IE406 Machine Learning

Lab Assignment - 6

Group 14

201901466: Miti Purohit

202001430: Aryan Shah

[2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
import seaborn as sns
import cv2
import h5py
from sklearn.datasets import fetch_openml
import warnings
warnings.filterwarnings("ignore")
```

Question 1

Given the following data use Principal Component Analysis to reduce the feature dimension from 3 to 1 also show eigen values. (Do manual calculation)

[ ]:
```
x1 = [8,13,4,7,19,5]
x2 = [14,9,3,2,8,18]
x3 = [3,6,15,1,4,11]
X = [x1, x2, x3]
X = np.array(X)
X = X.T
print(X)
```

```
[[ 8 14  3]
 [13  9  6]
 [ 4  3 15]
 [ 7  2  1]
 [19  8  4]
 [ 5 18 11]]
```

[ ]:
```
# defining required array
X = np.array([[8,14,3],[13,9,6],[4,3,15],[7,2,1],[19,8,4],[5,18,11]])

# standard scaler to make mean 0 and variance 1
sc = StandardScaler()
X_std = sc.fit_transform(X)

# performing PCA
pca = PCA(n_components =1)
principalComponent = pca.fit_transform(X_std)
pComponents = pd.DataFrame(data = principalComponent , columns = ['principal␣
 ↪component 1'])

# printing the variance ratio of each component
print("Variance ratio: ", pca.explained_variance_ratio_)

print ("\n\nThe first principal component is: ")
print(pComponents)

print ("\n\nThe eigenvalues after the dimension reduction by PCA of the given␣
 ↪data are :")
print(pca.components_)
```

```
Variance ratio:  [0.49991964]


The first principal component is:
   principal component 1
0              -0.204236
1              -0.586974
2               1.740811
3              -0.710952
4              -1.708027
5               1.469378


The eigenvalues after the dimension reduction by PCA of the given data are :
[[-0.69401025  0.16586365  0.70059904]]
```

[ ]:
```
#MANUAL CALCULATION
from numpy import mean
from numpy import cov
from numpy import array
from numpy.linalg import eig

# calculating the mean of each column
Mean = mean(X.T, axis=1)
```

```
# determining center columns by subtracting column means
X_STD = X - Mean

# calculating covariance matrix of centered matrix
Va = cov(X_STD.T)

# eigendecomposition of covariance matrix
values, vectors = eig(Va)
print("Eigen Vectors :\n",vectors,'\n')
print("Eigen Values :\n",values,'\n')

# projectnig the data
Prj = vectors.T.dot(X_STD.T)
print(Prj.T)
```

```
Eigen Vectors :
 [[ 0.65370829  0.70071725 -0.28576354]
 [-0.04563208 -0.34043372 -0.93916058]
 [ 0.75536957 -0.62697704  0.19056916]]

Eigen Values :
 [15.41684401 46.07005232 37.44643701]

[[-3.86945988 -0.33754247 -5.01353842]
 [ 1.89335068  2.98728129 -1.17484576]
 [ 3.08209469 -6.91936503  8.74711203]
 [-5.48632232  4.30089903  6.16101375]
 [ 4.35049337  8.78597262 -2.33140475]
 [ 0.02984346 -8.81724545 -6.38833684]]
```

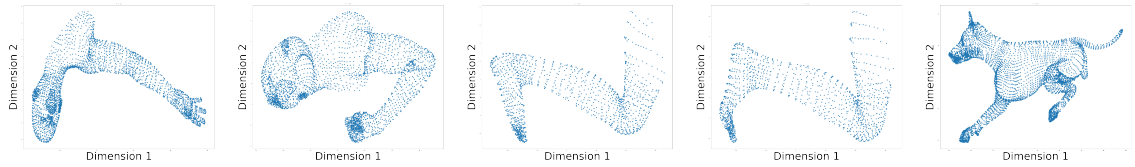
Question 2

A classic application of PCA is to project the 3-D point cloud onto a plane that could still retain the information or essence of the point distribution. Given a 3-D point cloud, estimate an optimal plane, onto which if the 3D data points when projected would still retain the essential information. We provide you with 5 different point clouds P_1.txt to P_5.txt containing the 3D coordinates of points in space. Perform PCA on these point clouds to obtain their projection on a 2D plane and visualize the results using python libraries (like matplotlib).

[ ]:
```
# reading the data files
p1 = np.array(pd.read_csv('/content/P_1.txt', header = None , delim_whitespace =␣
 ↪True))
p2 = np.array(pd.read_csv('/content/P_2.txt', header = None , delim_whitespace =␣
 ↪True))
p3 = np.array(pd.read_csv('/content/P_3.txt', header = None , delim_whitespace =␣
 ↪True))
```

```
p4 = np.array(pd.read_csv('/content/P_4.txt', header = None , delim_whitespace =␣
 ↪True))
p5 = np.array(pd.read_csv('/content/P_5.txt', header = None , delim_whitespace =␣
 ↪True))
```

[ ]:
```
fig = plt.figure(figsize=(150 , 20))
subplot_index = 1

