Facial Recognition using Eigenfaces

Introduction

Face recognition has become easier due to recent advancements in machine learning, but in the past, researchers employed various methods and skills to enable computers to identify individuals. One such method that showed moderate success in the early days was eigenface, which uses linear algebra techniques.

This project will demonstrate the creation of a basic face recognition system using simple linear algebra techniques, specifically principal component analysis.

Through this project, we will learn:

- The development of eigenface technique
- How to use principal component analysis to extract characteristic images from an image dataset
- How to express any image as a weighted sum of the characteristic images
- How to compare the similarity of images from the weight of principal components

Image and Face recognition

Images in computers are represented as matrices of pixels, where each pixel is assigned a numerical value representing its color. It is natural to question whether computers can interpret and understand the content of images, and if so, whether matrix mathematics can be used to describe the logic. To simplify this task, researchers have focused on identifying human faces within images. An early attempt involved treating the matrix as a high-dimensional dataset and extracting a lower-dimensional information vector from it to recognize the person. This method was necessary in the past due to limited computing power and memory. However, by developing techniques to compress images into smaller sizes, researchers can now compare whether two images portray the same human face, even if the images are not identical.

In 1987, Sirovich and Kirby published a paper proposing that all images of human faces could be represented as a weighted combination of a small number of "key images". These key images, referred to as "eigenpictures," are the eigenvectors of the covariance matrix of the mean-adjusted images of human faces. In the paper, they outlined the principal component analysis algorithm for processing the matrix of face images and obtaining the weights used in the weighted sum, which correspond to the projection of the face image onto each eigenpicture.

In 1991, Turk and Pentland introduced the term "eigenface," building upon the work of Sirovich and Kirby. They used the weights and eigenpictures as distinctive features for recognizing faces. Turk and Pentland's paper presented a memory-efficient method for computing the eigenpictures, as well as an algorithm for operating the face recognition system, including incorporating new faces and integrating it with a video capture system. The paper also highlighted that the concept of eigenface could aid in reconstructing partially obstructed images.

Overview of Eigenface

Before we proceed with the code, let's provide an overview of the steps involved in utilizing eigenface for face recognition, and highlight how some basic linear algebra techniques can aid in this task.

Let's suppose that we have a collection of grayscale images of human faces, all in the same pixel dimension (e.g., all are R×C grayscale images). If we have M distinct images and convert each image into a vector of L=R×C pixels, we can represent the entire dataset as an L×M matrix (which we will refer to as matrix A). Each element in this matrix corresponds to the grayscale value of a particular pixel.

Principal component analysis (PCA) can be utilized on any matrix, resulting in a set of vectors known as the principal components. Each principal component has the same length as the number of columns in the original matrix. The principal components derived from the same matrix are mutually orthogonal, meaning that the dot product of any two vectors is zero. Thus, the principal components form a vector space in which each column in the matrix can be expressed as a linear combination (i.e., a weighted sum) of the principal components.

Recalling that principal component analysis (PCA) can be applied to any matrix, and the result is several vectors called the principal components. Each principal component has the length same as the column length of the matrix. The different principal components from the same matrix are orthogonal to each other, meaning that the vector dot-product of any two of them is zero. Therefore, the various principal components constructed a vector space for which each column in the matrix can be represented as a linear combination (i.e., weighted sum) of the principal components.

The way it is done is to first take C=A-a where a is the mean vector of the matrix A. So, C is the matrix that subtracts each column of A with the mean vector a. Then the covariance matrix is

$$S=C.C^T$$

from which we find its eigenvectors and eigenvalues. The principal components are these eigenvectors in decreasing order of the eigenvalues. Because matrix S is a L×L matrix, we may consider to find the eigenvectors of a M×M matrix $C^T.C$ instead as the eigenvector v for $C^T.C$ can be transformed into eigenvector u of $C.C^T$ by u=C.v,, except we usually prefer to write u as normalized vector (i.e., norm of u is 1).

The physical meaning of the principal component vectors of A, or equivalently the eigenvectors of $S=C.C^T$, is that they are the key directions that we can construct the columns of matrix A. The relative importance of the different principal component vectors can be inferred from the corresponding eigenvalues. The greater the eigenvalue, the more useful (i.e., holds more information about A) the principal component vector. Hence, we can keep only the first K principal component vectors. If matrix A is the dataset for face pictures, the first K principal component vectors are the top K most important “face pictures”. We call them the eigenface picture.

