Facial Recognition using Principal Component Analysis

(From Scratch)

1 Introduction

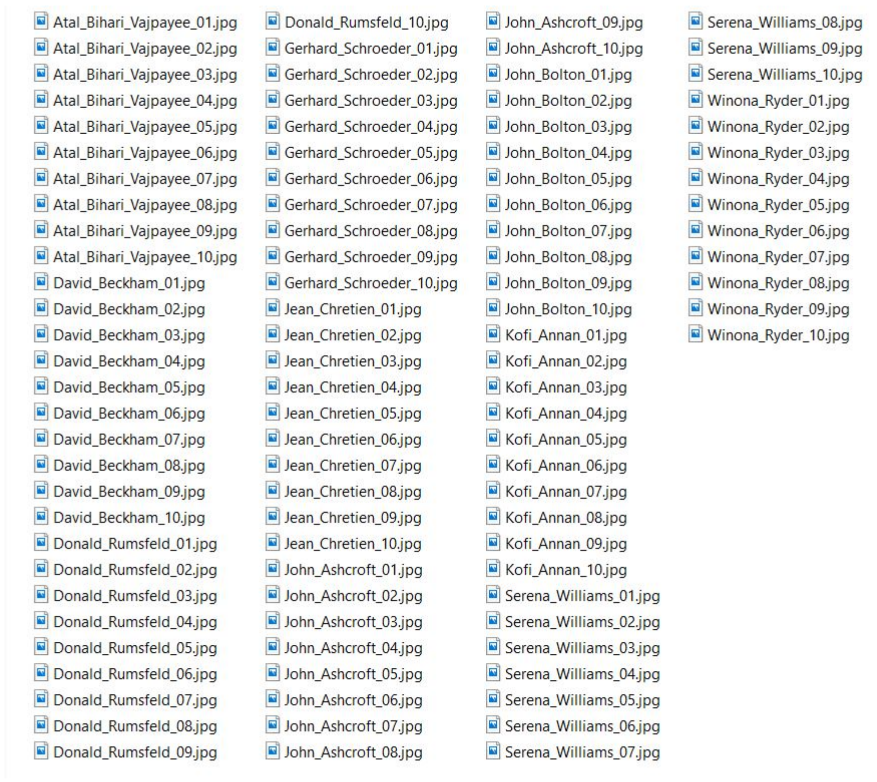
Principal Component Analysis (PCA) is a machine learning algorithm which is commonly used in exploratory data analysis and hence making predictive models.
PCA mainly uses the concept of dimensionality reduction . i.e. it keeps the directions where there is larger variance in the training data and neglects the direction with lower variance. Hence, it keeps only the direction with larger variance and is able to represent the whole data in those directions, thus reducing the dimensionality of the data.
There are lots of applications of PCA where it can be applied , like it is used for image compression, anamoly detection , medical field etc.
I, here used PCA for facial recognition .

2 Dataset

I take the dataset "Labeled faces in the wild (LFW)" from the kaggle. However , I did not use the whole "LFW" dataset in the project. I only used 100 images from this huge dataset for my purpose and then preprocessed them (all preprocessing steps mentions ahead) and then finally stored them under the folder name "dataset" .
These 100 images consists of 10 unique person each having 10 images (thus constituting 10*10 = 100 images).

The images are stored in the folder as following:
for example if a person's name is "Sumit Negi". Then its all 10 images are stored as:
"Sumit_Negi_01.jpg", "Sumit_Negi_02.jpg",....."Sumit_Negi_10.jpg".