# 2-D PLOTING
for point_cloud in [p1 , p2 , p3 , p4 , p5]:
  pca = PCA(n_components = 2)
  principal_components = pca . fit_transform(point_cloud , y = None)

  plt.subplot(1,5,subplot_index)
  plt.scatter( principal_components[: ,0] , principal_components[: ,1])
  plt.xlabel('Dimension 1', fontsize =80)
  plt.ylabel('Dimension 2', fontsize =80)
  plt.title('P_ '+ str(subplot_index) + '.txt')
  subplot_index += 1
```



Question 3

Another classic example of PCA comes in image compression. The human eye cannot perceive the minute and frequent changes in the image. A typical smartphone camera takes a 5MP image on average. We will utilize PCA and analyze how the reduction in the no. of principal components or dimensions affects the visual quality of the image. Given an image, apply PCA to investigate the effect of reducing the dimensions on the visual quality of the image. You will use this image and show the analysis.

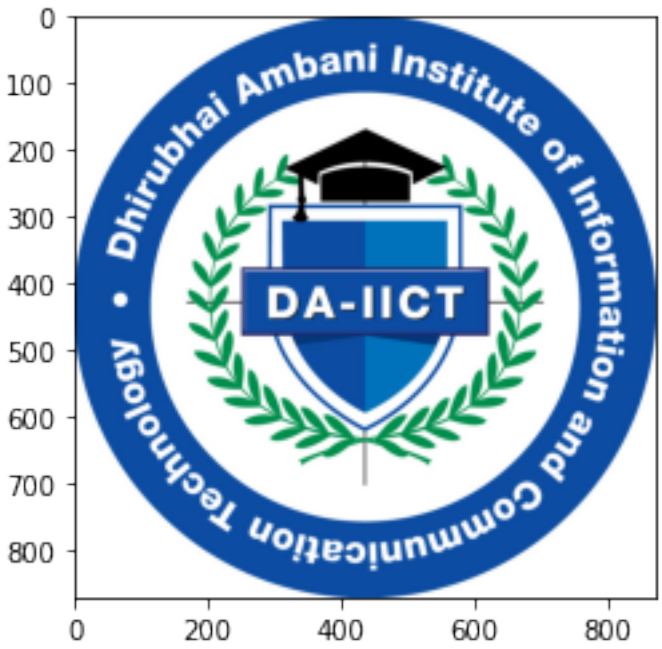
Answer

Code

[ ]:
```
img = cv2.cvtColor(cv2.imread('/content/DA-IICT-Emblem-Final Colour.jpg'), cv2.
 ↪COLOR_BGR2RGB)
print(img.shape)
plt.imshow(img)
```

```
(870, 870, 3)
```

[ ]:
```
<matplotlib.image.AxesImage at 0x7f8f10d72190>
```



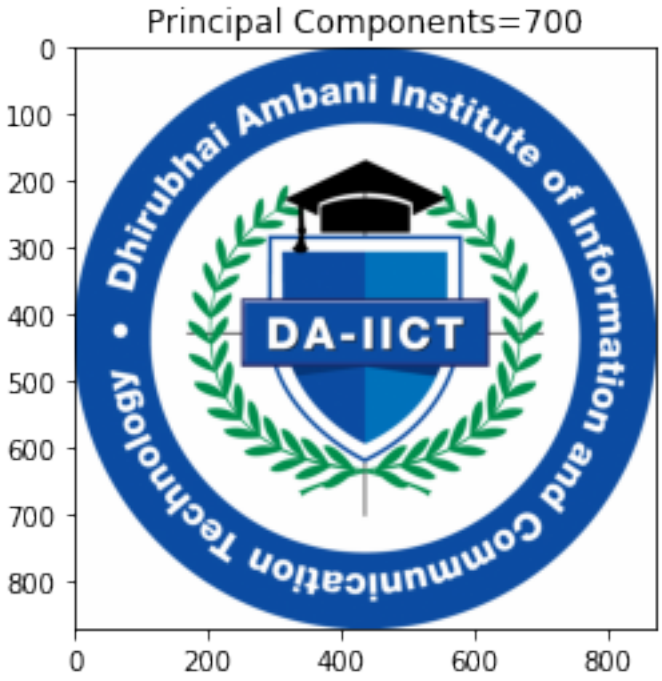
[ ]:
```
blue,green,red = cv2.split(img)
```

[ ]:
```
# Normalizing the Color arrays
scaled_blue = blue/255
scaled_green = green/255
scaled_red = red/255
# Taking different Components to test image transformation on different␣
 ↪amount␣of Principal Components
pcas=np.array([700,500,300,100,50,20,10,5,1])
variance_ratio_blue = []
variance_ratio_green = []
variance_ratio_red = []
for num in pcas:

  # Applying PCA on Blue Color
  pb = PCA(n_components=num)
  pb.fit(scaled_blue)
  transformed_scaled_blue = pb.transform(scaled_blue)
  new_blue = pb.inverse_transform(transformed_scaled_blue)

  # Applying PCA on Green Color
  pg = PCA(n_components=num)
  pg.fit(scaled_green)
  transformed_scaled_green = pg.transform(scaled_green)
  new_green = pg.inverse_transform(transformed_scaled_green)
```

```
# Applying PCA on Red Color
pr = PCA(n_components=num)
pr.fit(scaled_red)
transformed_scaled_red = pr.transform(scaled_red)
new_red = pr.inverse_transform(transformed_scaled_red)

# Computing Explained Variance
variance_ratio_blue.append(np.sum(pb.explained_variance_ratio_))
variance_ratio_green.append(np.sum(pg.explained_variance_ratio_))
variance_ratio_red.append(np.sum(pr.explained_variance_ratio_))

# Merging back the transformed arrays
new_image = cv2.merge((new_blue,new_green,new_red))
plt.figure()
print(new_image.shape)
plt.title('Principal Components='+str(num))
plt.imshow(new_image)
plt.show()
```

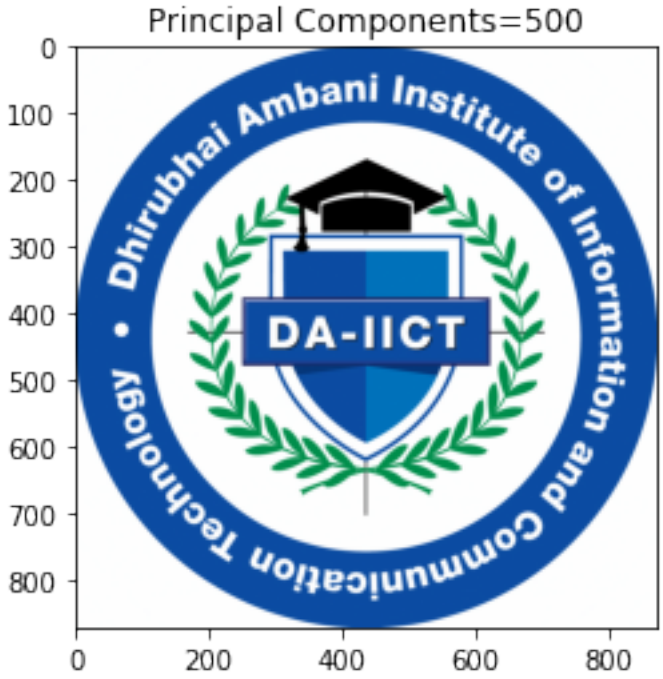
```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
RGB data ([0..1] for floats or [0..255] for integers).
```