To determine how closely a face picture is related to an eigenface picture, we can subtract the average face picture from the original face picture and then take the dot product between the resulting vector and the eigenface picture. If the face picture is not at all related to the eigenface, then the dot product will be zero. For the K eigenfaces, we can calculate K dot products for each given face picture. The resulting weights of the face picture with respect to the eigenfaces can be presented as a vector.

Conversely, if we have a weight vector, we can add up each eigenfaces subjected to the weight and reconstruct a new face. Let's denote the eigenfaces as matrix F, which is a L×K matrix, and the weight vector w is a column vector. Then for any w we can construct the picture of a face as

$$Z=F.w$$

which z is resulted as a column vector of length L. Because we are only using the top K principal component vectors, we should expect the resulting face picture is distorted but retained some facial characteristic.

Since the eigenface matrix is constant for the dataset, a varying weight vector w means a varying face picture. Therefore, we can expect the pictures of the same person would provide similar weight vectors, even if the pictures are not identical. As a result, we may make use of the distance between two weight vectors (such as the L2-norm) as a metric of how two pictures resemble.

Implementing Eigenface

In the upcoming code implementation, we will utilize the concepts of eigenface using numpy and scikit-learn libraries. Additionally, we will make use of OpenCV library to read image files. It is recommended to install the required packages using the pip command.

```
pip install opencv-python
```

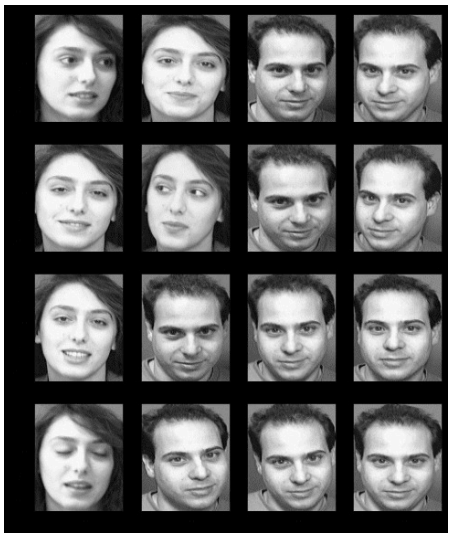
We will be using the ORL Database of Faces, which contains pictures of 40 individuals, with 10 pictures per person, resulting in a total of 400 pictures. The dataset can be downloaded as a 4MB zip file from Kaggle. Once downloaded, we can assume that the file is in the local directory and named as attface.zip.

https://www.kaggle.com/kasikrit/att-database-of-faces/download

There are two ways to access the pictures: we can either extract the pictures from the downloaded zip file, or we can use Python's zipfile package to read the contents of the zip file directly.

The PGM format is a format for grayscale images. We extract each PGM file into a sequence of bytes by calling image.read() and convert it into a numpy array of bytes. Then, we utilize OpenCV to convert the sequence of bytes into an array of pixels using cv2.imdecode(). OpenCV will automatically detect the file format. We store each image in a Python dictionary called "faces" for future use.

Here we can take a look on these pictures of human faces, using matplotlib.



We can also find the pixel size of each picture.

Face image shape: (112, 92)

The pictures of faces are identified by their file name in the Python dictionary. We can take a peek on the filenames:

```
1...
2print(list(faces.keys())[:5])
1['s1/1.pgm', 's1/10.pgm', 's1/2.pgm', 's1/3.pgm', 's1/4.pgm']
```

and therefore, we can put faces of the same person into the same class. There are 40 classes and totally 400 pictures:

```
1...
2classes = set(filename.split("/")[0] for filename in faces.keys())
3print("Number of classes:", len(classes))
4print("Number of pictures:", len(faces))
```

```
Number of classes: 40
Number of pictures: 400
```

To demonstrate how eigenface can be used for recognition, we will reserve some of the pictures before creating the eigenfaces. Specifically, we will remove all the pictures of one person and one picture of another person from our dataset, which will serve as our test set. The remaining pictures are vectorized and converted into a 2D numpy array.

```
# Take classes 1-39 for eigenfaces, keep entire class 40 and
# image 10 of class 39 as out-of-sample test
facematrix = []
facelabel = []
for key,val in faces.items():
    if key.startswith("s40/"):
        continue # this is our test set
    if key == "s39/10.pgm":
```

```
        continue # this is our test set
    facematrix.append(val.flatten())
    facelabel.append(key.split("/")[0])

# Create facematrix as (n_samples,n_pixels) matrix
facematrix = np.array(facematrix)
```

