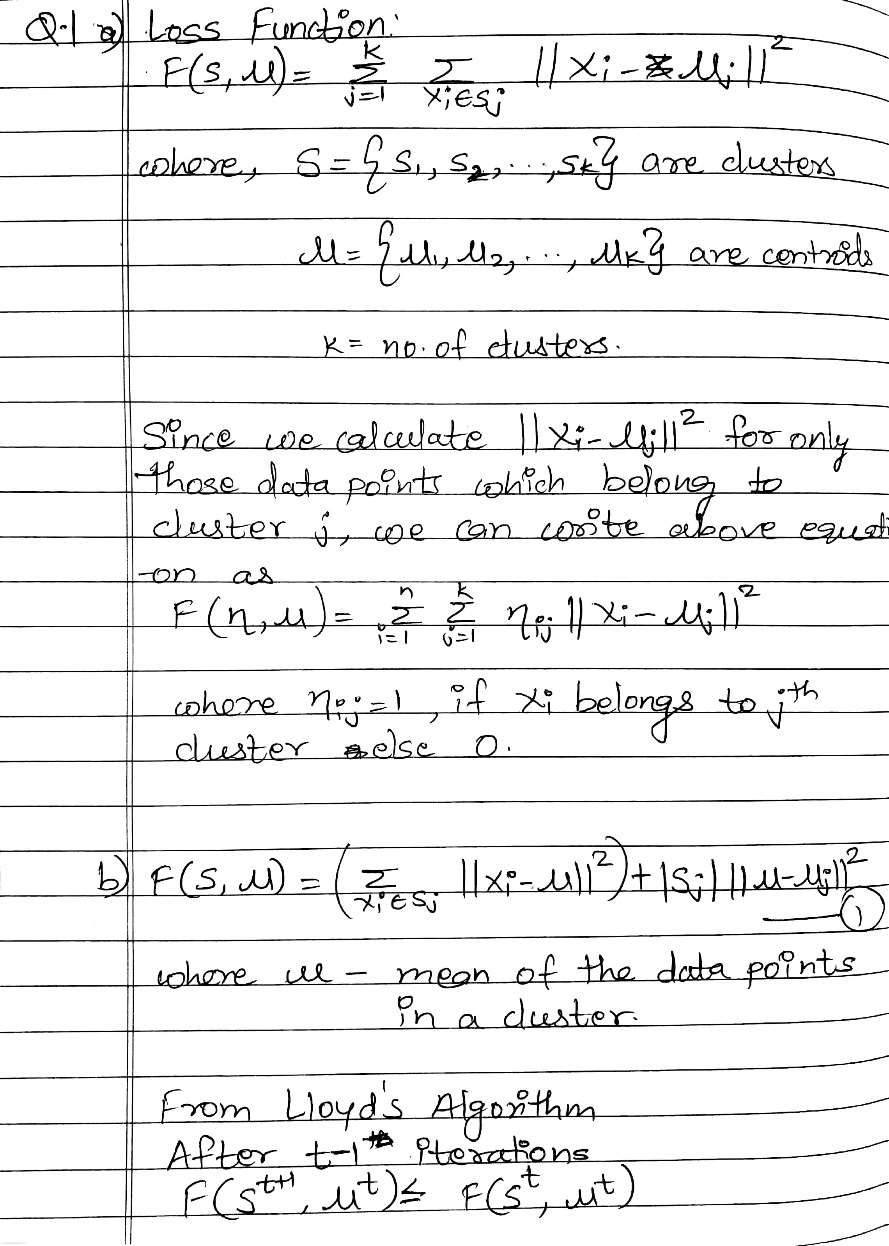
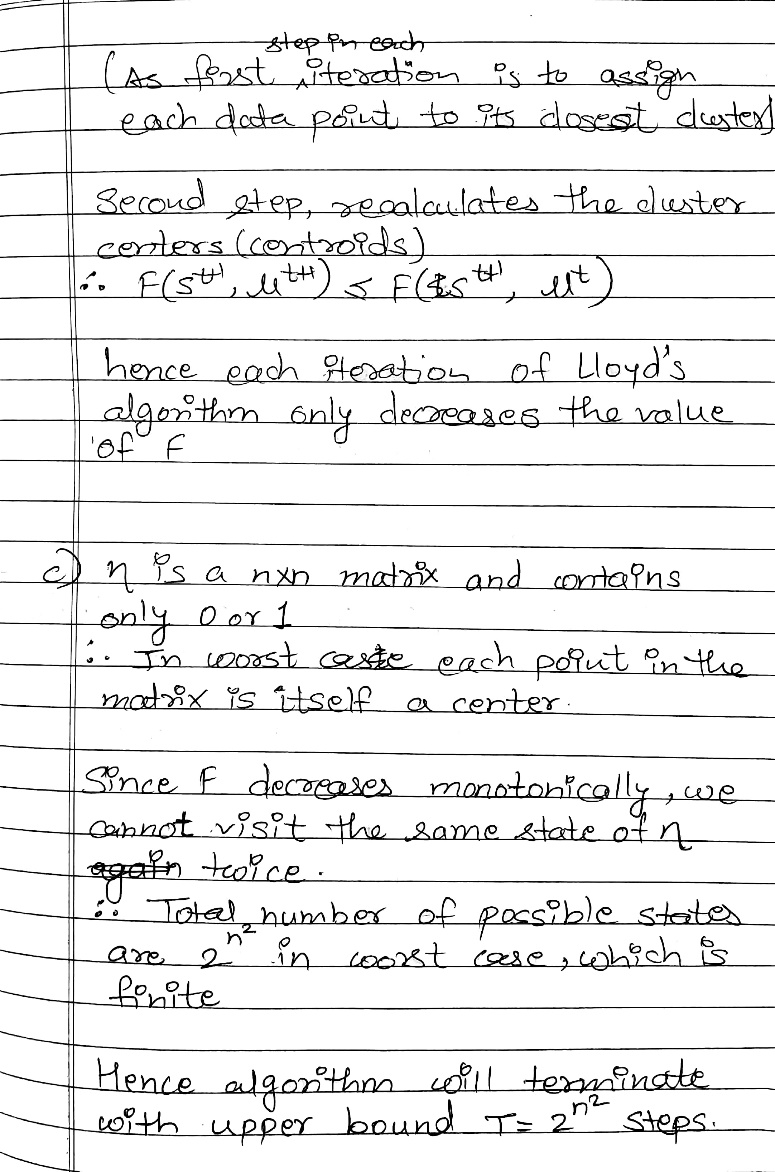