```
(870, 870, 3)
```



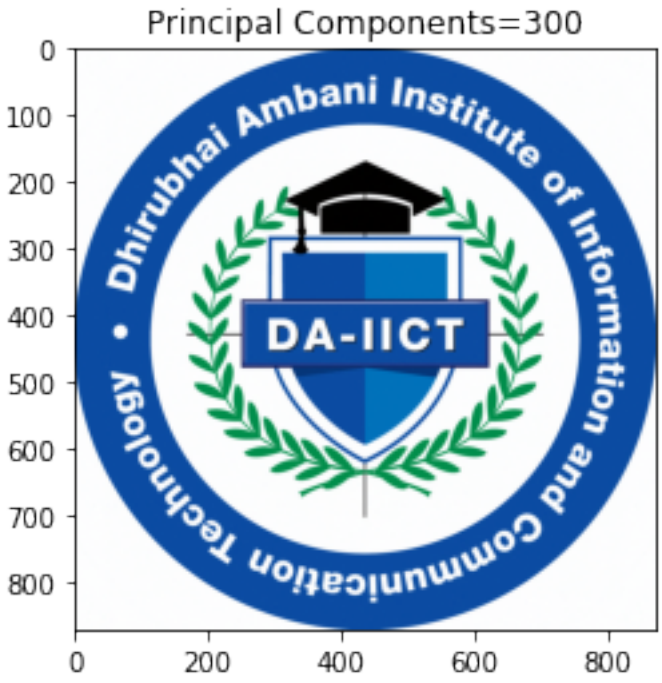
```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
RGB data ([0..1] for floats or [0..255] for integers).
```

```
(870, 870, 3)
```



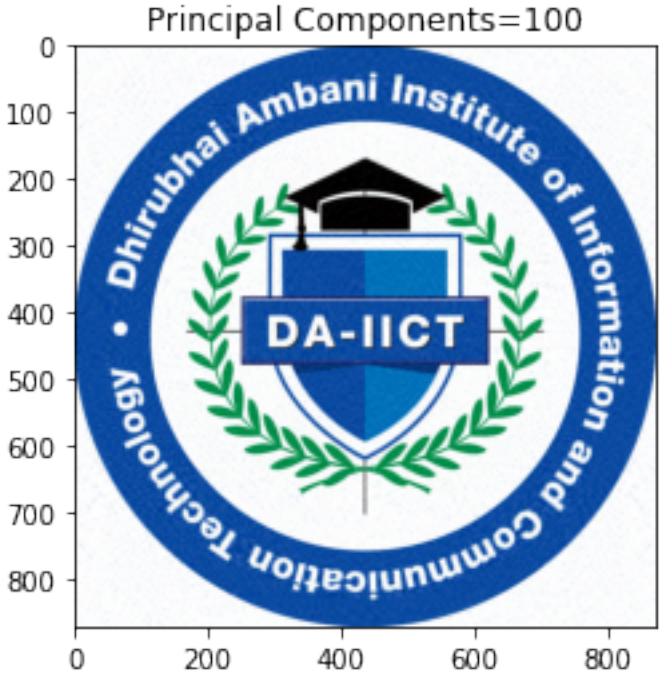

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
RGB data ([0..1] for floats or [0..255] for integers).
```

```
(870, 870, 3)
```



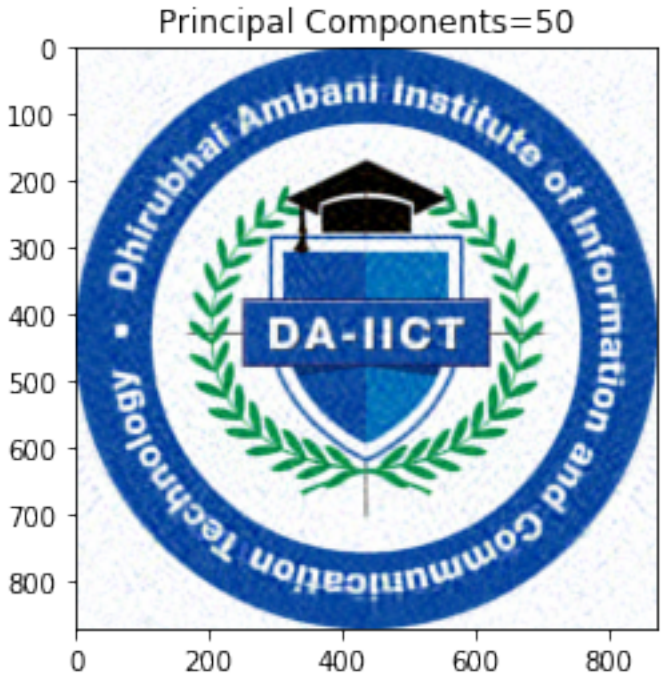

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
RGB data ([0..1] for floats or [0..255] for integers).
```

