688062211289, -0.7269810420626206, -0.67580905253796242, 0.7406286172239912, -0.
o    STDEV = [0.85313371374299685, 0.84083844906065774, 0.69703954926709311, 0.56171270111326621, 0.5
     Cluster #2
o    Mean = [0.66973488848643115, 0.67908934191644721, 0.63128843559258974, 0.69183784882220489, 0.63
o    STDEV = [0.57053562560960958, 0.56466077709413331, 0.80938887323436848, 0.80520342894758334, 0.8
     RMSE = 2.318294322







K = 4
     Centroid
o    #1 = [  0.5069816 ,    0.56489021,    0.65177682,    0.57505409, 0.2618512 ,    0.48775245,
0.84368215,    0.17453959,   0.29740037,    0.09952607,   16.5971564 ]
o    #2 = [-0.07318796, -0.07968438, -0.13430602, -0.29112721, -0.25619319,  -0.27977782, -0.290053
0.03650284, -0.03229531,  0.31883024, 9.32310984]
o    #3 = [  0.94158366,    0.93416607,    0.83045811,    1.04158118, 1.0634222 ,    1.04397664,
0.94129666,    0.2646502, 0.35190663,    0.03085299,   10.79854809]
o    #4 = [-1.38096384, -1.38956391, -1.2042279 , -1.18069102, -1.10962693, -1.15870829, -1.1859229 ,
0.82331731, 6.44591346]
     WCSS = 24707.3742191
     Cluster #1
o    Mean = [0.50698160306609308, 0.56489020524775013, 0.65177681968023704, 0.57505409155320963, 0.261
o    STDEV = [0.60150776258367777, 0.61622933175248096, 0.67297503535455261, 0.83489576587823, 0.730
     Cluster #2
o    Mean = [-0.073187959734737 99, -0.079684376999973383, -0.13430602274393769, -0.29112721451275314,
o    STDEV = [0.50650295177267868, 0.49700259271325514, 0.47139164102532194, 0.44846165491350215, 0.5
```

```
     Cluster #3
o    Mean = [0.94158366050504216, 0.93416606649836509, 0.83045811256129254, 1.0415811804421551, 1.063
o    STDEV = [0.41009092469715247, 0.40773347310716185, 0.87627424159793488, 0.671007068993978, 0.787
     Cluster #4
o    Mean = [-1.3809638351230098, -1.3895639126704156, -1.2042279015151485, -1.180691024912l627, -1.1
o    STDEV = [0.67639488022155569, 0.65458507478677019, 0.534908738627072, 0.2940522827646647 7, 0.331
     RMSE = 2.103287213
K = 8
     Centroid
o    #1 = [  1.34240587,   1.33247194,   1.31921426,   1.79456775, 1.82988455,   1.7625406 ,
1.61211907,   0.30836145, 0.34131083,   0.01470588,  11.66176471]
o    #2 = [-0.3465086 , -0.34834766, -0.33706667, -0.56674345, -0.56203671, -0.55539817, -0.50831647,