The "dataset" folder looks like as,

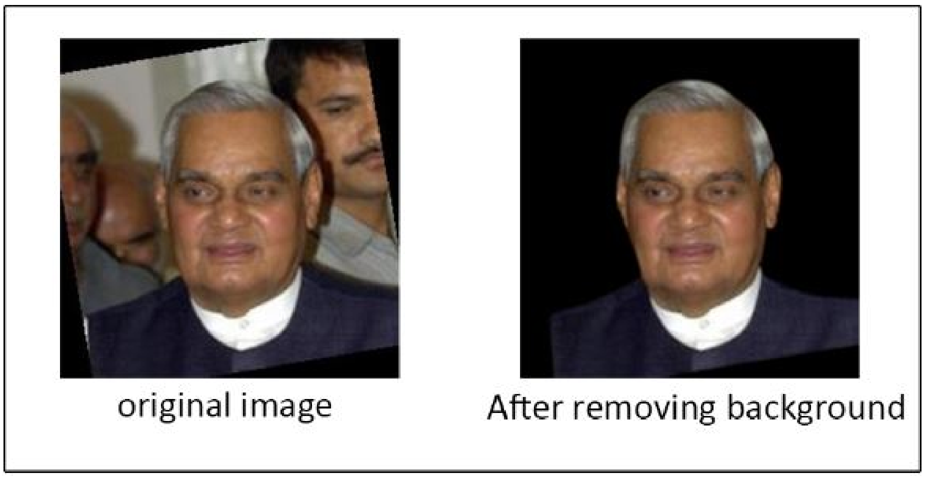


```
Atal_Bihari_Vajpayee_01.jpg
Atal_Bihari_Vajpayee_02.jpg
Atal_Bihari_Vajpayee_03.jpg
Atal_Bihari_Vajpayee_04.jpg
Atal_Bihari_Vajpayee_05.jpg
Atal_Bihari_Vajpayee_06.jpg
Atal_Bihari_Vajpayee_07.jpg
Atal_Bihari_Vajpayee_08.jpg
Atal_Bihari_Vajpayee_09.jpg
Atal_Bihari_Vajpayee_10.jpg
David_Beckham_01.jpg
David_Beckham_02.jpg
David_Beckham_03.jpg
David_Beckham_04.jpg
David_Beckham_05.jpg
David_Beckham_06.jpg
David_Beckham_07.jpg
David_Beckham_08.jpg
David_Beckham_09.jpg
David_Beckham_10.jpg
Donald_Rumsfeld_01.jpg
Donald_Rumsfeld_02.jpg
Donald_Rumsfeld_03.jpg
Donald_Rumsfeld_04.jpg
Donald_Rumsfeld_05.jpg
Donald_Rumsfeld_06.jpg
Donald_Rumsfeld_07.jpg
Donald_Rumsfeld_08.jpg
Donald_Rumsfeld_09.jpg
Donald_Rumsfeld_10.jpg
Gerhard_Schroeder_01.jpg
Gerhard_Schroeder_02.jpg
Gerhard_Schroeder_03.jpg
Gerhard_Schroeder_04.jpg
Gerhard_Schroeder_05.jpg
Gerhard_Schroeder_06.jpg
Gerhard_Schroeder_07.jpg
Gerhard_Schroeder_08.jpg
Gerhard_Schroeder_09.jpg
Gerhard_Schroeder_10.jpg
Jean_Chretien_01.jpg
Jean_Chretien_02.jpg
Jean_Chretien_03.jpg
Jean_Chretien_04.jpg
Jean_Chretien_05.jpg
Jean_Chretien_06.jpg
Jean_Chretien_07.jpg
Jean_Chretien_08.jpg
Jean_Chretien_09.jpg
Jean_Chretien_10.jpg
John_Ashcroft_01.jpg
John_Ashcroft_02.jpg
John_Ashcroft_03.jpg
John_Ashcroft_04.jpg
John_Ashcroft_05.jpg
John_Ashcroft_06.jpg
John_Ashcroft_07.jpg
John_Ashcroft_08.jpg
John_Ashcroft_09.jpg
John_Ashcroft_10.jpg
John_Bolton_01.jpg
John_Bolton_02.jpg
John_Bolton_03.jpg
John_Bolton_04.jpg
John_Bolton_05.jpg
John_Bolton_06.jpg
John_Bolton_07.jpg
John_Bolton_08.jpg
John_Bolton_09.jpg
John_Bolton_10.jpg
Kofi_Annan_01.jpg
Kofi_Annan_02.jpg
Kofi_Annan_03.jpg
Kofi_Annan_04.jpg
Kofi_Annan_05.jpg
Kofi_Annan_06.jpg
Kofi_Annan_07.jpg
Kofi_Annan_08.jpg
Kofi_Annan_09.jpg
Kofi_Annan_10.jpg
Serena_Williams_01.jpg
Serena_Williams_02.jpg
Serena_Williams_03.jpg
Serena_Williams_04.jpg
Serena_Williams_05.jpg
Serena_Williams_06.jpg
Serena_Williams_07.jpg
Serena_Williams_08.jpg
Serena_Williams_09.jpg
Serena_Williams_10.jpg
Winona_Ryder_01.jpg
Winona_Ryder_02.jpg
Winona_Ryder_03.jpg
Winona_Ryder_04.jpg
Winona_Ryder_05.jpg
Winona_Ryder_06.jpg
Winona_Ryder_07.jpg
Winona_Ryder_08.jpg
Winona_Ryder_09.jpg
Winona_Ryder_10.jpg
```

