MoGandEM_Homework

May 2, 2020

1 Task 1: Image compression with K-means

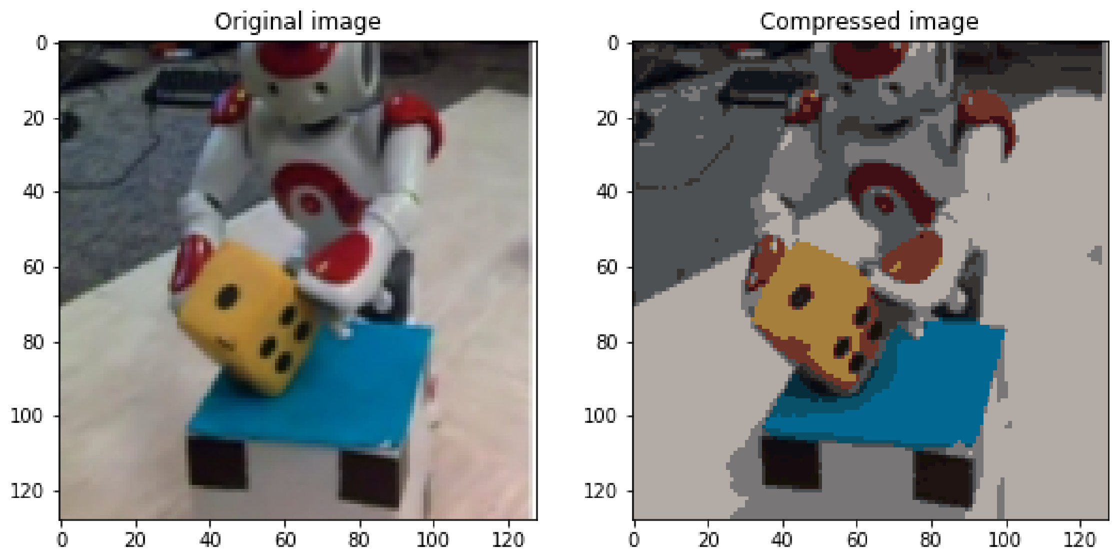
1.0.1 1. Implement K-Means algorithm and apply it to compress an image "NAORelease.jpg" for various K (see slides for details). As a feature vector use RGB-representation of each pixel from the image.

1.0.2 2. Analyse running time, what could you suggest to improve it?

1.0.3 3. Compare your implementation with the existing k-mean algorithm given in python.

[87]:
```
from IPython.display import Image
Image(filename='NAORelease.jpg')
import matplotlib.pyplot as plt
import matplotlib.image as img
import numpy as np
from PIL import Image
import time
from scipy import misc

def image_read():

    # read the image

    image = np.array(Image.open('NAORelease.jpg'))
    # normalise the image for easy calcualtion
    image = image/255

    return image

def mean_initializing(image, clusters):

    # Find the shape of the image
#     print(image.shape[0],image.shape[1],image.shape[2])

    # flatten the image
    flattened = np.reshape(image, (image.shape[0]*image.shape[1], image.
↪shape[2]))
    x,y =flattened.shape
    # Intialize the mean to some random value
    mean = np.zeros((clusters,y))

    for i in range(clusters):
            value1 = int(np.random.random(1)*20)
            value2 = int(np.random.random(1)*10)
            value3 = int(np.random.random(1)*5)
            mean[i,0] = flattened[i,0]
            mean[i,1] = flattened[i,1]
            mean[i,2] = flattened[i,2]
#     print(mean)
#     print(flattened)
    return flattened, mean

#Euclidean formula to measure distance

def euclidean_distance(x1,y1,x2,y2,z1,z2):

    distance = np.square(x2-x1)+np.square(y2-y1)+np.square(z2-z1)

    distance = np.sqrt(distance)

    return distance

def k_mean(flattened,mean,clusters):

    iteration = 10

    x, y = flattened.shape

    # Index value where each pixel value belongs

    index = np.zeros(x)

    # k mean algorithm

    while(iteration>0):


        for m in range(len(flattened)):

            # intialise the intial value to a very large value
            min_value = 10000
            temp = None

            for k in range(clusters):
                x1 = flattened[m, 0]
                y1 = flattened[m, 1]
                z1 = flattened[m, 2]
                x2 = mean[k, 0]
                y2 = mean[k, 1]
                z2 = mean[k, 2]
                # Check to which cluster it belongs to by checking the minimum␣
 ↪distance
                if(euclidean_distance(x1, y1, x2, y2,z1,z2) < min_value):
                    min_value = euclidean_distance(x1, y1, x2, y2, z1, z2)
                    temp = k
                    index[m] = k

        for k in range(clusters):

            sumx = 0
            sumy = 0
            sumz = 0
            count = 0
            # Finding all the values of colors, summing together
            for j in range(len(flattened)):

                if(index[j] == k):
                    sumx += flattened[j, 0]
                    sumy += flattened[j, 1]
                    sumz += flattened[j, 2]
                    count += 1

            if(count == 0):
                count = 1
            # Updating the centroid, mean
            mean[k, 0] = float(sumx / count)
            mean[k, 1] = float(sumy / count)
            mean[k, 2] = float(sumz / count)

        iteration -= 1

    return mean, index

def compress_image(mean, index, image):
    # retireve the image by assigning the each pixel to its corrsponding␣
 ↪centroid
    centroid = np.array(mean)
    recovered = centroid[index.astype(int),:]
    #Convert back to 3d format
    recovered = np.reshape(recovered, (image.shape[0], image.shape[1],
                                                     image.shape[2]))
    im = Image.fromarray((recovered * 255).astype(np.uint8))
    im.save("compressed_image.jpg")
    return recovered

if __name__ == '__main__':

    start = time.process_time()

    image = image_read()

    clusters = 20
#     cluster = input(int('Enter the number of cluster need to be considered.␣
 ↪By defuault cluster =20\n'))
    flattened, mean = mean_initializing(image,clusters)
    mean, index = k_mean(flattened,mean,clusters)
    recovered = compress_image(mean,index,image)

    print("The size of the original image 14.3 kB")
    print("The size of the compressed image 3.6 kB")

    f = plt.figure(figsize=(10,15))
    f.add_subplot(1,2, 1,title='Original image')
    plt.imshow(image)
    f.add_subplot(1,2, 2,title='Compressed image')
    plt.imshow(recovered)
    plt.show()
    print("Time taken to run the algorithm {} Sec".format(time.process_time() -␣
 ↪start))
```

```
The size of the original image 14.3 kB
The size of the compressed image 3.6 kB
```