,  0.50381679, 9.91030534]
o    #3 = [-0.6119885 , -0.63341246, -0.62005657, -0.76377278, -0.68640679, -0.76129881, -0.77977898,
0.65817091, 7.55622189]
o    #4 = [ 0.2051069 ,   0.24874393,   0.30565139,   0.09756598, -0.09069966,   0.0945577 ,
0.26065972,   0.13026988, 0.28717465,   0.12559242,  13.64454976]
o    #5 = [ 0.61863568,  0.60529664,  0.44817591,  0.50758598,  0.54414349, 0.54697517,
0.43859499, -0.76004331,  1.48358422,  0, 9.55947955]
o    #6 = [  0.723215,  0.77672827,   0.901119,   0.91191041, 0.50151721,   0.73077299,
1.28527417,   0.16940049, 0.33499644,   0.08457711,  18.63681592]
o    #7 = [-1.80641134, -1.80780781, -1.52048138, -1.37153486, -1.30374546, -1.33740451, -1.37430471,
0.88470067, 5.71175166]
o    #8 = [ 0.62324071,  0.6187955,  0.44906744,  0.51588798,  0.59006525, 0.52302311,
0.41277842,  1.21108281, -0.6740433,  0.0504065, 9.62113821]
     WCSS =  16061.5339709
     Cluster #1
o    Mean = [1.3424058749162124, 1.33247194418041, 1.3192142605603281, 1.7945677502486754, 1.82988455
o    STDEV = [0.33834787364796631, 0.33838636769752972, 1.3479445976086311, 0.60943549319993373, 0.79
     Cluster # 2
o    Mean = [-0.34650860183536397, -0.34834765520064254, -0.33706667191568224, -0.56674345355562294,
o    STDEV = [0.51764416843916694, 0.50041747567743389, 0.44240686157308923, 0.37384195695663319, 0.4
     Cluster # 3
o    Mean = [-0.61198850399900107, -0.63341246090829262, -0.62005656574594548, -0.76377278402582549,
o    STDEV = [0.47940752233676215, 0.46981646984373282, 0.4347459292070604, 0.33451778523639902, 0.37
     Cluster # 4
o    Mean = [0.2051068957867789, 0.24874392763655781, 0.30565139207841469, 0.097565980962607643, -0.0
o    STDEV = [0.48557850907773809, 0.48793925083152989, 0.57592144675323709, 0.52720835948740108, 0.5
     Cluster # 5
o    Mean = [0.61863567594575197, 0.60529663661722177, 0.44817590564701032, 0.50758598249223863, 0.54
o    STDEV = [0.38108070571663744, 0.37829657246392862, 0.44896825473288954, 0.46494479381947257, 0.5
     Cluster # 6
o    Mean = [0.72321499669962208, 0.77672826649735982, 0.90111900407511503, 0.91191041478374157, 0.50
o    STDEV = [0.57183429836220168, 0.59512960817783578, 0.61989453164910668, 0.86818159825698182, 0.7