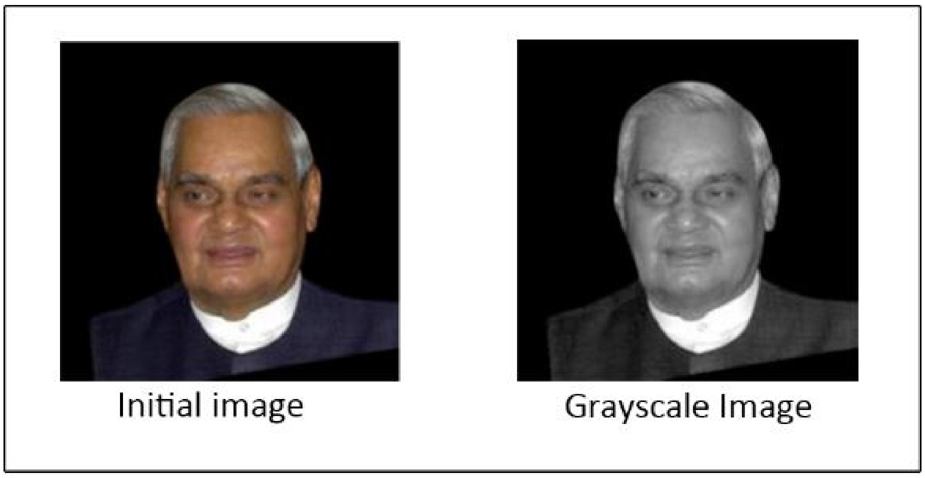
3 Data Pre-processing

I have performed several steps in the preprocessing part. These are as follows:

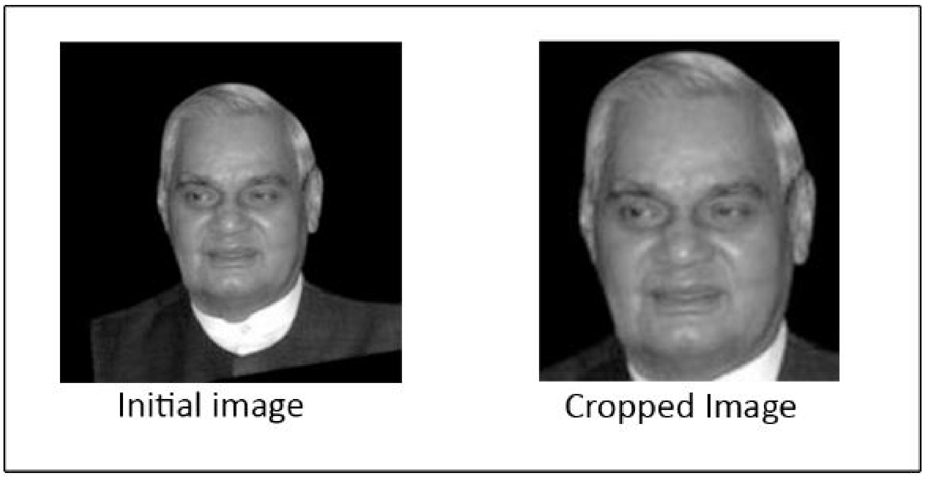
• **Removal of Background :** I have removed the background from the images so that the background pixels won't take part in deciding who the person is.
The images look like this after removal of background.