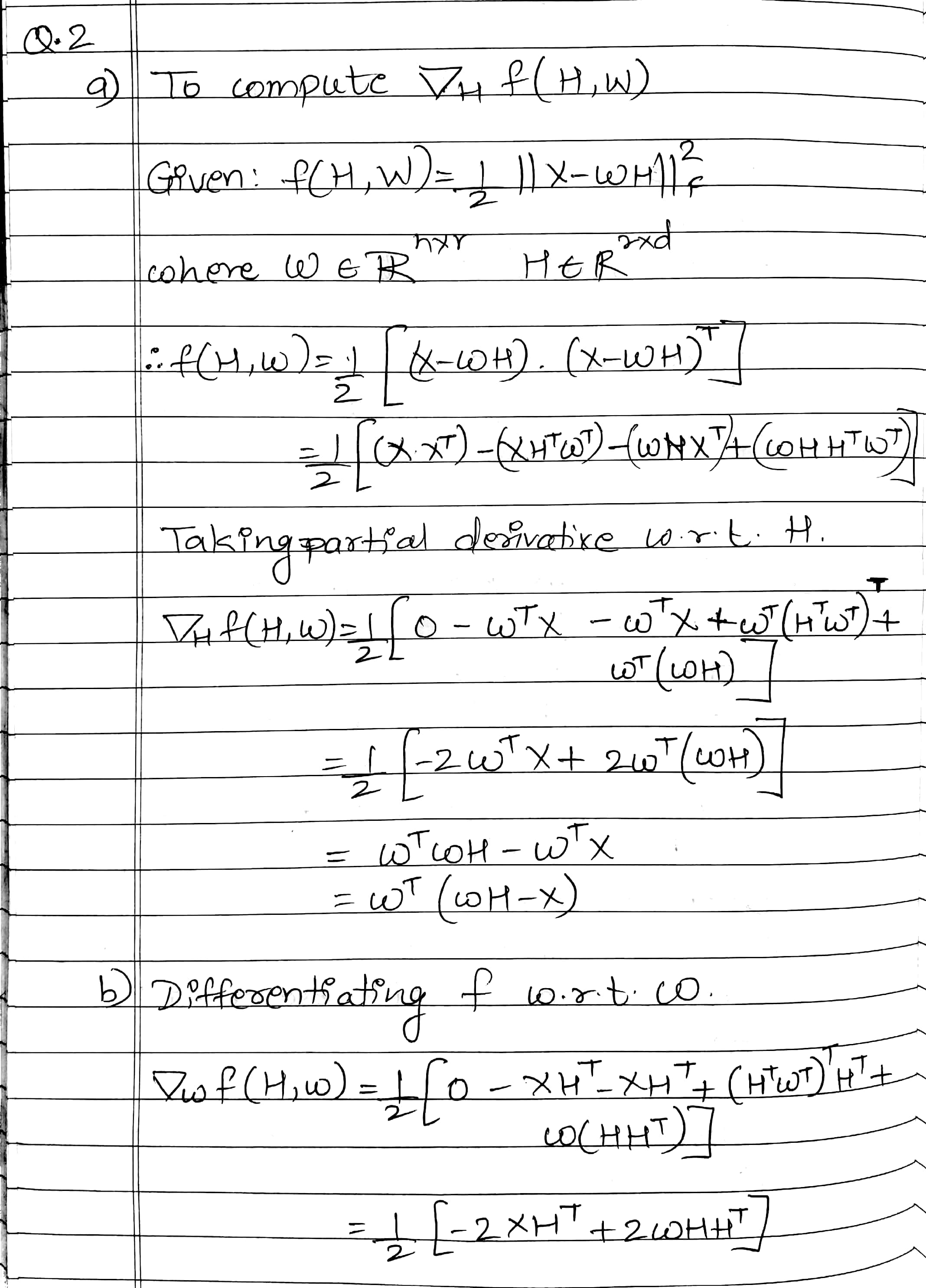