We can now apply principal component analysis to the dataset matrix. Rather than computing PCA manually, we can utilize the PCA function from scikit-learn library. This function can automatically compute the eigenvectors and eigenvalues of the covariance matrix of the dataset, and we can easily retrieve the principal components, eigenvalues, and mean face vector using the output of the function.

```
...
# Apply PCA to extract eigenfaces
from sklearn.decomposition import PCA

pca = PCA().fit(facematrix)
```

We can identify how significant is each principal component from the explained variance ratio:

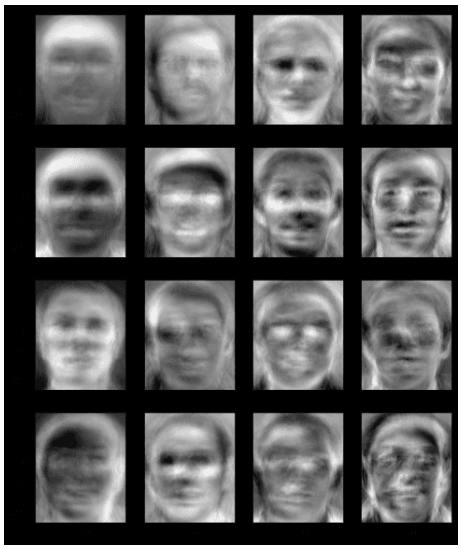
```
...
print(pca.explained_variance_ratio_)
```

```
[1.77824822e-01 1.29057925e-01 6.67093882e-02 5.63561346e-02
 5.13040312e-02 3.39156477e-02 2.47893586e-02 2.27967054e-02
 1.95632067e-02 1.82678428e-02 1.45655853e-02 1.38626271e-02
 1.13318896e-02 1.07267786e-02 9.68365599e-03 9.17860717e-03
 8.60995215e-03 8.21053028e-03 7.36580634e-03 7.01112888e-03
 6.69450840e-03 6.40327943e-03 5.98295099e-03 5.49298705e-03
 5.36083980e-03 4.99408106e-03 4.84854321e-03 4.77687371e-03
...
 1.12203331e-04 1.11102187e-04 1.08901471e-04 1.06750318e-04
 1.05732991e-04 1.01913786e-04 9.98164783e-05 9.85530209e-05
 9.51582720e-05 8.95603083e-05 8.71638147e-05 8.44340263e-05
 7.95894118e-05 7.77912922e-05 7.06467912e-05 6.77447444e-05
 2.21225931e-32]
```

Alternatively, we can decide on a fixed number of principal components, for example 50, to be used as eigenfaces. We can easily extract the eigenfaces from the PCA result and store them as a numpy array. The eigenfaces are represented as rows in a matrix, but we can convert it back to 2D for display purposes. Below, we display some of the eigenfaces to visualize how they look like.

```
...
# Take the first K principal components as eigenfaces
n_components = 50
eigenfaces = pca.components_[:n_components]

# Show the first 16 eigenfaces
fig, axes = plt.subplots(4,4,sharex=True,sharey=True,figsize=(8,10))
for i in range(16):
    axes[i%4][i//4].imshow(eigenfaces[i].reshape(faceshape), cmap="gray")
plt.show()
```



Looking at the image, we can observe that eigenfaces appear as blurred faces. However, each eigenface still contains some unique facial features that can be utilized to construct a face image.

Since our goal is to build a face recognition system, we first calculate the weight vector for each input picture:

```
...
# Generate weights as a KxN matrix where K is the number of eigenfaces and N the number of samples
weights = eigenfaces @ (facematrix - pca.mean_).T
```

The above code is using matrix multiplication to replace loops. It is roughly equivalent to the following:

```
...
weights = []
for i in range(facematrix.shape[0]):
    weight = []
    for j in range(n_components):
        w = eigenfaces[j] @ (facematrix[i] - pca.mean_)
        weight.append(w)
    weights.append(weight)
```