     Cluster # 7
o    Mean = [-1.8064113386743947, -1.8078078069498886, -1.5204813830247421, -1.3715348633853524, -1.3
o    STDEV = [0.5667158410224713, 0.53549972481276253, 0.45094726799208623, 0.17604349974333003, 0.22
     Cluster # 8
o    Mean = [0.62324071184105034, 0.6187955010331162, 0.44906744077354455, 0.51588798210087716, 0.590
o    STDEV = [0.34312680092238473, 0.34655514467950044, 0.4213431513292103, 0.42403080778539026, 0.49
     RMSE = 2.24223245987
K = 16
     Centroids
```

```
o    #1 = [ 0.41460895,   0.44571075,   0.50242684,   0.38789495, 0.1585249 ,   0.36257415,
0.57661174,   1.01641919, -0.6740433 ,   0.14418605,  14.57674419]
o    #2 = [ -0.3876885 ,  -0.3618681 ,  -0.24570022,  -0.54816482, -0.62922253,  -0.52226459,
-0.43655624,  -0.16222506, 0.44792301,   0.192,  11.8 ]
o    #3 = [ 0.71152324,   0.75887438,   0.88976445,   0.91650724, 0.48711405,   0.70724462,
1.30659704,   0.18053445, 0.32048814,   0.0859375 ,  19.7890625 ]
o    #4 = [-0.29271597, -0.33191979, -0.39935204, -0.58917932, -0.52039524, -0.600934
, -0.58122919, -0.76004331, -0.6740433 ,  1 , 8.24836601]
o    #5 =  [-1.00669795, -1.02673931, -0.93025531, -1.03027431, -0.94874273, -1.02053119, -1.0632727
0.85810811, 5.89189189]
o    #6 = [-2.67358169, -2.61591131, -2.1506056 , -1.59486578, -1.53134494, -1.54902418, -1.61374021,
0.92, 3.98666667]
o    #7 = [ 1.64913665e-01,   1.64091833e-01,   6.42475016e-02, -1.64097948e-02,   3.78519512e-02,
7.86141201e-03, -6.01378346e-02,  -7.60043306e-01,   1.48358422, 0.00000000,   8.75637394]
o    #8 = [ 0.38635521,  0.37239928,  0.19622876,  0.20280844,  0.30099201, 0.20801248,
0.08363992,  1.3157145 , -0.6740433 ,  0,  8.75]
o    #9 = [-0.78208234, -0.77100646, -0.67556357, -0.83952259, -0.80137385, -0.81109068, -0.82477139,
1.3157145 , -0.6740433 ,  0, 8.57268722]
o    #10 = [ 1.54839087,   1.52175829,   1.4957016 ,   2.27758882, 2.40005734,   2.22714833,
1.96839755,   0.75469888, -0.12005785,   0.01351351,  11.62837838]
o    #11 = [ 8.32151937e-01,   8.35581588e-01,   6.92463637e-01, 0.28576901e-01,   8.73325319e-01,
8.28998745e-01, 7.36715766e-01,   1.30421446,  -6.74043296e-01, 5.54016620e-03,   1.05207756e+01]