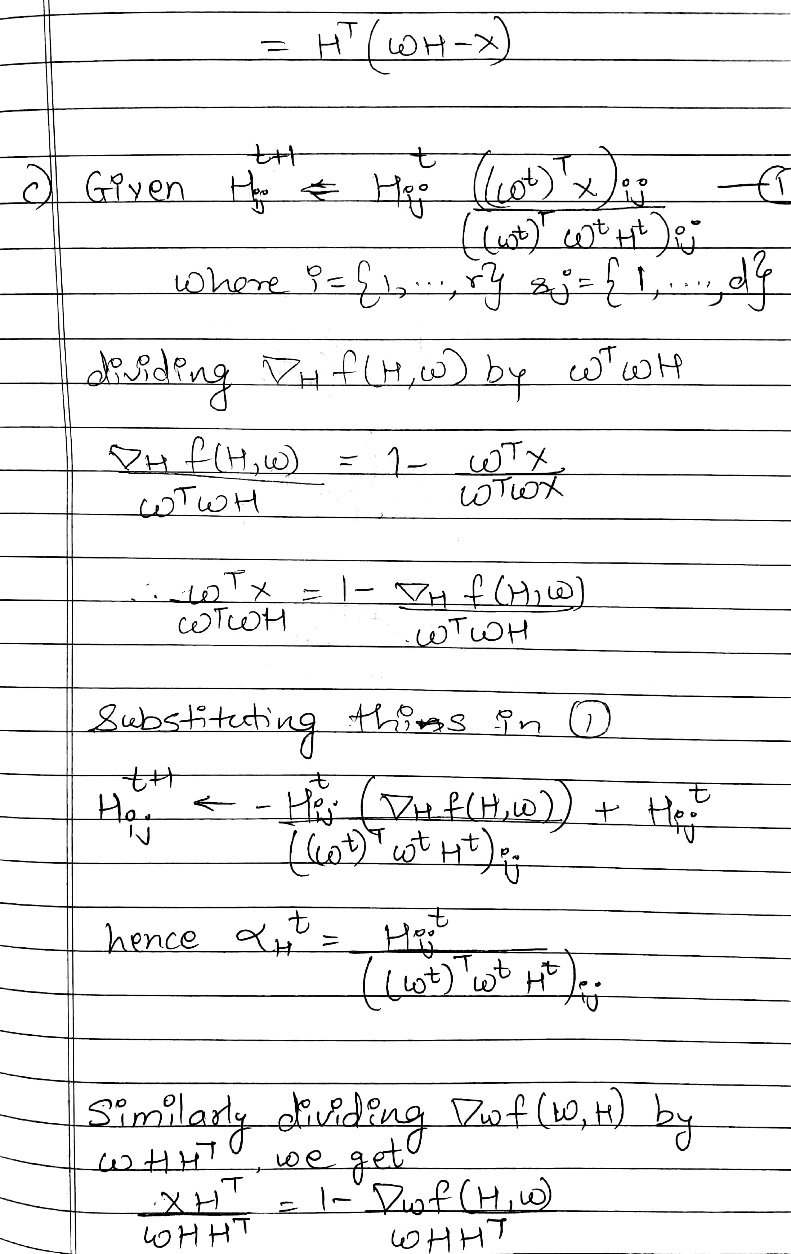
**EE 525 – Assignment 4**