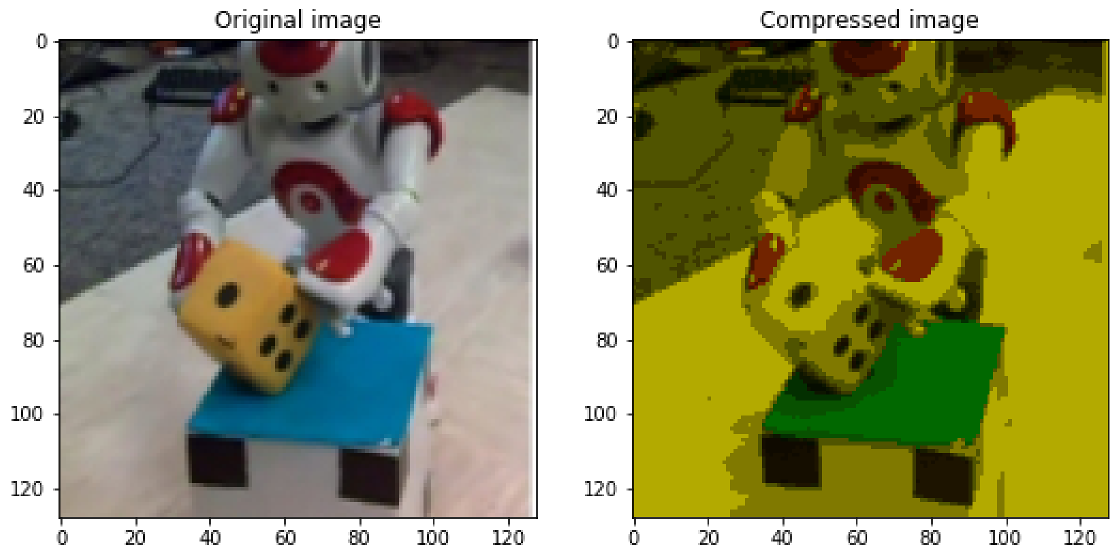


```
Time taken to run the algorithm 48.97926987800008 Sec
```