At this point, we have finished building our face recognition system. We trained the eigenface model using pictures of 39 individuals, and we set aside one picture for everyone to use as a test. We will use the test pictures to see if our system can correctly recognize the faces.

```
...
# Test on out-of-sample image of existing class
query = faces["s39/10.pgm"].reshape(1,-1)
query_weight = eigenfaces @ (query - pca.mean_).T
euclidean_distance = np.linalg.norm(weights - query_weight, axis=0)
best_match = np.argmin(euclidean_distance)
print("Best match %s with Euclidean distance %f" % (facelabel[best_match], euclidean_distance[best_match]))
# Visualize
fig, axes = plt.subplots(1,2,sharex=True,sharey=True,figsize=(8,6))
axes[0].imshow(query.reshape(faceshape), cmap="gray")
axes[0].set_title("Query")
```

```
axes[1].imshow(facematrix[best_match].reshape(faceshape), cmap="gray")
axes[1].set_title("Best match")
plt.show()
```

In the previous code snippet, we first calculated the mean-subtracted vector of the test image by subtracting the average vector obtained from the PCA result. Then we calculated the projection of this mean-subtracted vector onto each eigenface and obtained the weights for this image. Next, we compared the weight vector of this test image to that of each existing image and found the one with the smallest Euclidean distance, which we considered as the best match. As we can see, the system was able to successfully identify the closest match in the same class as the test image.

```
Best match s39 with Euclidean distance 1559.997137
```

and we can visualize the result by comparing the closest match side by side:

Next, we can repeat the process using the test image of the 40th person which was held out from the PCA. It is expected that the recognition system will fail because it has not been trained with this person's face. Nevertheless, we can observe the magnitude of the distance metric and how incorrect the recognition will be.

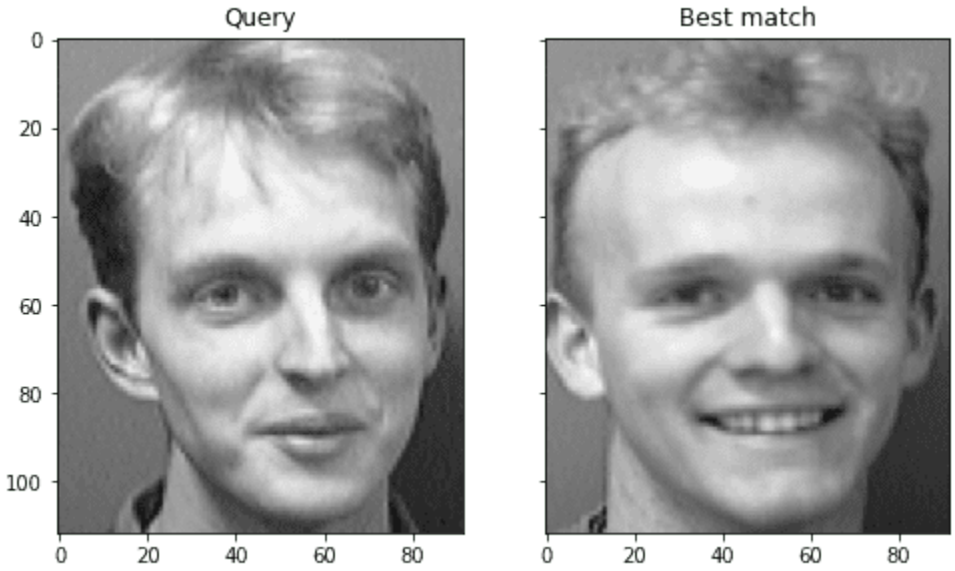
```
...
# Test on out-of-sample image of new class
query = faces["s40/1.pgm"].reshape(1,-1)
query_weight = eigenfaces @ (query - pca.mean_).T
euclidean_distance = np.linalg.norm(weights - query_weight, axis=0)
best_match = np.argmin(euclidean_distance)
print("Best match %s with Euclidean distance %f" % (facelabel[best_match], euclidean_distance[best_match]))
# Visualize
fig, axes = plt.subplots(1,2,sharex=True,sharey=True,figsize=(8,6))
axes[0].imshow(query.reshape(faceshape), cmap="gray")
axes[0].set_title("Query")
axes[1].imshow(facematrix[best_match].reshape(faceshape), cmap="gray")
axes[1].set_title("Best match")
```

```
plt.show()
```

We can see that it's best match has a greater L2 distance:

```
Best match s5 with Euclidean distance 2690.209330
```