o    #12 = [ 3.30492209e-01,   3.05336481e-01,   1.28559424e-01, 7.95469133e-03,   5.45129226e-02,
-3.47181835e-02, 5.62456557e-02,  -7.60043306e-01,  -6.74043296e-01, 1.00000000,   9.88068182]
o    #13 = [ 0.9542143 ,   0.94075241,   0.84302087,   0.99420456, 1.0011443 ,   1.03434411,
0.90228008,  -0.76004331, 1.48358422,   0,  10.51948052]
o    #14 = [-1.92413023, -1.93457422, -1.60562154, -1.43248801, -1.35820799, -1.3916672 , -1.4304473
0.85106383, 5.70744681]
o    #15 = [  0.59317092,   0.6655414 ,   0.71660647,   0.6304139 , 0.33240402,   0.59287655,
0.85092328,  -0.76004331, 1.44573111,   0.01754386,  14.75438596]
o    #16 = [-1.13820879, -1.15261884, -1.02782757, -1.12043433, -1.05810267, -1.10103614, -1.1076287
0.9009901, 7.42574257]
     WCSS = 11123.606474
     Cluster # 1
o    Mean = [0.41460894884149113, 0.44571074603581057, 0.50242684474117982, 0.38789494808163699, 0.1
o    STDEV = [0.49017395282115328, 0.48819933814941652, 0.57664161759001087, 0.59006876612365655, 0.5
     Cluster # 2
o    Mean = [-0.38768849586313159, -0.36186810383061796, -0.24570022233801983, -0.54816482161979341,
o    STDEV = [0.44679270902362062, 0.44445119959943596, 0.48016270319991, 0.35866116365941492, 0.337
     Cluster # 3
o    Mean = [0.7115232389694397, 0.75887438411207409, 0.88976445413357519, 0.91650723678993606, 0.48
o    STDEV = [0.5960970842797968, 0.61667943520515767, 0.61515801001876846, 0.91090481507782284, 0.80
     Cluster # 4
o    Mean = [-0.29271597041823688, -0.33191978683611773, -0.39935204353454007, -0.5891793161127129, -
o    STDEV = [0.31606930379676812, 0.31261561144740851, 0.34030152198165609, 0.23701313967605947, 0.2
     Cluster # 5
o    Mean = [-1.0066979539298075, -1.0267393083107659, -0.93025531475218581, -1.0302743107160455, -0
o    STDEV = [0.36293869946323526, 0.34448968518221479, 0.3353402436743001, 0.22818532252424281, 0.26
     Cluster # 6
o    Mean = [-2.6735816918446837, -2.6159113059308154, -2.1506055968051045, -1.5948657751767008, -1.5
o    STDEV = [0.38225584560156262, 0.34676684350270953, 0.37785133517605468, 0.05982942698268161 5, 0
     Cluster # 7
o    Mean = [0.16491366497167484, 0.16409183301417266, 0.064247501601336424, -0.016409794780048306, 0
```

```