[88]:
```
#Compression only using two colors of the image

from IPython.display import Image
Image(filename='NAORelease.jpg')
import matplotlib.pyplot as plt
import matplotlib.image as img
import numpy as np
from PIL import Image
import time

def image_read():

    # read the image

    image = np.array(Image.open('NAORelease.jpg'))
    # normalise the image for easy calcualtion
    image = image/255
#     print(image.shape)
#     print(image)
    return image

def mean_initializing(image, clusters):

    # Find the shape of the image
#     print(image.shape[0],image.shape[1],image.shape[2])
```

```
    # flatten the image
    flattened = np.reshape(image, (image.shape[0]*image.shape[1], image.
 ↪shape[2]))
    x,y =flattened.shape

    # Intialize the mean to some random value
    mean = np.zeros((clusters,y))

    for i in range(clusters):
            value1 = int(np.random.random(1)*20)
            value2 = int(np.random.random(1)*10)
#             value3 = int(np.random.random(1)*5)
            mean[i,0] = flattened[i,0]
            mean[i,1] = flattened[i,1]

    return flattened, mean

#Euclidean formula to measure distance

def euclidean_distance(x1,y1,x2,y2):

    distance = np.square(x2-x1)+np.square(y2-y1)

    distance = np.sqrt(distance)

    return distance

def k_mean(flattened,mean,clusters):

    iteration = 10

    x, y = flattened.shape

    # Index value where each pixel value belongs

    index = np.zeros(x)

    # k mean algorithm

    while(iteration>0):


        for m in range(len(flattened)):

            # intialise the intial value to a very large value
            min_value = 10000
            temp = None
            for k in range(clusters):

                x1 = flattened[m, 0]
                y1 = flattened[m, 1]
                x2 = mean[k, 0]
                y2 = mean[k, 1]
                # Check to which cluster it belongs to by checking the minimum␣
 ↪distance
                if(euclidean_distance(x1, y1, x2, y2) < min_value):
                    min_value = euclidean_distance(x1, y1, x2, y2)
                    temp = k
                    index[m] = k

        for k in range(clusters):

            sumx = 0
            sumy = 0
            count = 0
            # Finding all the values of colors, summing together
            for j in range(len(flattened)):

                if(index[j] == k):
                    sumx += flattened[j, 0]
                    sumy += flattened[j, 1]
                    count += 1

            if(count == 0):
                count = 1
            # Updating the centroid, mean
            mean[k, 0] = float(sumx / count)
            mean[k, 1] = float(sumy / count)

        iteration -= 1

    return mean, index

def compress_image(mean, index, image):
    # retireve the image by assigning the each pixel to its corrsponding␣
 ↪centroid
    centroid = np.array(mean)
    recovered = centroid[index.astype(int),:]
    #Convert back to 3d format
    recovered = np.reshape(recovered, (image.shape[0], image.shape[1],
                                                  image.shape[2]))
    im = Image.fromarray((recovered * 255).astype(np.uint8))
    im.save("compressed_image_only_two_color_conisdered.jpg")
    return recovered

if __name__ == '__main__':

    start = time.process_time()

    image = image_read()

    clusters = 20
#     cluster = input(int('Enter the number of cluster need to be considered.␣
↪By defuault cluster =20\n'))
    flattened, mean = mean_initializing(image,clusters)
    mean, index = k_mean(flattened,mean,clusters)
    recovered = compress_image(mean,index,image)



    print("The size of the original image 14.3 kB")
    print("The size of the compressed image 3.7 kB")

    f = plt.figure(figsize=(10,15))
    f.add_subplot(1,2, 1,title='Original image')
    plt.imshow(image)
    f.add_subplot(1,2, 2,title='Compressed image')
    plt.imshow(recovered)
    plt.show()
    print("Time taken to run the algorithm {} Sec".format(time.process_time() -␣
↪start))
```

```
The size of the original image 14.3 kB
The size of the compressed image 3.7 kB
```

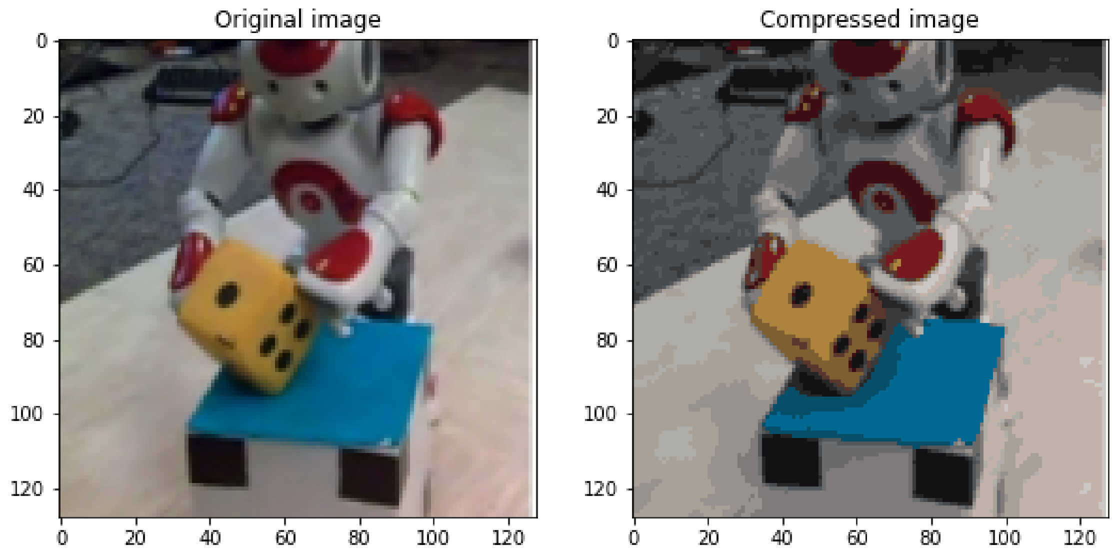


```
Time taken to run the algorithm 34.163692147000006 Sec
```