• **Grayscaling the image :** The colour images (i.e. images with RGB component) have components corresponding to three parts(R,G and B) . Taking colour images and directly applying PCA can makes calculations messy (computationally large time taking even with small number of images) . So, I converted them into grayscale before applying PCA.
The grayscale images look like this .

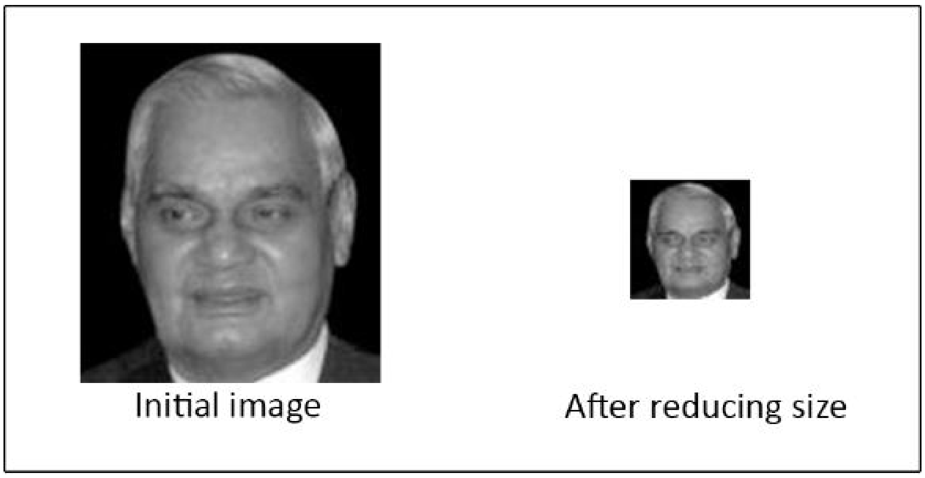


• **Cropping the face :** After converting the coloured images into grayscale version, then I cropped the face part from the images. It is done to remove the additional parts (clothes or whatever) from the images so that the there is less and less contribution of the things other than the face in the PCA calculations.
The images look like this after cropping of face.



• **Reducing the size of the image :** The processed image still has large size. So, large number of pixels in the image makes calculations lengthy . For example, the original image in the LFW dataset was of 250*250 dimensions. If we carry on with 250*250 dimensions, then each image has (250*250=6250000) features in grayscale . And we have taken 100 images in our dataset . So, our data matrix will be of 100*6250000 size, where 100 denotes the number of data points and 6250000 denotes the features per image.
After that the covariance matrix will be of size 6250000*6250000 . This will be a very large matrix . Eigen value, eigen vectors calculations of such matrix will take huge amount of time.
So, before going into calculation part, I reduces the size of the image . I reduces size of each image to 50*50 dimension. Doing this significantly reduces the execution time.

The transformed images looked like as:



• **Manual adjusting of image to bring face in the center:** Apart from above preprocessing, I also performed some manual centering of face. Some faces were not in the center of the original image, so i manually centered them by adjusting the size of the cropped window so that all the faces are centered.

The preprocessing code (python code) is contained in the "preprocessing codes" folder.