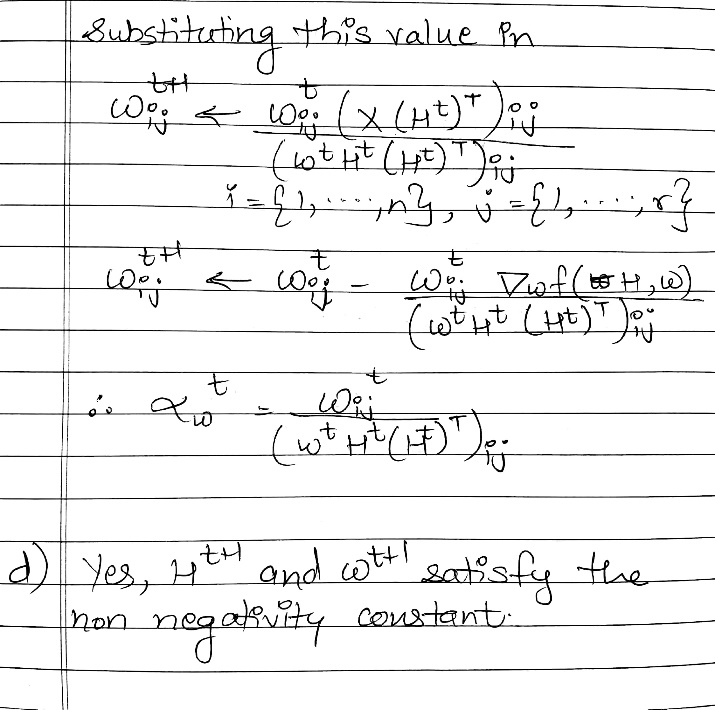
Tanmay Gore – tgore03@iastate.edu

1. **K Means:**

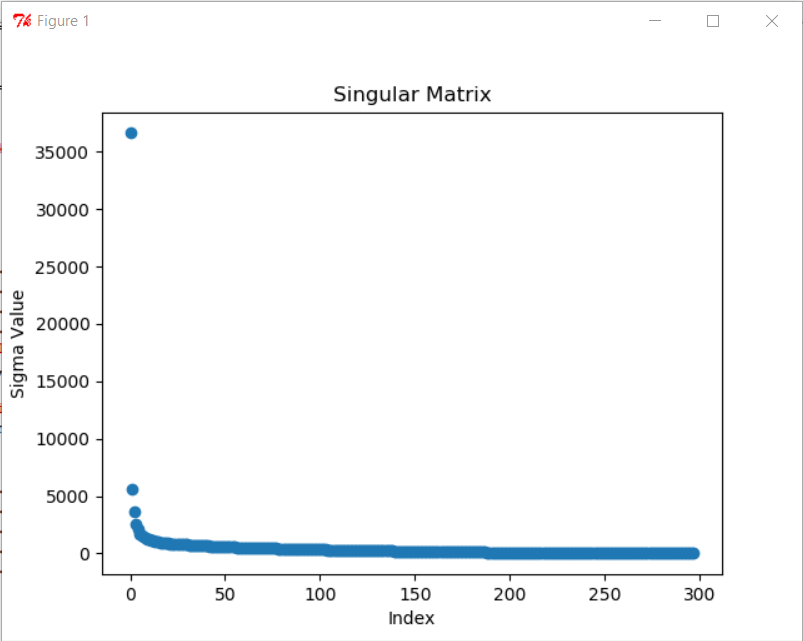
 

1. **Nonnegative Matrix Factorization**

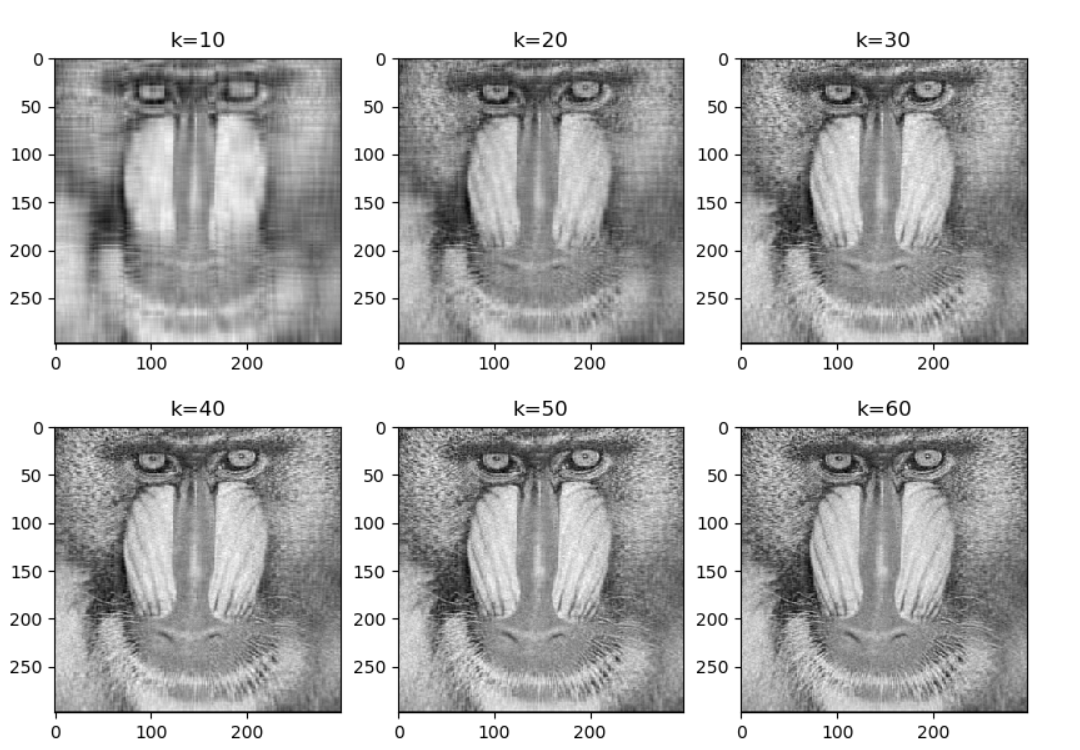
 



1. **SVD Image Compression:**
   1. **Plot Singular Matrix:**



* 1. **Approximation of X using SVD:**



* 1. **Compression Ratio**

k Compression Ratio

10 0.0335570469799

20 0.0671140939597

30 0.10067114094

40 0.134228187919

50 0.167785234899

60 0.201342281879

As seen above images after k = 40 does provide significant clarity in the image however the storage space required to store them keeps on increasing. Therefore storing images with k >40 does not provide any benefits.

* 1. **Code**

import numpy as np

import matplotlib.pyplot as plt

import math

#Read the Image file

from PIL import Image

im = Image.open("mandrill-grayscale.jpg") #Can be many different formats.

pix = im.load()

l,h= im.size

x = np.empty((l,h), dtype=float)

#Take Average of the pixels and save the matrix

for i in range(l):

for j in range(h):

val = im.getpixel((i,j))

x[j][i] = sum(val)/3

plt.figure(1)

plt.imshow(x, cmap='gray')

plt.title("Original Image in gray scale")

plt.show()

#Calculate the SVD of x

u,s,v = np.linalg.svd(x)

#Print singular values