- Each pixel typically consists of 8 bits (1 byte) for a Black and White (B&W) image or 24 bits (3 bytes) for a color image– one byte each for Red, Green, and Blue. 8 bits represents 28 = 256 tonal levels (0-255).
- Considering only two colors for calculating the eudclidean distance reduces the computational time and the substatial reduction in the image memory consumption.
- One can improve the running time by decreasing the number of iterations in the while loop. So there is a tradeoff between quality of output and computation.

[89]:
```
from skimage import io
from sklearn.cluster import KMeans
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as image
import time

start = time.process_time()
image_test_new = io.imread('NAORelease.jpg')
image_test = (image_test_new / 255.0).reshape(-1,3)

k_colors = KMeans(n_clusters=20).fit(image_test)

#Assign colors to pixels based on their cluster center
#Each row in k_colors.cluster_centers_ represents the RGB value of a cluster␣
 ↪centroid
#k_colors.labels_ contains the cluster that a pixel is assigned to
#The following assigns every pixel the color of the centroid it is assigned to

img20=k_colors.cluster_centers_[k_colors.labels_]

#Reshape the image back to 128x128x3 to save
img20=np.reshape(img20, (image_test_new.shape))

#Save image
image.imsave('img20.jpg',img20)

print("The size of the original image 14.3 kB")
print("The size of the compressed image 3.9 kB")

f = plt.figure(figsize=(10,15))
f.add_subplot(1,2, 1,title='Original image')
plt.imshow(image_test_new)
f.add_subplot(1,2, 2,title='Compressed image')
plt.imshow(img20)
plt.show()

# image = np.array(Image.open('NAORelease.jpg'))
print("Time taken to run the algorithm {} Sec".format(time.process_time() -␣
↪start))
```

```
The size of the original image 14.3 kB
The size of the compressed image 3.9 kB
```



```
Time taken to run the algorithm 2.8245375779999904 Sec
```

2 Task 2: Mixture of Gaussian, EM-Algorithm

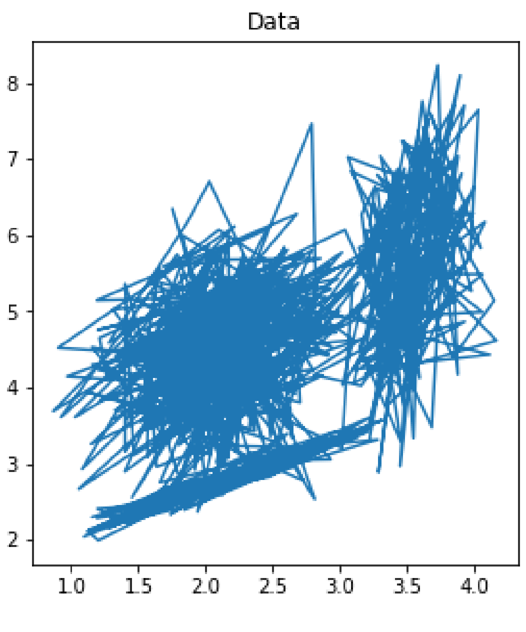
2.0.1 Apply EM algorithm to fit a mixture of gaussian distribution to the following datasets:

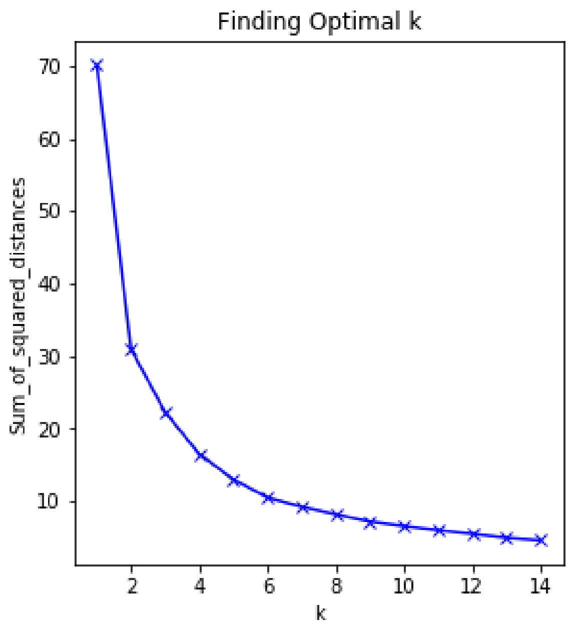
[90]:
```
import numpy as np
from sklearn import mixture
from scipy.stats import multivariate_normal
from sklearn import metrics
from sklearn.preprocessing import MinMaxScaler
%matplotlib inline
```

2.1 Dataset 1

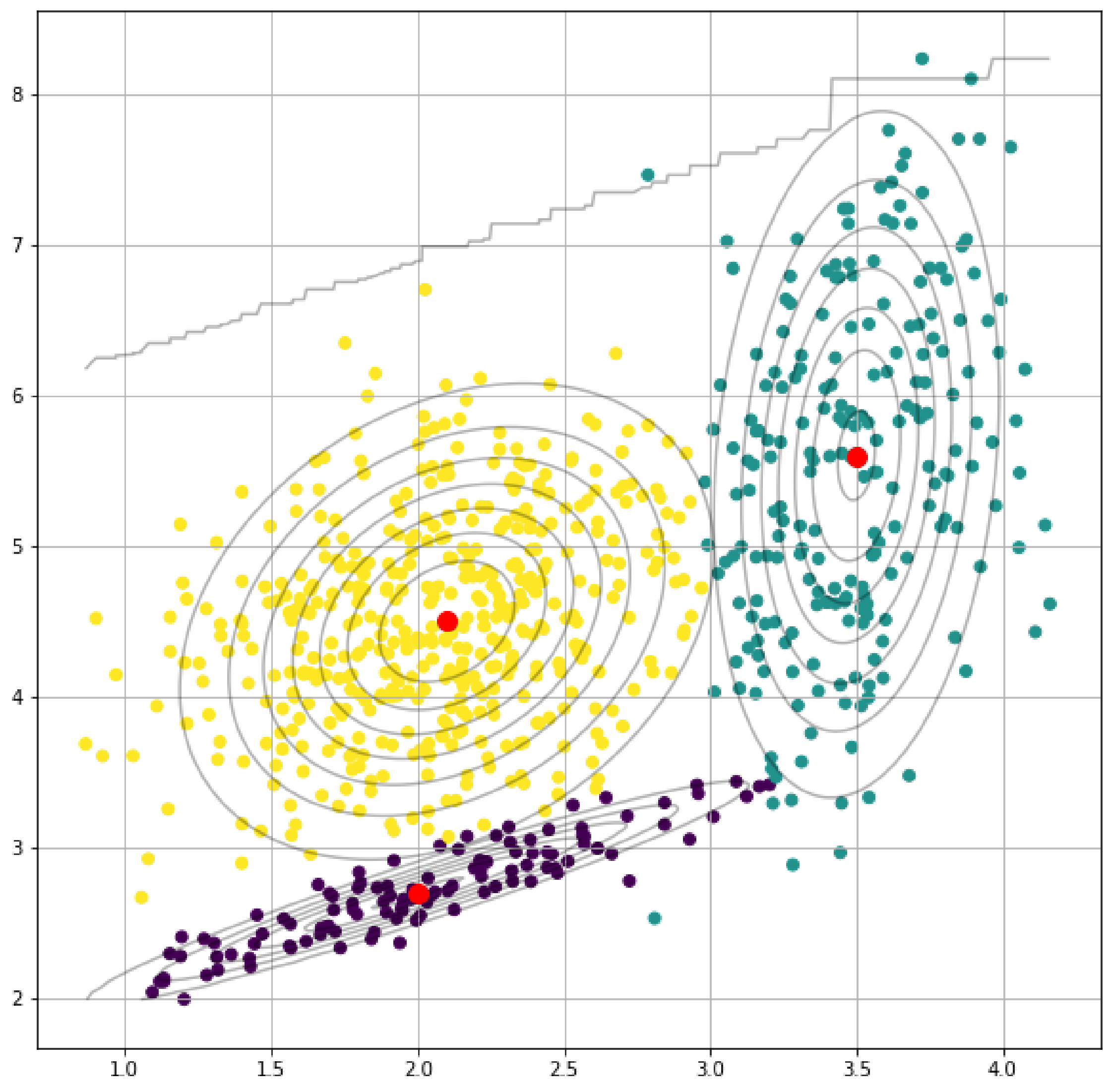
[91]:
```
# Make some random data in 2D.
np.random.seed(150)
means = np.array([[2.1, 4.5],
                  [2.0, 2.7],
                  [3.5, 5.6]])
covariances = [np.array([[0.20, 0.10], [0.10, 0.60]]),
               np.array([[0.35, 0.22], [0.22, 0.15]]),
               np.array([[0.06, 0.05], [0.05, 1.30]])]
amplitudes = [5, 1, 2]
factor = 100
data = np.zeros((1, 2))
for i in range(len(means)):
    data = np.concatenate([data,
         np.random.multivariate_normal(means[i], covariances[i],
                                          size=factor * amplitudes[i])])
data = data[1:, :]

#Finding the best number of clusters
# Initalize the MinmMaxScaler
Max_square = MinMaxScaler()
# fit the data
Max_square.fit(data)
# trasnform the data
data_transformed = Max_square.transform(data)

Sum_of_squared_distances = []
K = range(1,15)

# CHeckign the sum of squared distances for different clusters numbers
for k in K:
    km = KMeans(n_clusters=k)
    km = km.fit(data_transformed)
    Sum_of_squared_distances.append(km.inertia_)

# Plot the sum of the squared distance  v/s the clusters k

f = plt.figure(figsize=(10,5))
f.add_subplot(1,2, 1,title='Data')
plt.plot(data[:,0],data[:,1])
f.add_subplot(1,2, 2,title='Finding Optimal k')