4 Implementing PCA

• **Formation of data matrix :** We have 100 images in the "dataset" folder.
Now, I have sampled 90 images from "dataset" folder into "train_image_list" and the remaining 10 images goes into "test_image_list" . So, our training : testing set is 90:10 .
Now, each image is of dimension 50*50 i.e. 2500 features per image.
So, our training data matrix is of 90*2500 dimension.

$$\text{i.e. train\_data\_matrix} = \begin{bmatrix} --------------\text{Train Image 1}\ ------------- \\ --------------\text{Train Image 2}\ ------------- \\ \vdots \\ --------------\text{Train Image 3}\ ------------- \\ --------------\text{Train Image 90}\ ------------- \end{bmatrix}_{90*2500}$$

and test data matrix is of 10*2500 dimension

$$\text{i.e. test\_data\_matrix} = \begin{bmatrix} --------------\text{Test Image 1}\ ------------- \\ --------------\text{Test Image 2}\ ------------- \\ \vdots \\ --------------\text{Test Image 3}\ ------------- \\ --------------\text{Test Image 10}\ ------------- \end{bmatrix}_{10*2500}$$

• **Calculating average face :** Average face is calculated by the training images. The average face is stored in "average_face" in the code. Hence, average_face is 1-D array of size 1*2500.

I have executed the code for 10 iterations and each time, there is new training and test set. So,here I am showing average face of some random iteration. It look like as:

Average face of all the training faces is as :

• **Centering of data matrix :** Centering of the images is done by subtracting the average face from the all training images and all test images in the train_data_matrix and test_data_matrix respectively.
Let now the centred matrix be called centred_train_data_matrix and centred_test_data_matrix respectively.

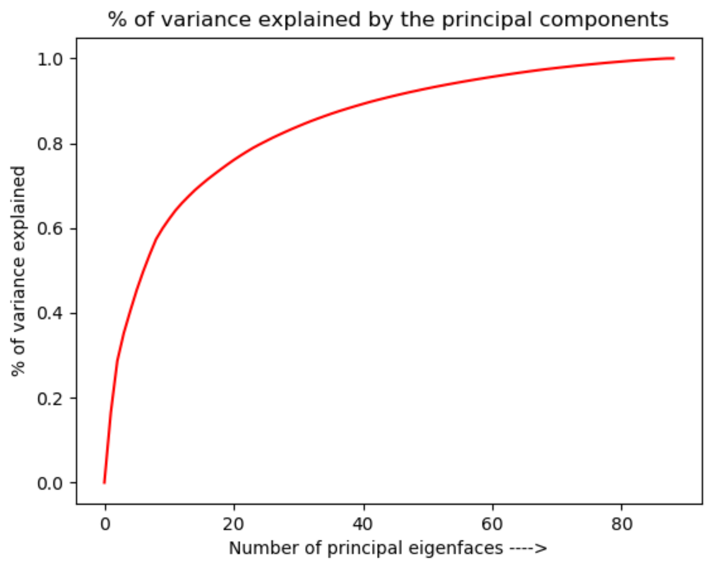
• **Finding Covariance matrix :** Covariance matrix will be 2500*2500 matrix and will be calculated as:
Covariance matrix = $\frac{1}{n}$ (centred_train_data_matrix)^T (centred_train_data_matrix)
, where n is no of images in training set.

• **Finding prinicipal eigenfaces :** Principal eigenfaces will be the eigenvectors corresponding to larger eigenvalues.
The first principal eigenface , when flattened into 1-d array will be the eigenvector corresponding to the largest eigenvalue.
The second principal eigenface , when flattened into 1-d array will be the eigenvector corresponding to the second largest eigenvalue.
and so on in this way, top principal eigenfaces can be calculated.

scipy library is used to calculate the eigenvectors and eigenvalues.

The value of the eigenvalue denotes how much variance is explained by the corresponding eigenvector. In the code, the user defined function "top_k_eigenfaces_calculator" is the function which calculates the number of eigenvectors(let say k eigenvectors) which explains 99.99% of the variance in the training images.