o    STDEV = [0.4512921775905992, 0.43935702169694968, 0.48100264724692182, 0.4551476100 6243359, 0.52
     Cluster # 8
o    Mean = [0.38635521335042006, 0.37239927857855232, 0.19622876269934561, 0.20280843710434499, 0.30
o    STDEV = [0.3292585766063566, 0.33170174863099017, 0.36943155552264523, 0.33142008925575578, 0.43
     Cluster # 9
o    Mean = [-0.78208233509643466, -0.77100645768357501, -0.67556356969351727, -0.83952258670156732,
o    STDEV = [0.50444409015951652, 0.50447722176132914, 0.43754509421341198, 0.32144718293077296, 0.3
     Cluster # 10
o    Mean = [1.5483908746346848, 1.5217582852863862, 1.495701598599223, 2.2775888194426241, 2.4000573
o    STDEV = [0.29150673145831396, 0.30059089715340848, 0.72838195373982517, 0.57638221137061485, 0.8
     Cluster # 11
o    Mean = [0.8321519371002809, 0.83558158813605354, 0.69246363671211852, 0.82857690088394187, 0.873
o    STDEV = [0.30169020515770251, 0.30286809187055364, 0.38920714409598717, 0.42716913009109575, 0.5
     Cluster # 12
o    Mean = [0.33049220908126103, 0.30533648089728954, 0.12855942395626405, 0.0079546913272099916, 0
o    STDEV = [0.30782169984036706, 0.30160392040059064, 0.34822714533369492, 0.32020652110376413, 0.3
     Cluster # 13
o    Mean = [0.95421430324856327, 0.94075241264777043, 0.8430208749589192, 0.99420456244720989, 1.001
o    STDEV = [0.33246099804531398, 0.34442406587565988, 1.231313066791041, 0.47133174308659376, 0.558
     Cluster # 14
o    Mean = [-1.9241302281366228, -1.9345742158470067, -1.6056215445545938, -1.43248801231297, -1.358
o    STDEV = [0.29500440087086816, 0.27626271606438585, 0.2797951229754434, 0.089466582249638521, 0.2
     Cluster # 15
o    Mean = [0.593170916903205, 0.66554139877374185, 0.7166064734504326, 0.63041390025761213, 0.3324
o    STDEV = [0.54172483224398027, 0.55292784489492242, 0.64865909821430723, 0.70095236830864027, 0.6
     Cluster # 16
o    Mean = [-1.138208794019224, -1.1526188393957206, -1.0278275672049784, -1.1204343271015913, -1.05
o    STDEV = [0.37299073169735247, 0.36578778773167742, 0.35451474080899126, 0.18664288877728741, 0.2
     RMSE = 2.235387432984
```

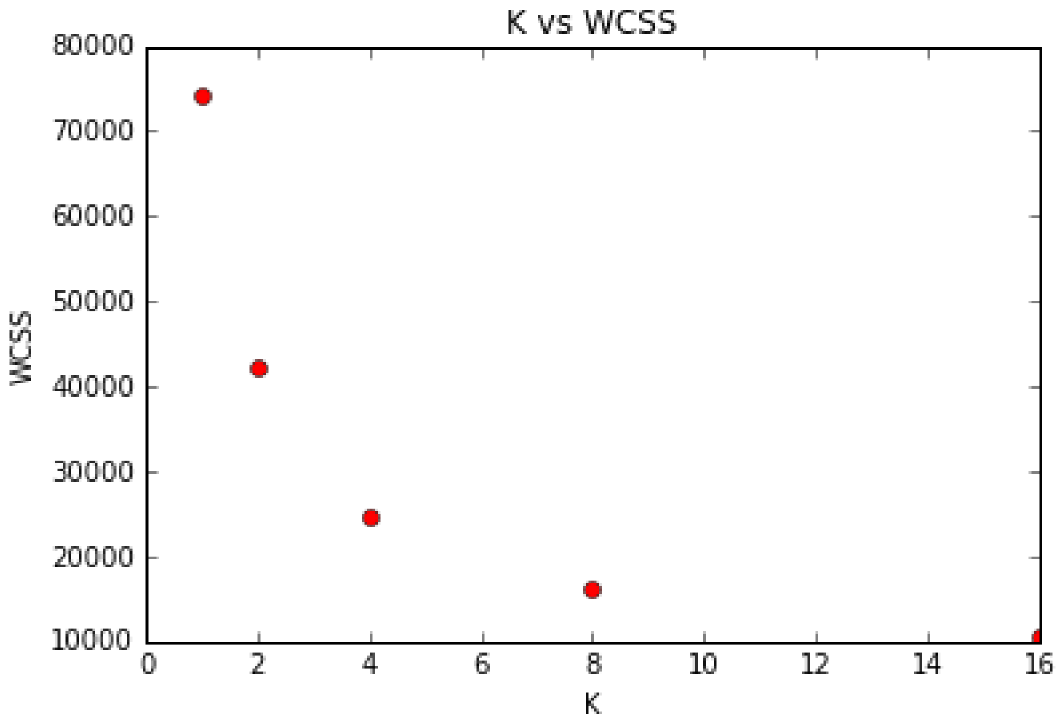

Figure 1: K vs. Within Cluster Sum of Squares

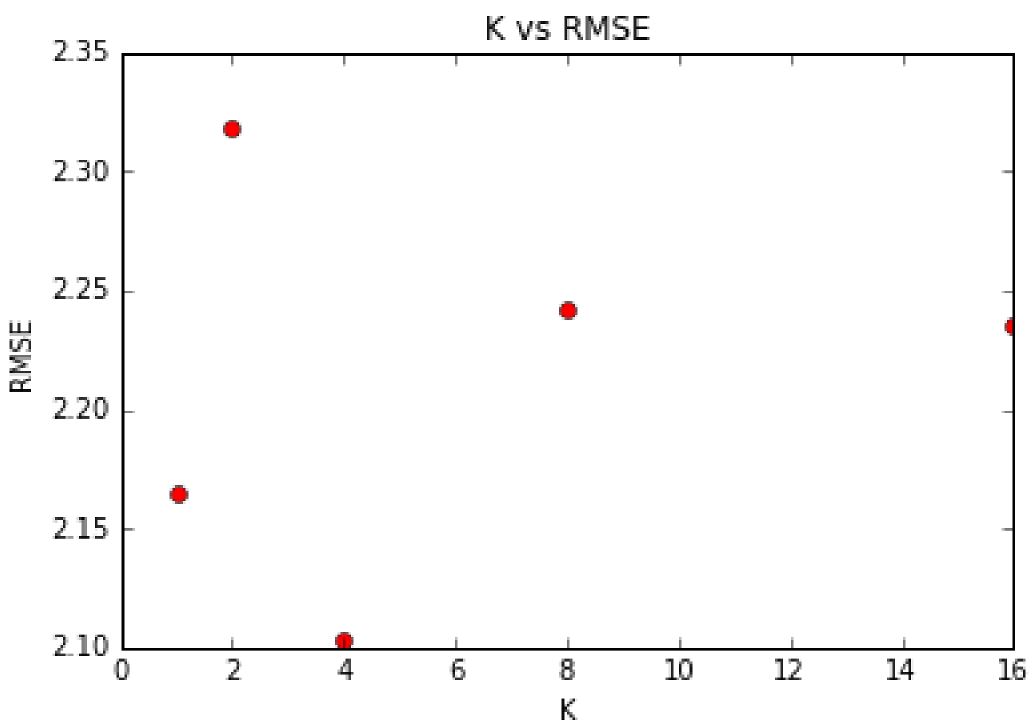

Figure 2: K vs. RMSE