plt.plot(K, Sum_of_squared_distances, 'bx-')
plt.xlabel('k')
plt.ylabel('Sum_of_squared_distances')
plt.show()

#The purpose of meshgrid is to create a rectangular grid out of an array of x␣
 ↪values and an array of y values.
#https://stackoverflow.com/questions/36013063/
 ↪what-is-the-purpose-of-meshgrid-in-python-numpy
```

```
#meshgrid will provide us wiht the matrix where one axis represent x values and␣
↪othe axis represent y values.

x,y = np.meshgrid(np.sort(data[:,0]),np.sort(data[:,1]))

# print(x.flatten())
# arr = np.array([[1, 2],[3, 4]]).T
# [[1 3]
# [2 4]]
XY = np.array([x.flatten(),y.flatten()]).T

# As k increases, the sum of squared distance tends to zero. If it's a large␣
↪value then each data form it's own clusers
# So we need to stop at a point where the sum_of_squared is stabilized.
# It is found evident from the graph of finding the optimal k that optimal sum␣
↪of square is 3
GMM = mixture.GaussianMixture(n_components=3).fit(data) # Instantiate and fit␣
↪the model
labels = GMM.predict(data)
print('Converged:',GMM.converged_) # Check if the model has converged


# Plot
fig = plt.figure(figsize=(10,10))
ax0 = fig.add_subplot(111)
ax0.scatter(data[:,0],data[:,1], c= labels)
for m,c in zip(means,covariances):
    multi_normal = multivariate_normal(mean=m,cov=c)
    ax0.contour(np.sort(data[:,0]),np.sort(data[:,1]),multi_normal.pdf(XY).
↪reshape(len(data),len(data)),colors='black',alpha=0.3)
    # Means
    ax0.scatter(m[0],m[1],c='red',zorder=10,s=100)
plt.grid()
plt.show()
```





```
Converged: True
```