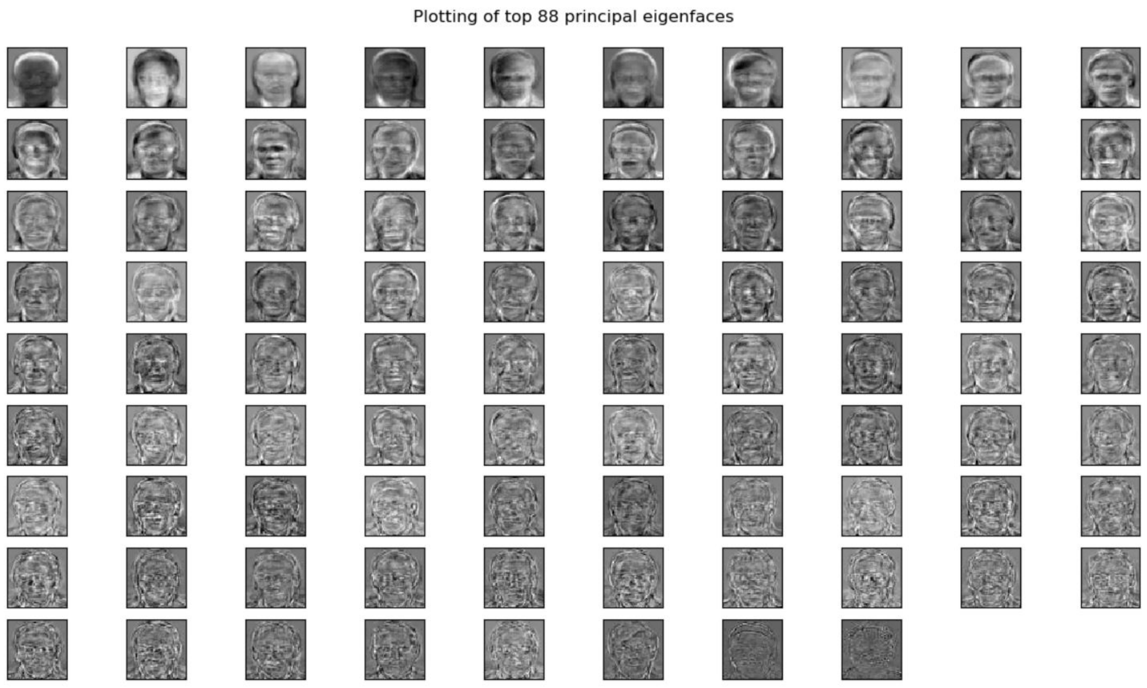
The variances explained by the top principal components looked liked as:



These k eigenvectors(in above figure 88 or 89) will comprises the top principal faces explaining 99.99% of the variance in the training images.

In the code, the user defined function "show_principal_faces" outputs the principal eigenfaces .

I have executed the code for 10 iterations and each time, there is new training and test set. So,here I am showing principal eigenfaces of some random iteration. It look like as:



• **Projection of training data images onto the principal eigenfaces :**
We have got the principal eigenvector(eigenfaces) in the previous step. We have taken top k eigenvectors explaining the 99.99% of the training images variance. So, these k eigenfaces forms eigenspace. Now, the training images are projected upon these new eigenspace.

The projection of a image x_i on principal eigen face e_j is given by $e_j^T x_i$.
$\implies$ one training images is represented by k components , i.e. one component each in direction of each of the top k eigenfaces.
So, now by projecting on eigenspace, we reduces required dimensions of training data matrix to 90*k size matrix.
In the code, the projection of training images onto the principal direction is stored in the matrix "train_projection_matrix" .

• **Projection of testing data images onto the principal eigenfaces :** Similiar to projecting of training images , the test images are also projected onto the eigenspace and the projection of test images are stored into the matrix "test_projection_matrix" and it will be of 10*k dimension.