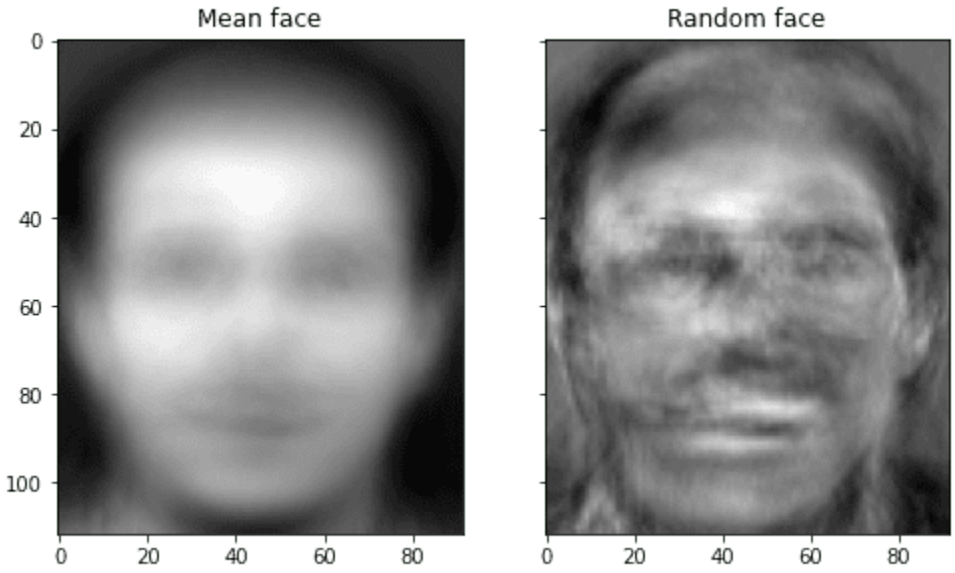
but we can see that the mistaken result has some resemblance to the picture in question:



The paper by Turk and Pentland recommends setting up a threshold for the L2 distance. If the distance of the best match is less than the threshold, we consider the face to be recognized as the same person. If the distance is greater than the threshold, we consider the picture to be of someone we have never seen before, even if the best match can be found numerically. In this case, we can include this as a new person in our model by remembering this new weight vector.

We can perform an additional step in this process, which is generating new faces using the eigenfaces. However, the resulting faces may not look very realistic. Below, we generate a new face using a random weight vector and display it alongside the "average face".

```
...
# Visualize the mean face and random face
fig, axes = plt.subplots(1,2,sharex=True,sharey=True,figsize=(8,6))
axes[0].imshow(pca.mean_.reshape(faceshape), cmap="gray")
axes[0].set_title("Mean face")
random_weights = np.random.randn(n_components) * weights.std()
newface = random_weights @ eigenfaces + pca.mean_
axes[1].imshow(newface.reshape(faceshape), cmap="gray")
axes[1].set_title("Random face")
plt.show()
```



Eigenface is a surprisingly effective model given its simplicity. However, Turk and Pentland conducted tests under various conditions and found that its accuracy was 96% on average with light variation, 85% with orientation variation, and 64% with size variation. Therefore, it may not be very practical as a face recognition system, as pictures can be distorted a lot in the principal component domain when zooming in and out. As a result, the modern alternative is to use convolutional neural networks, which are more tolerant to various transformations.