```
(870, 870, 3)
```



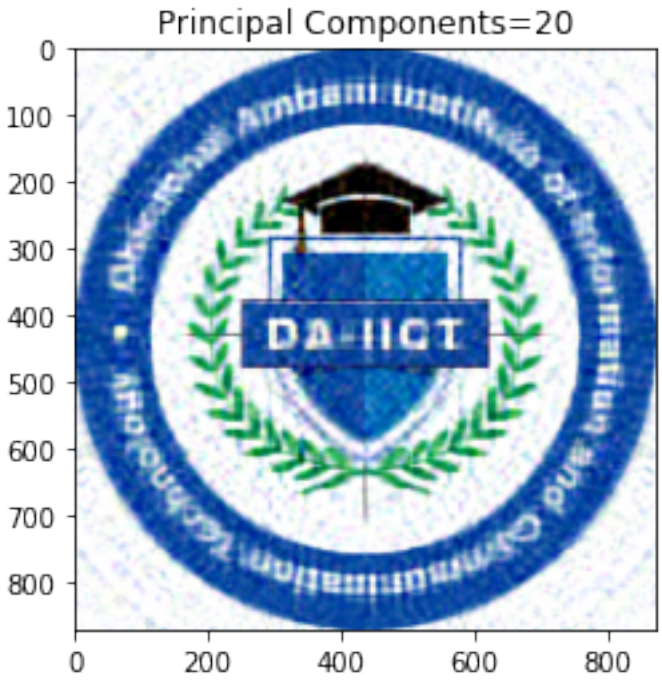
```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
RGB data ([0..1] for floats or [0..255] for integers).
```

```
(870, 870, 3)
```



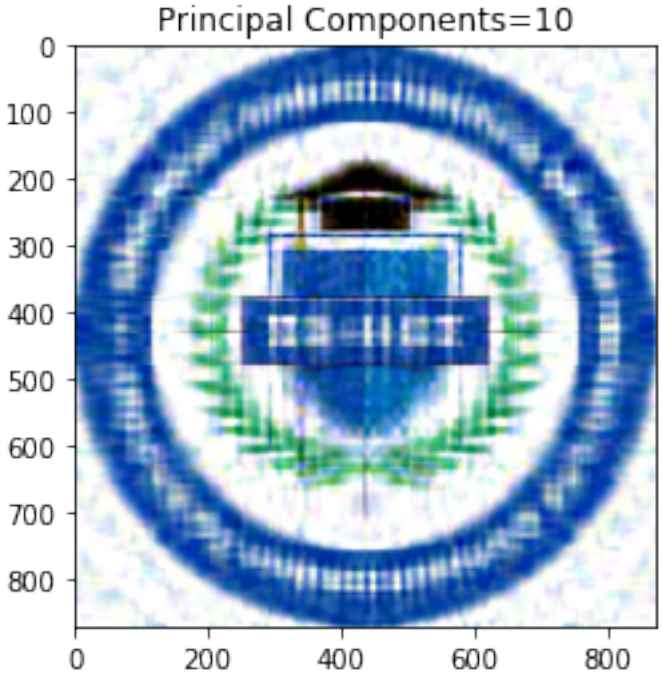

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
RGB data ([0..1] for floats or [0..255] for integers).
```

```
(870, 870, 3)
```



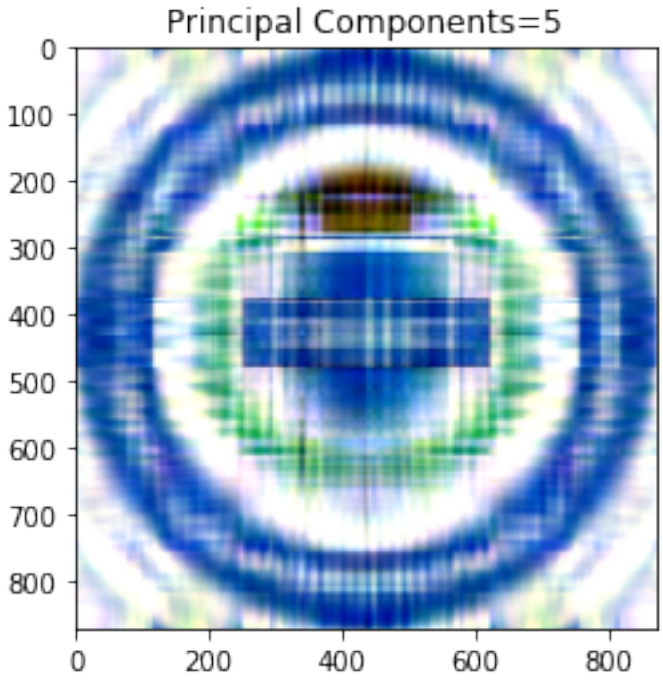

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
RGB data ([0..1] for floats or [0..255] for integers).
```

```
(870, 870, 3)
```



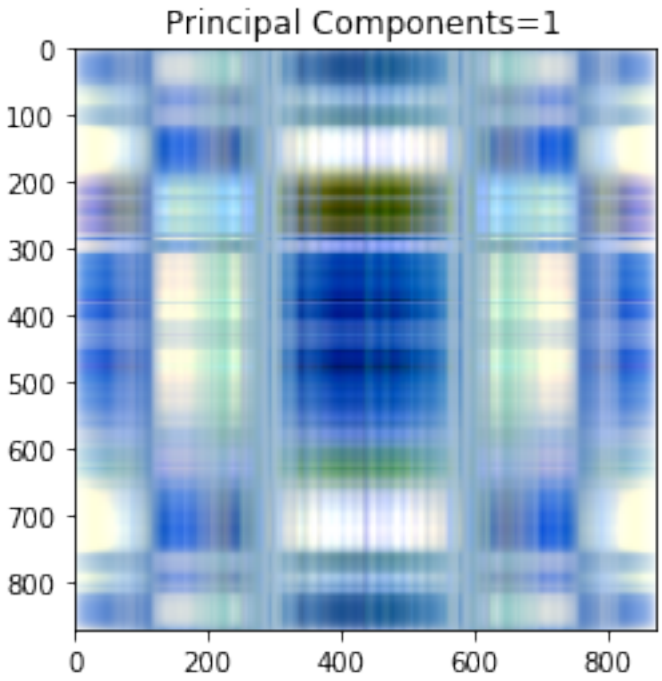
```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
RGB data ([0..1] for floats or [0..255] for integers).
```