s\_index = np.empty(s.size, dtype=int)

for i in range(s.size):

s\_index[i] = i

plt.scatter(s\_index, s)

plt.xlabel("Index")

plt.ylabel("Sigma Value")

plt.title("Singular Matrix")

plt.show()

def reconstruct(k):

uk = u[:,:k]

sk = np.diag(s[:k])

vk = v[:k,:]

no\_stored = uk.size + k + vk.size

#print "Numbers stored for k =",k,":", no\_stored

compression\_ratio = float(no\_stored)/(u.size+v.size+s.size)

#print "Compression Ratio =", compression\_ratio\*100,"%\n"

#Reconstruct image

t = np.dot(uk,sk)

return np.dot(t,vk), compression\_ratio

#Reconstruct x

plt.figure(2)

plt.tight\_layout()

print "k Compression Ratio"

plt.subplot(2, 3, 1)

plt.title("k=10")

nx, ratio = reconstruct(10)

plt.imshow(nx, cmap='gray')

print "10 ", ratio

plt.subplot(2, 3, 2)

plt.title("k=20")

nx, ratio = reconstruct(20)

plt.imshow(nx, cmap='gray')

print "20 ", ratio

plt.subplot(2, 3, 3)

plt.title("k=30")

nx, ratio = reconstruct(30)

plt.imshow(nx, cmap='gray')

print "30 ", ratio

plt.subplot(2, 3, 4)

plt.title("k=40")

nx, ratio = reconstruct(40)

plt.imshow(nx, cmap='gray')

print "40 ", ratio

plt.subplot(2, 3, 5)

plt.title("k=50")

nx, ratio = reconstruct(50)

plt.imshow(nx, cmap='gray')

print "50 ", ratio

plt.subplot(2, 3, 6)

plt.title("k=60")