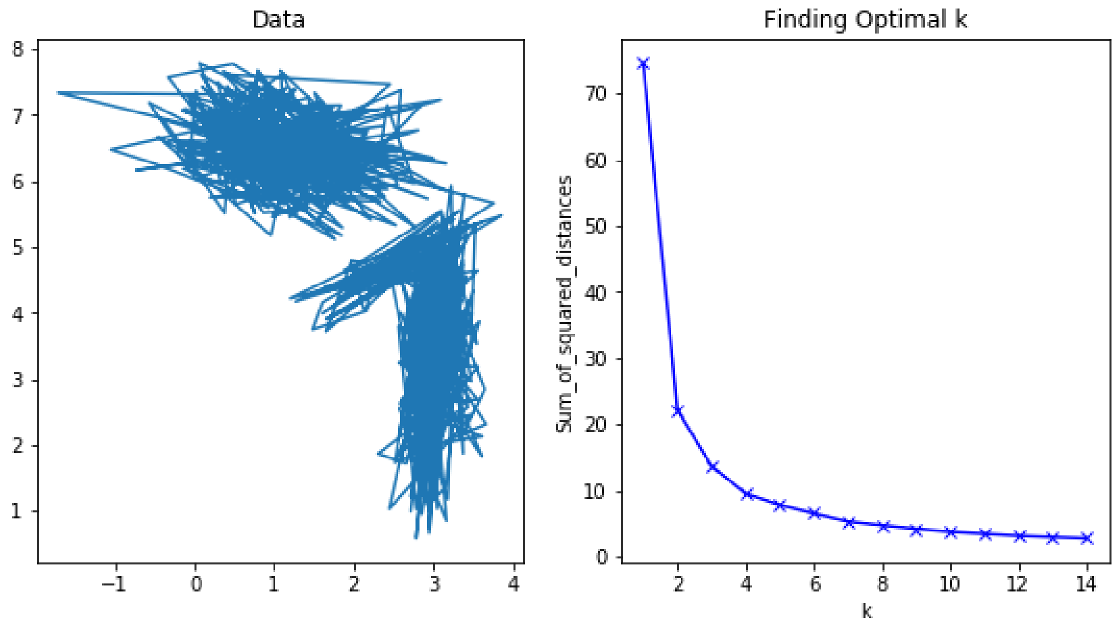


2.2 Dataset2

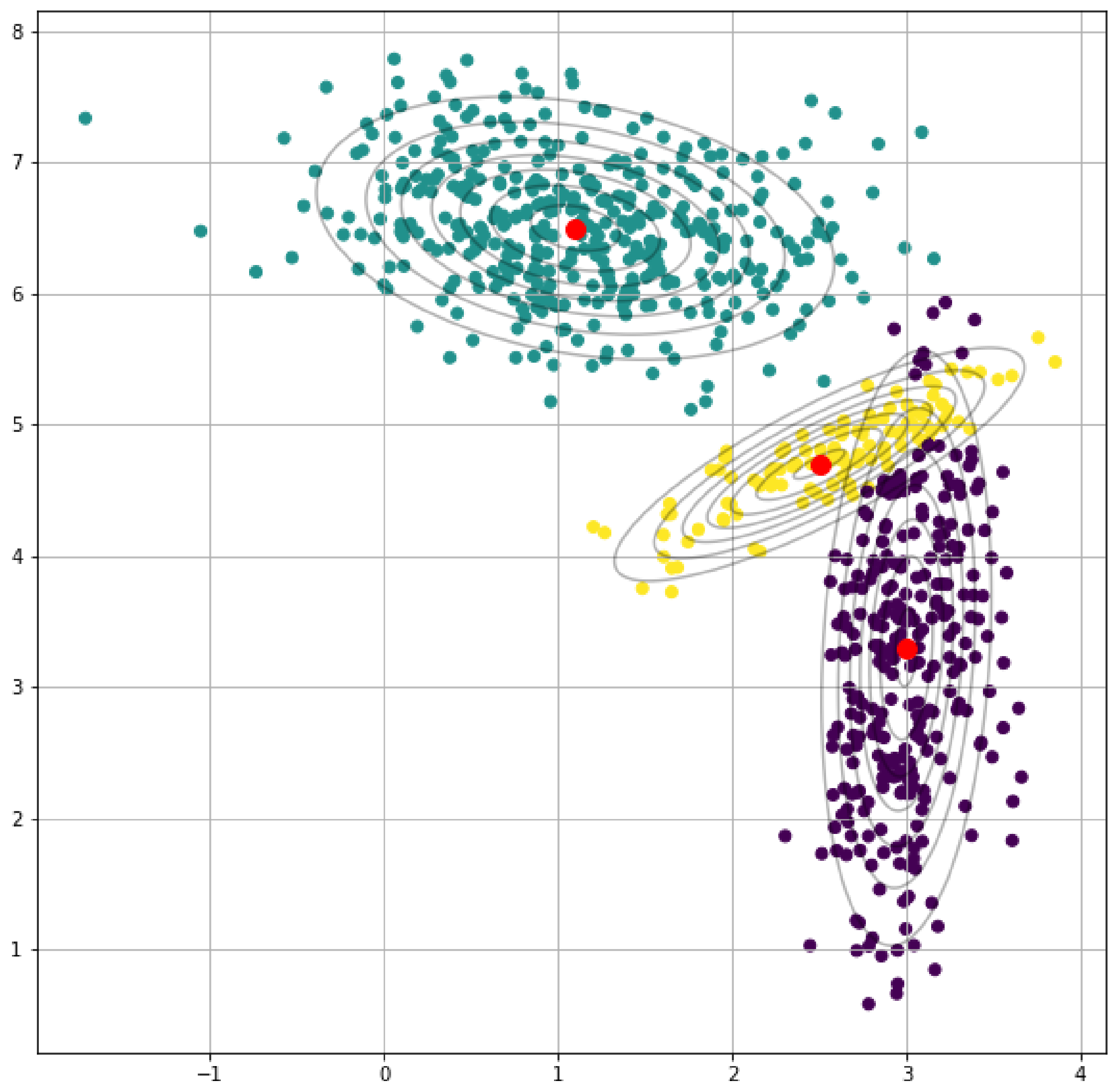
```
[92]: # Make some random data in 2D.
np.random.seed(150)
means = np.array([[1.1, 6.5],
                  [2.5, 4.7],
                  #[3.0, 2.6],
                  [3.0, 3.3]])
covariances = [np.array([[0.55, -0.10], [-0.10, 0.25]]),
                np.array([[0.35, 0.22], [0.22, 0.20]]),
                #np.array([[0.06, 0.05], [0.05, 1.30]]),
                np.array([[0.06, 0.05], [0.05, 1.30]])]
amplitudes = [4, 1, 3]
```

```
factor = 100

data = np.zeros((1, 2))
for i in range(len(means)):
    data = np.concatenate([data,
          np.random.multivariate_normal(means[i], covariances[i],
                                                  size=factor * amplitudes[i])])
data = data[1:, :]

#Finding the best number of clusters
# Initalize the MinmMaxScaler
Max_square = MinMaxScaler()
# fit the data
Max_square.fit(data)
# trasnform the data
data_transformed = Max_square.transform(data)

Sum_of_squared_distances = []
K = range(1,15)

# CHeckign the sum of squared distances for different clusters numbers
for k in K:
    km = KMeans(n_clusters=k)
    km = km.fit(data_transformed)
    Sum_of_squared_distances.append(km.inertia_)

# Plot the sum of the squared distance  v/s the clusters k

f = plt.figure(figsize=(10,5))
f.add_subplot(1,2, 1,title='Data')
plt.plot(data[:,0],data[:,1])
f.add_subplot(1,2, 2,title='Finding Optimal k')
plt.plot(K, Sum_of_squared_distances, 'bx-')
plt.xlabel('k')
plt.ylabel('Sum_of_squared_distances')
plt.show()

x,y = np.meshgrid(np.sort(data[:,0]),np.sort(data[:,1]))
XY = np.array([x.flatten(),y.flatten()]).T

# As k increases, the sum of squared distance tends to zero. If it's a large␣
 ↪value then each data form it's own clusers