```
(870, 870, 3)
```




```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
RGB data ([0..1] for floats or [0..255] for integers).
```

```
(870, 870, 3)
```



Result

The images for different value of principal components is as shown above.

Observation/Justification

As the number of principal components decreases, the quality of the image

Question 4

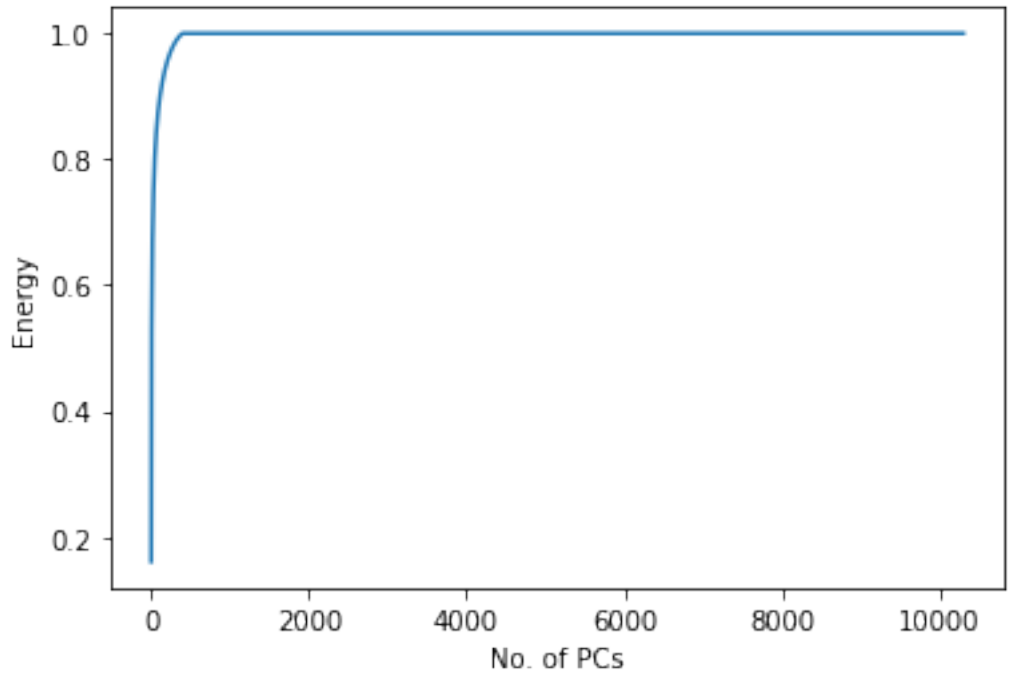
[69]:
```
#loading data from .mat file
filepath = "/content/faceimages.mat"

with h5py.File(filepath, 'r') as file:
    print(list(file.keys()))
    data = np.array(file['data'])
x = data[:-1, :]
y = data[-1, :]
x_mean = [np.mean(j) for j in x]
x = x.T
x = StandardScaler().fit_transform(x)
```

```
['Database_name', 'Record_Name', 'data', 'height', 'width']
```

[70]:
```
x_cov_mat = np.cov(x.T)
eig_val_asc, eig_vect_asc = np.linalg.eigh(x_cov_mat)
```

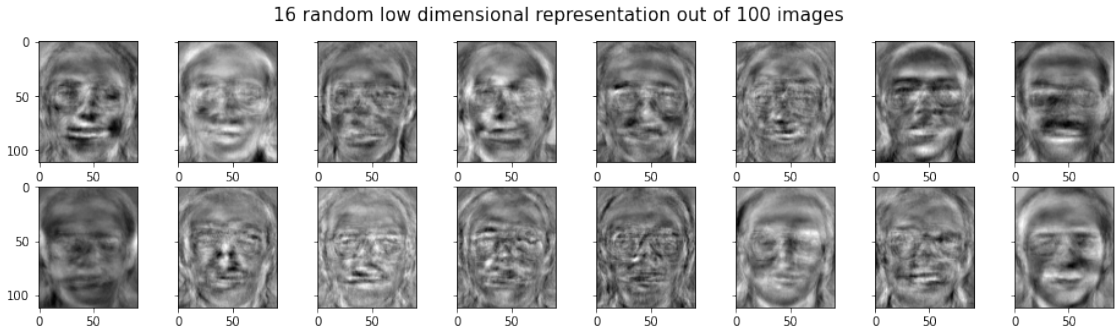
```
eig_val = np.flip(eig_val_asc)
eig_vect = np.flip(eig_vect_asc, axis=1)
var_exp = [(i/sum(eig_val)) for i in eig_val]
cum_var_exp = np.cumsum(var_exp)