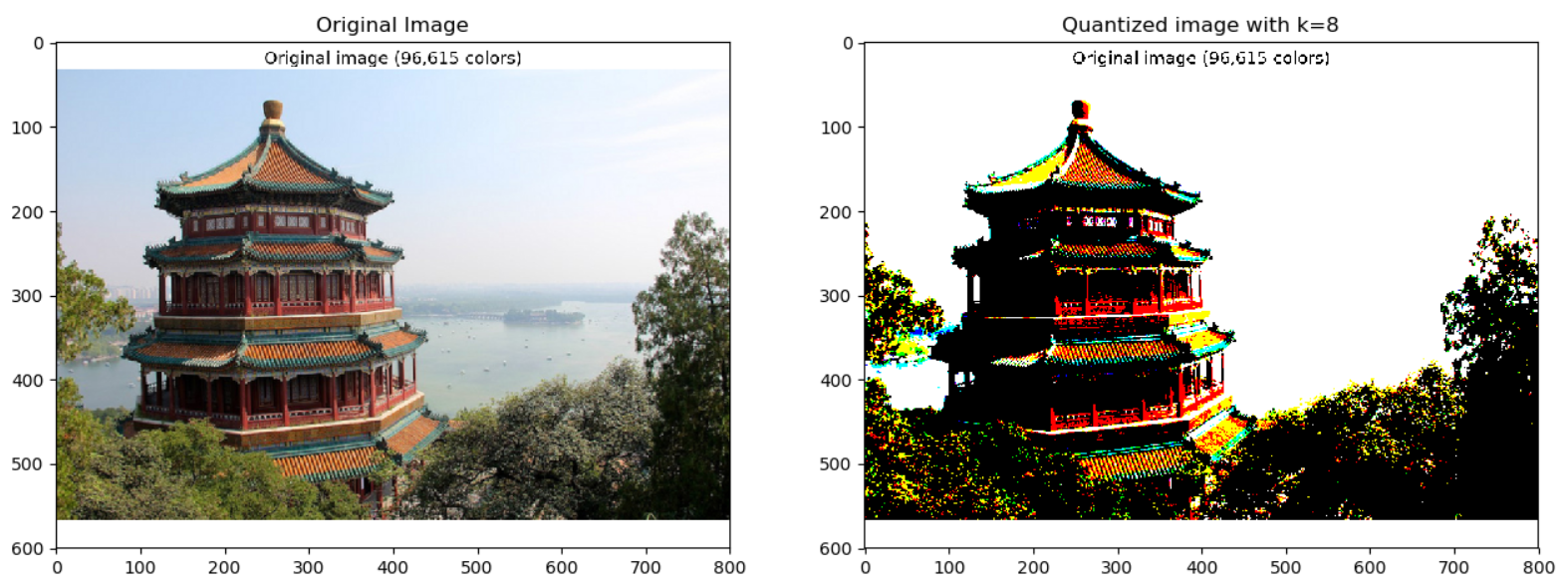
nx, ratio = reconstruct(60)

plt.imshow(nx, cmap='gray')

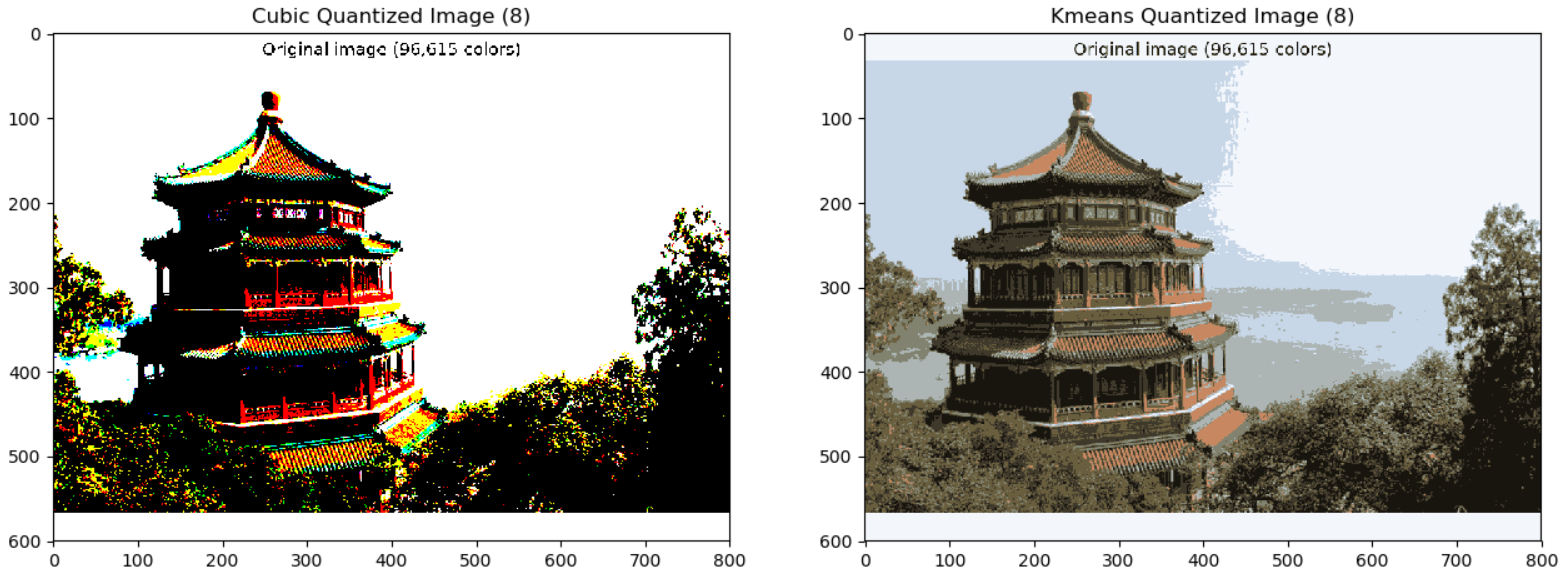
print "60 ", ratio

plt.show()