5 Calculating Accuracy :

We have projection of 10 test images in the "test_projection_matrix" and projection of 90 training images in the "train_projection_matrix".

test_projection_matrix[i] : 1-D array of size k i.e. projection of ith test image onto top k the principal eigenspace.
train_projection_matrix[j] : 1-D array of size k i.e. projection of jth training image onto the top k principal eigenspace.

I have calculated the accuracy by 4 methods. These are as follows :

• **Accuracy by average closest distance :**
average distance of ith test image from person "p" =

$$\frac{\text{sum of distances of projection of i}^{th}\text{test image from all the projection of person "p" in the training images}}{\text{total number of images of person "p" in the training images}}$$

In this way, average distance from each unique person in the training image can be calculated and then that person in the training images set for which the average distance is minimum is the matching person to the test image i according to method 1.

• **Accuracy by closest matching face :**
euclidean distance of projection of ith test image from the projections of all the 90 training images is calculated and the training person corresponding to the least value among these 90 distances is the closest matching person to the test image i according to method 2.

• **Accuracy by KNN i.e. k-nearest matching faces :**
Let's parameter of KNN is 3, i.e. we are looking for 3 closest matching faces in the training images. For a test image i, we calculate euclidean distance of projection of ith test image from the projections of all the 90 training images . Then the training images corresponding to first 3 least distances are taken into consideration. If a person occurs 2 or 3 times among these 3 images, then that is the result of KNN , otherwise the person with the least distance is the output of KNN(modified KNN).

• **Accuracy by hybrid of average closest distance and closest matching face :**
This method is something different. In this method, I have taken consideration the result of 1st method and the 2nd method.
Here 2 cases arises:
1). If the matching face according to method 1 and method 2 are same :
In this case, this method also gives the same matching face as output.

2).If the matching face according to method 1 and method 2 are not same :
Let for test image i, method 1 output matching person as "x" and method 2 output matching person as "y".
Now, the average distance of test image i from the both person "x" and "y" is calculated (as done in method 1).
If the difference between average distance of them is very less (let say , differ by $< 5\%$), then this method output closest matching image as "y".
The motto behind this method is that if there is not significant difference between the average distance of the test image from the two outputs ("x" and "y"), then instead of making decision on the basis of average distance(method 1), we should make decision on the basis of closest matching face(i.e. by method 2), because they both ("x" and "y") are close to test image i because difference in average distance is less($< 5\%$).

6 Results :

On 10 iteration of the whole code : The accuracy of all the 4 methods comes out to be as :

• Accuracy by average distance case (method 1)	: [0.8, 0.9, 0.7, 0.9, 0.6, 0.9, 1.0, 0.9, 0.7, 0.9]
• Accuracy by closest matching face (method 2)	: [0.7, 0.8, 0.7, 1.0, 0.5, 0.9, 0.9, 0.9, 0.8, 0.8]
• Accuracy by KNN (method 3)	: [0.7, 0.8, 0.7, 1.0, 0.6, 0.9, 1.0, 0.9, 0.8, 0.9]
• Accuracy by Hybrid case (method 4)	: [0.8, 0.9, 0.9, 1.0, 0.7, 0.9, 0.9, 0.9, 0.7, 1.0]

If we calculate the average accuracy , then these are as:

• Average Accuracy by average distance case (method 1)	= 0.8300000000000001
• Average Accuracy by closest matching face (method 2)	= 0.8000000000000002
• Average Accuracy by KNN (method 3)	= 0.8300000000000001
• Average Accuracy by Hybrid case (method 4)	= 0.8700000000000001

7 Conclusion :

The following conclusion can be made:
1). Almost top 88 or 89 eigenvectors were able to capture the 99.99% of the variance in the training data. So, all the images whether training or test images were projected upon eigenspace formed by 88 or 89 eigenfaces.
Hence, each image can be projected using only 88 or 89 features , while initially , all the images had 2500 features. So, the dimensionality of the image is reduced by the PCA .