plt.plot(range(10304), cum_var_exp[:])
plt.xlabel('No. of PCs')
plt.ylabel('Energy')
```

[70]: Text(0, 0.5, 'Energy')



[71]:
```
fig, ax = plt.subplots(2,8, sharey = True, figsize = (16,4))
random_index = random.sample(range(100), 16)
for i in range(16):
    ax[int(i/8)][i%8].imshow(eig_vect[:, random_index[i]].reshape(92,112).T,␣
 ↪cmap='gray')
fig.suptitle('16 random low dimensional representation out of 100 images',␣
 ↪fontsize=15)
```

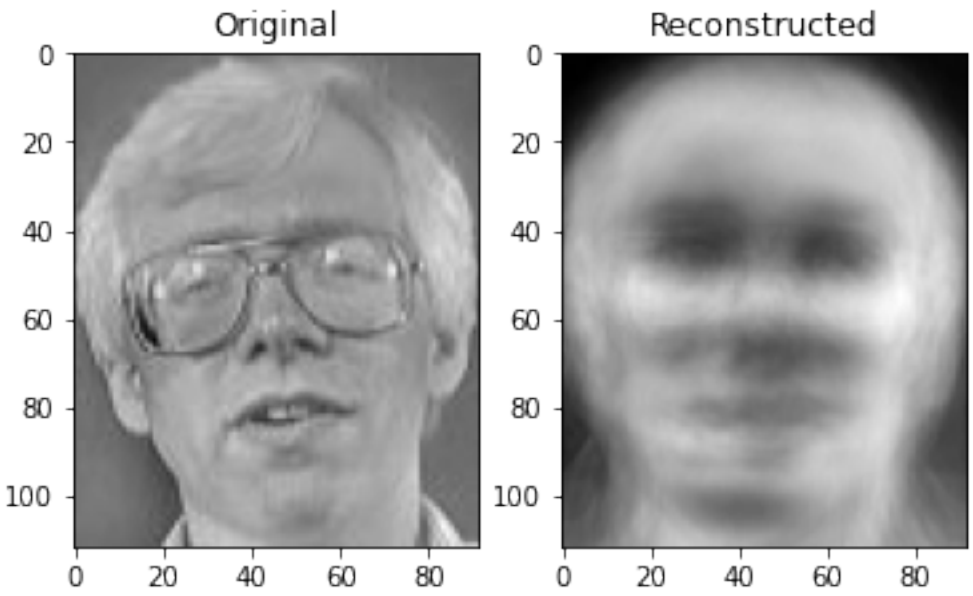
[71]: Text(0.5, 0.98, '16 random low dimensional representation out of 100 images')



[72]:
```
def n_pcs(n, ith):
    #ith is the ith image out of 400 images available in the dataset
    #taking last n components as the eigen values are in ascending order
    eigenvec_n = eig_vect[:, :n]

    plt.subplot(121)
    plt.imshow(x[ith, :].reshape(92, 112).T, cmap='gray')
    plt.title('Original')

    #reducing dimensions
    x_red = x[ith].dot(eigenvec_n)

    #reconstuct
    x_rec = x_red.dot(eigenvec_n.T)
    plt.subplot(122)
    plt.imshow(x_rec[:].reshape(92, 112).T, cmap='gray')
    plt.title('Reconstructed')

    recon_error = sqrt(mean_squared_error(x[ith], x_rec))
    print('Reconstruction Error:', recon_error)
```

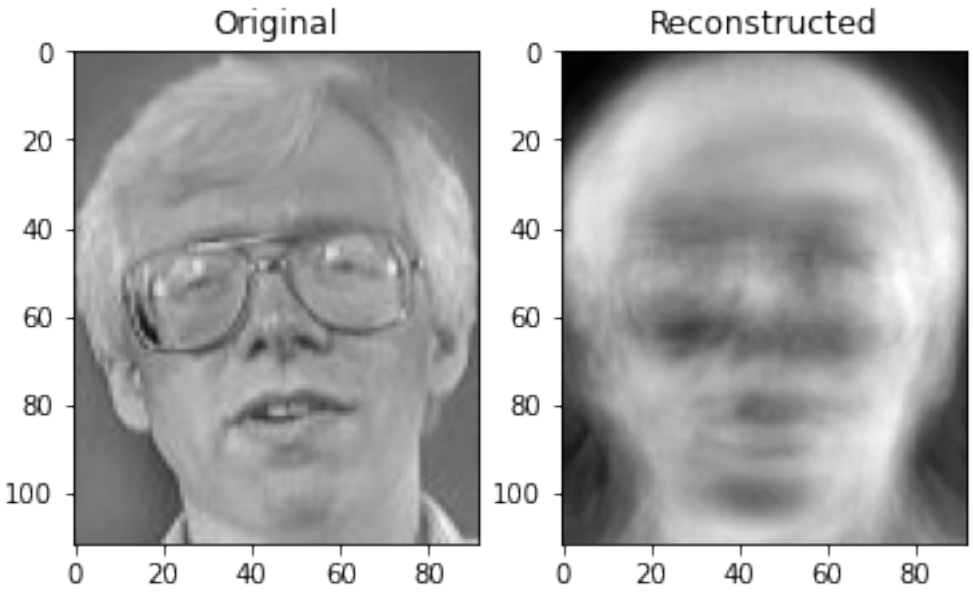
[73]:
```
n_pcs(5,10)
```

```
Reconstruction Error: 0.7375134196135179
```



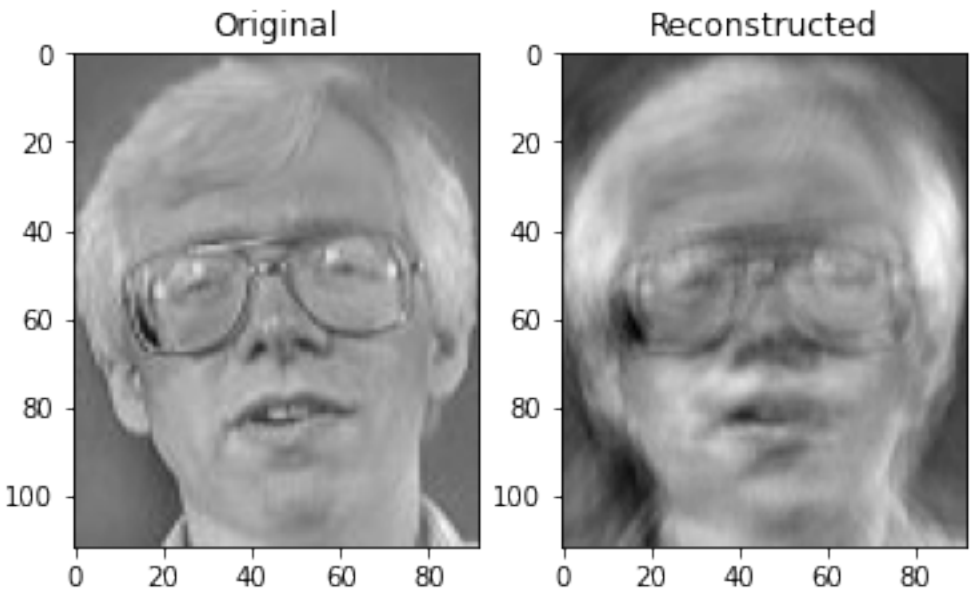
[74]:
```
n_pcs(10,10)
```