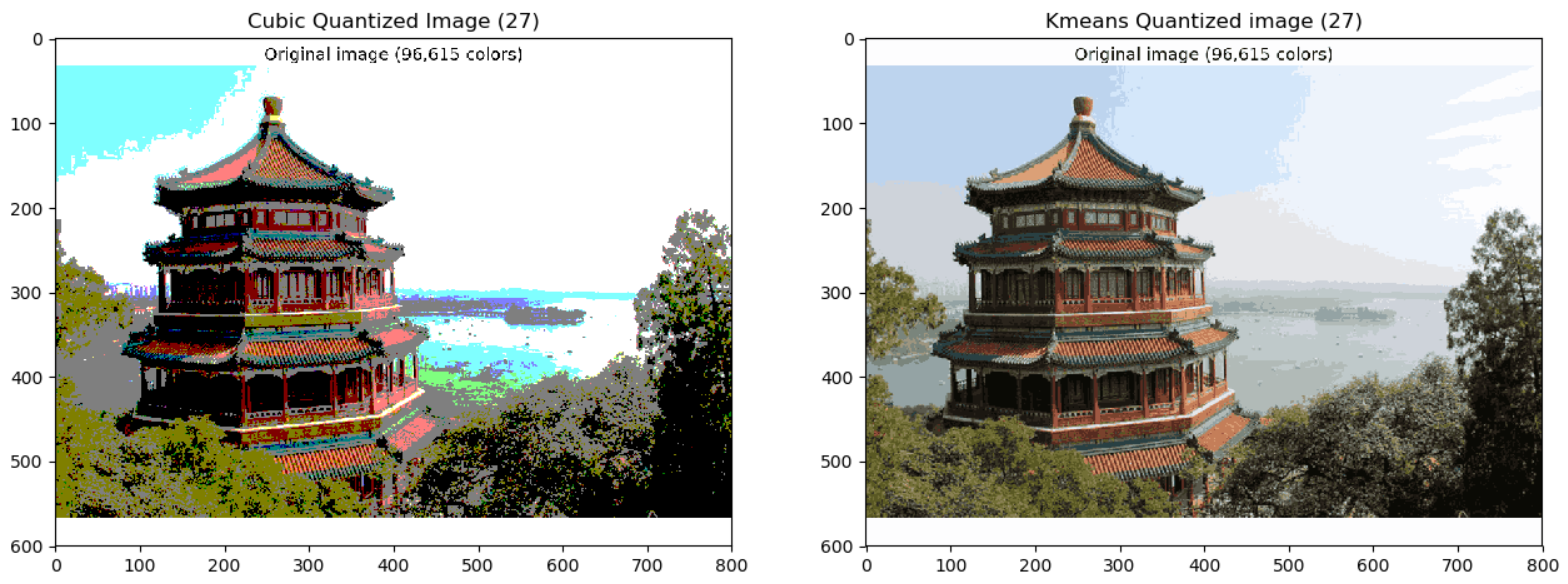
1. **Cubic vs KMeans Quantization:**
   1. **Cubic Quantization for k = 8**