# So we need to stop at a point where the sum_of_squared is stabilized.
# It is found evident from the graph of finding the optimal k that optimal sum␣
 ↪of square is 3
GMM = mixture.GaussianMixture(n_components=3).fit(data) # Instantiate and fit␣
 ↪the model
```

```
labels = GMM.predict(data)

print('Converged:',GMM.converged_) # Check if the model has converged



# Plot
fig = plt.figure(figsize=(10,10))
ax0 = fig.add_subplot(111)

ax0.scatter(data[:,0],data[:,1],c=labels)
for m,c in zip(means,covariances):
    multi_normal = multivariate_normal(mean=m,cov=c)
    ax0.contour(np.sort(data[:,0]),np.sort(data[:,1]),multi_normal.pdf(XY).
 ↪reshape(len(data),len(data)),colors='black',alpha=0.3)
    ax0.scatter(m[0],m[1],c='red',zorder=10,s=100)
plt.grid()
plt.show()
```



```
Converged: True
```



2.2.1 1. Implement Expectation Maximisation algorithm

- follow the technical advises given in the lecture, namely run first k-means for the first 10-100 iterations
- check if you need to add regilarisation

2.2.2 2. Visualise the results

- plot the samples colors, show using colors soft-assignments of the samples
- plot ellipsoids for Gaussians

Reference

- https://stackoverflow.com/questions/36013063/what-is-the-purpose-of-meshgrid-in-python-numpy

- https://towardsdatascience.com/gaussian-mixture-model-clusterization-how-to-select-the-number-of-components-clusters-553bef45f6e4
- https://www.python-course.eu/expectation_maximization_and_gaussian_mixture_models.php
- https://stackoverflow.com/questions/36013063/what-is-the-purpose-of-meshgrid-in-python-numpy
- https://dilloncamp.com/projects/kmeans.html
- https://www.geeksforgeeks.org/image-compression-using-k-means-clustering/