2). All the method tends to give pretty good accuracy , but out of them ,the hybrid method seems to be best with accuracy above 85%. Also, if we repeat the experiment, the accuracy remains above 80% most of the time.

Hence, we had successfully implemented facial recognition using PCA!!!

Library/modules used in code :

- skimage : reading the images from the folder "dataset".
- PIL : reading the images from the folder "dataset".
- matplotlib : For figure plotting purpose.
- scipy.linalg : For finding eigenvalues and eigenvectors.
- math : For ceil,pow etc. math functions.
- random : For random sampling of images into training and test images.
- os : For interacting with operating system.

Tips for executing Code :

"dataset" folder contains the required images for the project .
"main.py" is the python file containing the code for facial recognition.
keep the dataset folder and code "main.py" under the same directory and then execute the "main.py" .
There is additional folder "preprocessing codes" which contain the codes, I used for preprocessing the images.

After loading the images from this folder , they are randomly sampled into training and test images during the execution.

Functions used in the code:
- train_test_listmaker :

Argument: None
Return value : 2 lists -train_image_list and test_image_list. train_image_list contains the name of images in training images and test_image_list contain the name of images in testing images. These 2 lists are generating by random sampling.

- data_matrix_maker :

Argument: a list containing name of images.
Return value: a 2-D matrix of n*d size, where n represents the number of images(names) in the argument list and d is the size of the image .

- average_image_maker :

Argument: a 2-D matrix of size n*d, where n is no of training images and d is size of a training image.
What it outputs: prints average face.
Return value: a 1-D array of size d(size of a image) . This array is the flattened form (1-D) of average face.

- centred_matrix_maker :

Argument: a 2-D matrix(which is to be centred , either training data matrix or testing data matrix) and average image(i.e. average face).
Return value: a 2-D matrix of size n*d whose every row (i.e every image) is centred .

- covariance_matrix_maker :

Argument: centred training data matrix .It will be of size n*d, where n is number of images in training list and d is size of each image.

Return value: d*d 2-dimensional covariance matrix.

- eigen_value_and_vector_maker :
Argument: d*d covariance matrix.
Return value: a list of eigenvalues sorted in descending order and a d*d array of eigenvectors where each row represents a eigenvector and they are arranged according to eigenvalue list.

- top_k_eigenfaces_calculator :
Argument: d*d eigenvalues and eigenvectors.
What it outputs : a graph representing the variance explained by the principal eigenfaces.
Return value: a number representing the number of eigenfaces required to explain 99.99% of the variance in the training images.

- projection_matrix_maker :
Argument: n*d data matrix(centred_train_data_matrix or centred_test_data_matrix) , eigenvector matrix and top_k_eigenfaces value.
Return value: a n*k matrix which is the projection matrix formed by taking projection of the data matrix onto the top k principal eigenfaces.

- show_principal_faces :
Argument: eigenvectors and top_k_eigenfaces value.
What it outputs : Prints the top k eigenfaces.

- accuracy_average_closest_distance :
Argument: test_projection_matrix, train_projection_matrix , test_image_list, train_image_list and top_k_eigenfaces value.
Return value : no of correct prediction and prediction array.

- accuracy_closest_person :
Argument: test_projection_matrix, train_projection_matrix , test_image_list, train_image_list and top_k_eigenfaces value.
Return value : no of correct prediction and prediction array.

- accuracy_knn :
Argument: test_projection_matrix, train_projection_matrix , test_image_list, train_image_list , top_k_eigenfaces value and n_nearest_neighbor i.e. kNN parameter.
Return value : no of correct prediction and prediction array.

- accuracy_hybrid :
Argument: test_projection_matrix, train_projection_matrix , test_image_list, train_image_list, top_k_eigenfaces value , prediction array of accuracy_average_distance_, prediction array of accuracy_closest_person_.
Return value : no of correct prediction and prediction array.