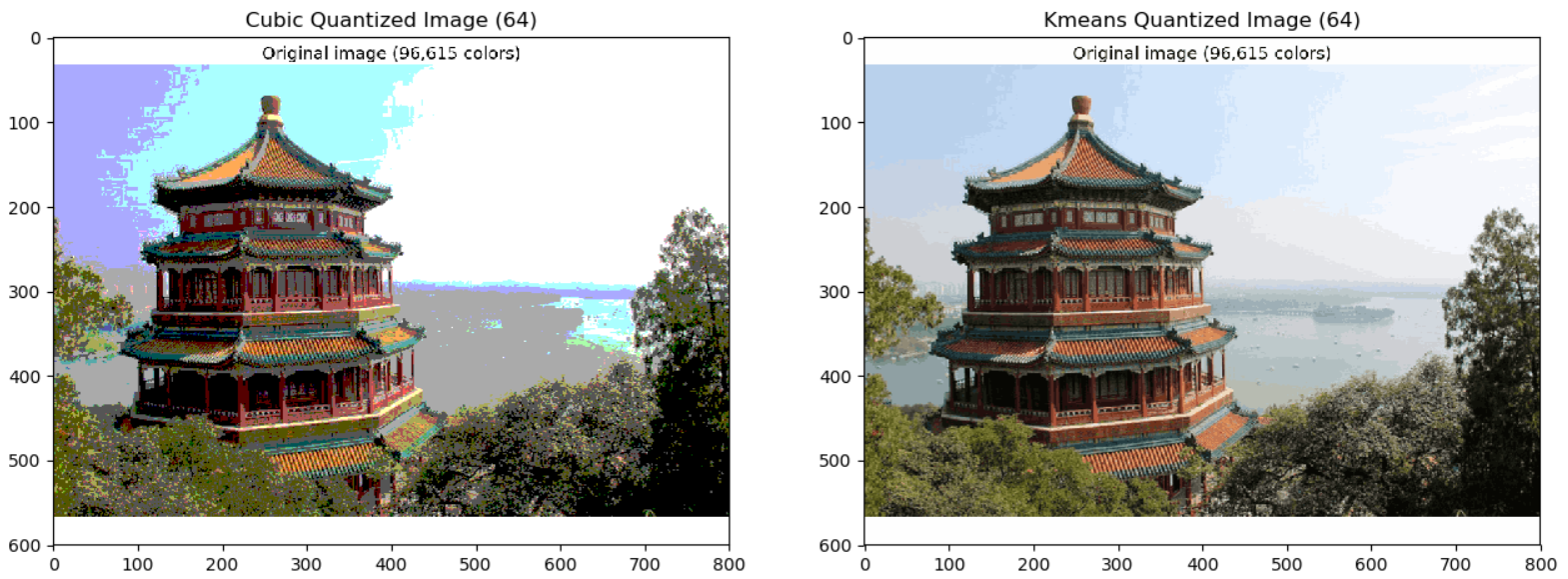
****

* 1. **Kmeans Quantization for k = 8**

****

* 1. **For k = 27 & 64**





**Code:**

import numpy as np

import matplotlib.pyplot as plt

import itertools

from scipy.spatial import distance

from time import time

from sklearn.cluster import KMeans

from PIL import Image

def quantization(color\_set, i, j):

min\_dst = np.inf

min\_lst = []

for lst in color\_set:

dst = distance.euclidean(np.array(lst),x[i,j])

if dst < min\_dst:

min\_dst = dst

min\_lst = lst

return np.array(min\_lst)

def cubic\_quantization(cset):

#generate color representative set

color\_set = list(itertools.product(cset, repeat=3))

x\_quantized = np.empty\_like(x)

x\_shape = x.shape

for i in range(x\_shape[0]):

for j in range(x\_shape[1]):

x\_quantized[i,j] = quantization(color\_set, i, j)

return x\_quantized

#K-Means

def kmeans\_quantization(k):

#Train K-Means and predict clusters

kmeans=KMeans(n\_clusters=k)

clusters = kmeans.fit\_predict(nx)

#Reconstruct image

x\_centroid = np.empty\_like(x)

idx = 0

for i in range(x.shape[0]):

for j in range(x.shape[1]):

x\_centroid[i,j] = kmeans.cluster\_centers\_[clusters[idx]]

idx+=1

return x\_centroid

def plot\_images(k, x\_cubic, x\_kmeans):

#Plot Kmeans vs cubic quantized image

plt.figure(k)

plt.subplot(1, 2, 1)

plt.imshow(x\_cubic)

plt.title("Cubic Quantized Image ("+str(k)+")")

plt.subplot(1,2,2)

plt.title("Kmeans Quantized Image ("+str(k)+")")

plt.imshow(x\_kmeans)

plt.show()

#Read image

im = Image.open("palace.png")

im.load()

x = np.array(im)

#Reshape x

nx = np.reshape(x, (x.shape[0]\*x.shape[1], x.shape[2]))

#Perform cubic quantization on k=8

t0 = time()

x\_cubic = cubic\_quantization([0,255])

print "Cubic Quantizatoin: time required =", time() - t0,"secs\n"

#Plot quantized vs original image

plt.figure(1)

plt.subplot(1, 2, 1)

plt.imshow(x)

plt.title("Original Image")

plt.subplot(1,2,2)

plt.title("Quantized image with k=8")

plt.imshow(x\_cubic)

plt.show()

#Perform Kmeans quantization on k=8

t0 = time()

x\_kmeans = kmeans\_quantization(8)

print "Kmeans Quantization: time required =", time() - t0,"secs\n"

#Plot Kmeans vs cubic quantized image

plot\_images(8, x\_cubic, x\_kmeans)

#Perform cubic quantization on k=27

t0 = time()

x\_cubic = cubic\_quantization([0,127,255])

print "Cubic Quantizatoin: time required =", time() - t0,"secs\n"

#Perform Kmeans quantization on k=27

t0 = time()

x\_kmeans = kmeans\_quantization(27)

print "Kmeans Quantization: time required =", time() - t0,"secs\n"

#Plot Kmeans vs cubic quantized image

plot\_images(27, x\_cubic, x\_kmeans)

#Perform cubic quantization on k=64

t0 = time()

x\_cubic = cubic\_quantization([0,85,170,255])

print "Cubic Quantizatoin: time required =", time() - t0,"secs\n"

#Perform Kmeans quantization on k=64

t0 = time()

x\_kmeans = kmeans\_quantization(64)

print "Kmeans Quantization: time required =", time() - t0,"secs\n"

#Plot Kmeans vs cubic quantized image

plot\_images(64, x\_cubic, x\_kmeans)

**References:**

1. Discussed with Nitesh Gupta however all solutions were compiled and written by me.