```
Reconstruction Error: 0.6842355351284524
```



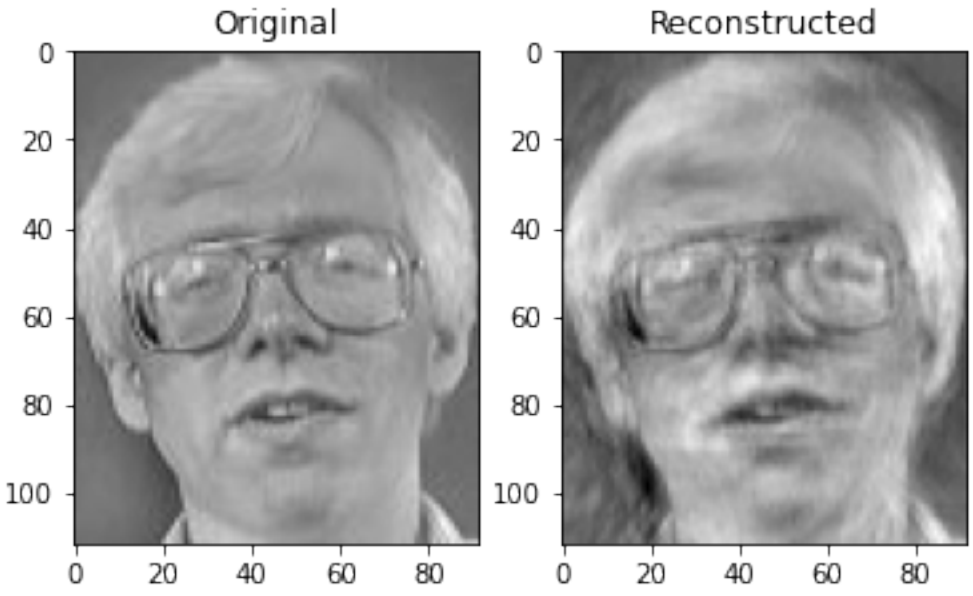
[75]:
```
n_pcs(50,10)
```

```
Reconstruction Error: 0.48366687077576925
```



[76]:
```
n_pcs(100,10)
```

```
Reconstruction Error: 0.3618079946389883
```



Question 5

To fetch the MNIST Dataset

[77]:
```
# Fetching the MNIST Dataset
mnist_dataset = fetch_openml('mnist_784')
print(mnist_dataset.target)
```

```
['5' '0' '4' ... '4' '5' '6']
```

[78]:
```
# Creating Data & Target arrays From the dataset
X = pd.DataFrame(data = mnist_dataset.data, columns =  mnist_dataset.
 ↪feature_names)
y = mnist_dataset.target.astype('int64')
print(y)
```

```
[5 0 4 ... 4 5 6]
```

[79]:
```
# Constructing a PCA model with 200 components
pca_model = PCA(n_components=200)
pca_model.fit(X)
```

[79]:
```
PCA(copy=True, iterated_power='auto', n_components=200, random_state=None,
    svd_solver='auto', tol=0.0, whiten=False)
```

[80]:
```
# Computing cumulative variance of the model
cumulative_variance = pca_model.explained_variance_ratio_.cumsum() * 100
print(cumulative_variance)
```

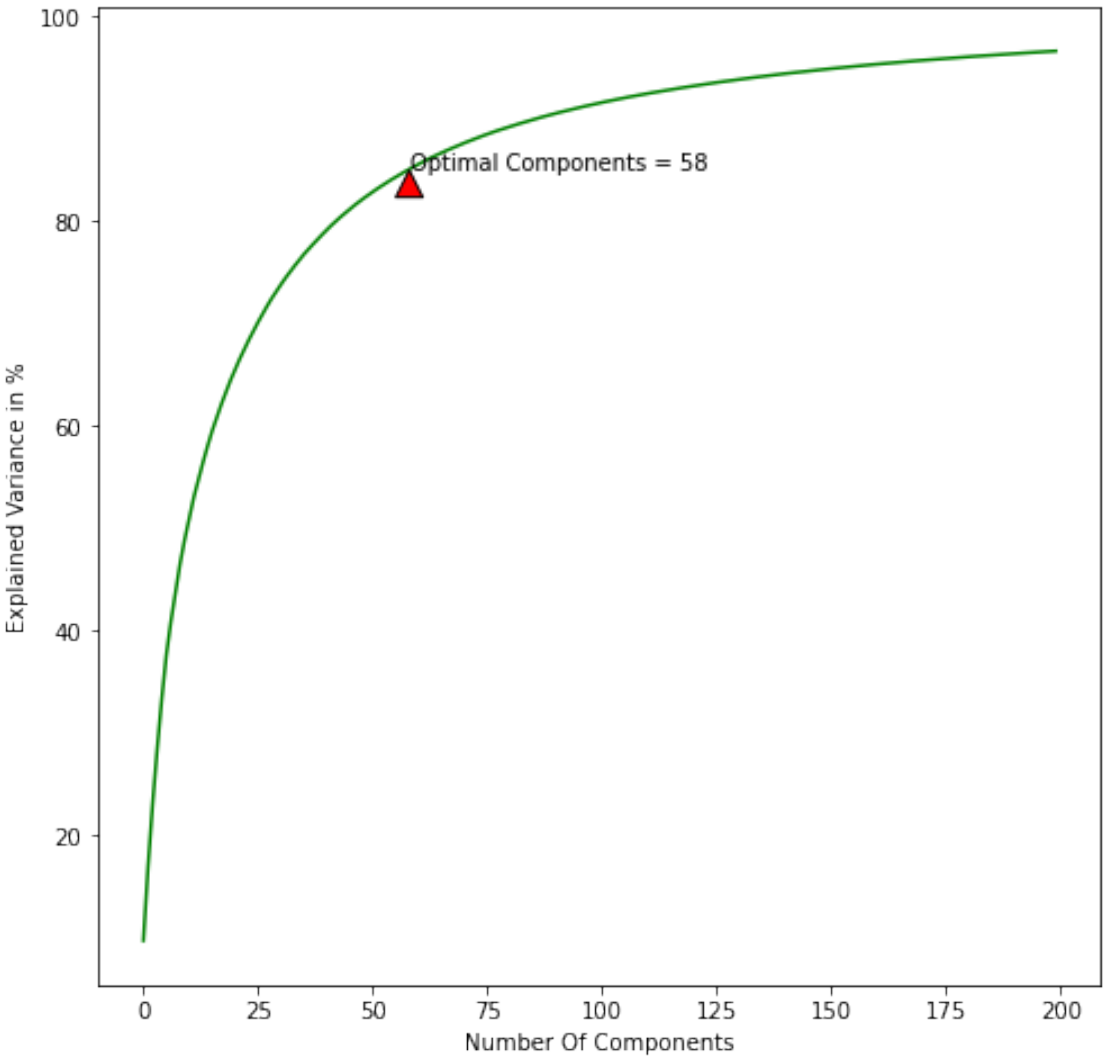
```
[ 9.74611592 16.90156051 23.05109149 28.45447602 33.34340972 37.64863675
 40.92689828 43.81653984 46.57490404 48.91704421 51.02373276 53.061286
 54.76835031 56.46237009 58.04575233 59.53209723 60.85145572 62.13046998
 63.31774246 64.47067917 65.53671895 66.54651329 67.50566452 68.4152996
 69.29858642 70.13740462 70.94723642 71.73295432 72.47321655 73.16323071
 73.8193747  74.46484499 75.06566429 75.65127604 76.2180296  76.76150026
 77.26621733 77.75329675 78.23225188 78.69984553 79.15421416 79.59913217
 80.01734877 80.41351268 80.79722042 81.17300523 81.53443151 81.88345611
 82.22218771 82.54188388 82.85873791 83.16888282 83.46536318 83.75246462
 84.03497842 84.30440113 84.57279344 84.82930277 85.08247063 85.32711866
 85.56682089 85.80540189 86.03463564 86.25558463 86.4686451  86.67496256
 86.87774394 87.07277906 87.26424968 87.45279918 87.63977532 87.81987957
 87.99666661 88.17002526 88.33487461 88.49811164 88.65951968 88.81382373
 88.96084416 89.10304361 89.24405976 89.38420593 89.52381076 89.65883518
 89.7912066  89.92309949 90.0522969  90.17746888 90.2999983  90.42041749
 90.53678023 90.65109903 90.76364585 90.87350736 90.98185038 91.08902553
 91.19271748 91.2961355  91.39672585 91.49665809 91.59433137 91.68849218
 91.78207436 91.87323973 91.96331262 92.05226179 92.13841799 92.22366955
 92.3077681  92.38948821 92.46808786 92.54583432 92.623412   92.69988774
 92.77594189 92.85081633 92.92383732 92.99636497 93.06794542 93.13827984
 93.20744499 93.27613204 93.34406547 93.41123028 93.4773477  93.54160622
 93.60487485 93.66770153 93.72965549 93.78977956 93.84978722 93.90914779
 93.9677528  94.02608263 94.08394794 94.14119946 94.19751106 94.25269838
 94.30620472 94.35874572 94.41098519 94.46193112 94.51207828 94.56184797
```

```
94.61137835 94.66054389 94.70883815 94.75678475 94.80389751 94.85048088
94.89679603 94.94294922 94.98863412 95.03335023 95.0778739  95.12181908
95.16545536 95.20791049 95.24982426 95.2914313  95.33230637 95.37292099
95.41250037 95.4515765  95.49059637 95.52911956 95.56724887 95.60476058
95.64206692 95.67884642 95.71503555 95.75083846 95.7864872  95.82159987
95.85620008 95.89065422 95.92489121 95.9586661  95.99223051 96.02542636
96.05786891 96.08981756 96.12171987 96.15327761 96.18442189 96.21542841
96.24583693 96.27595494 96.3059501  96.33561787 96.364891   96.39402242
96.42281001 96.45122858 96.47930281 96.50726044 96.53481865 96.56225576
96.58901735 96.61558966]
```

[81]:
```
# Extracting Optimal number of PCA components required to get 85% variance
components = 0
for i in range(len(cumulative_variance)):
  if cumulative_variance[i] > 85:
    components = i
    break

print(components)
```

```
58
```

[82]:
```
plt.figure(figsize = (8,8))
plt.plot(cumulative_variance, 'g')
plt.annotate('Optimal Components = 58', xy =(58, 85),
                arrowprops = dict(facecolor ='red',
                                  shrink = 0.05),)
plt.xlabel('Number Of Components')
plt.ylabel('Explained Variance in %')
plt.show()
```


Optimal Components = 58
Explained Variance in %
Number Of Components
100
80
60
40
20
0
25
50
75
100
125
150
175
200