Putting everything together, the following is the complete code:

```
import zipfile
import cv2
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

# Read face image from zip file on the fly
faces = {}
with zipfile.ZipFile("attface.zip") as facezip:
    for filename in facezip.namelist():
        if not filename.endswith(".pgm"):
            continue # not a face picture
        with facezip.open(filename) as image:
            # If we extracted files from zip, we can use cv2.imread(filename) instead
            faces[filename] = cv2.imdecode(np.frombuffer(image.read(), np.uint8), cv2.IMREAD_GRAYSCALE)

# Show sample faces using matplotlib
fig, axes = plt.subplots(4,4,sharex=True,sharey=True,figsize=(8,10))
faceimages = list(faces.values())[-16:] # take last 16 images
for i in range(16):
    axes[i%4][i//4].imshow(faceimages[i], cmap="gray")
print("Showing sample faces")
plt.show()

# Print some details
faceshape = list(faces.values())[0].shape
```

```
print("Face image shape:", faceshape)

classes = set(filename.split("/")[0] for filename in faces.keys())
print("Number of classes:", len(classes))
print("Number of images:", len(faces))

# Take classes 1-39 for eigenfaces, keep entire class 40 and
# image 10 of class 39 as out-of-sample test
facematrix = []
facelabel = []
for key,val in faces.items():
    if key.startswith("s40/"):
        continue # this is our test set
    if key == "s39/10.pgm":
        continue # this is our test set
    facematrix.append(val.flatten())
    facelabel.append(key.split("/")[0])

# Create a NxM matrix with N images and M pixels per image
facematrix = np.array(facematrix)

# Apply PCA and take first K principal components as eigenfaces
pca = PCA().fit(facematrix)

n_components = 50
eigenfaces = pca.components_[:n_components]

# Show the first 16 eigenfaces
fig, axes = plt.subplots(4,4,sharex=True,sharey=True,figsize=(8,10))
for i in range(16):
    axes[i%4][i//4].imshow(eigenfaces[i].reshape(faceshape), cmap="gray")
print("Showing the eigenfaces")
plt.show()

# Generate weights as a KxN matrix where K is the number of eigenfaces and N the number of samples
weights = eigenfaces @ (facematrix - pca.mean_).T
print("Shape of the weight matrix:", weights.shape)

# Test on out-of-sample image of existing class
query = faces["s39/10.pgm"].reshape(1,-1)
query_weight = eigenfaces @ (query - pca.mean_).T
euclidean_distance = np.linalg.norm(weights - query_weight, axis=0)
best_match = np.argmin(euclidean_distance)
print("Best match %s with Euclidean distance %f" % (facelabel[best_match], euclidean_distance[best_match]))
# Visualize
fig, axes = plt.subplots(1,2,sharex=True,sharey=True,figsize=(8,6))
axes[0].imshow(query.reshape(faceshape), cmap="gray")
axes[0].set_title("Query")
axes[1].imshow(facematrix[best_match].reshape(faceshape), cmap="gray")
axes[1].set_title("Best match")
plt.show()

# Test on out-of-sample image of new class
query = faces["s40/1.pgm"].reshape(1,-1)
query_weight = eigenfaces @ (query - pca.mean_).T
euclidean_distance = np.linalg.norm(weights - query_weight, axis=0)
best_match = np.argmin(euclidean_distance)
```

```
print("Best match %s with Euclidean distance %f" % (facelabel[best_match], euclidean_distance[best_match]))
# Visualize
fig, axes = plt.subplots(1,2,sharex=True,sharey=True,figsize=(8,6))
axes[0].imshow(query.reshape(faceshape), cmap="gray")
axes[0].set_title("Query")
axes[1].imshow(facematrix[best_match].reshape(faceshape), cmap="gray")
axes[1].set_title("Best match")
plt.show()
```

Mathematical Concepts Explained

Step 1: Data Preparation

Collect a set of grayscale images of human faces, all with the same pixel dimensions.

Convert each image into a vector of length L, where L is the total number of pixels in each image.

Represent the entire dataset as an L×M matrix A, where M is the number of distinct images and each column corresponds to a vectorized image.

Step 2: Mean Subtraction

Compute the mean vector a by averaging all the columns of matrix A: a = (1/M) * ΣA.

Subtract the mean vector from each column of matrix A to obtain a new matrix C: C = A - a.

Step 3: Covariance Matrix Calculation

Compute the covariance matrix S by multiplying C with its transpose: S = C * C^T.

Step 4: Eigenvalue and Eigenvector Computation

Find the eigenvectors and eigenvalues of the covariance matrix S. Let's denote the eigenvectors as U and the eigenvalues as λ.

The eigenvectors U and eigenvalues λ satisfy the equation: S * U = U * Λ, where Λ is a diagonal matrix containing the eigenvalues.

Step 5: Eigenface Selection

Sort the eigenvectors U in decreasing order of their corresponding eigenvalues λ.

Select the first K eigenvectors (principal components) with the highest eigenvalues.

These K eigenvectors form the eigenface matrix F, which is an L×K matrix: F = [u_1, u_2, ..., u_K], where u_i represents the i-th eigenvector.

Step 6: Face Representation and Recognition

To determine the similarity between a face image and the eigenfaces, subtract the mean face vector a from the original face image: x = x' - a, where x' represents the original face image.

Calculate the dot product between the resulting vector x and each eigenface in matrix F: w = F^T * x.

These dot products represent the weights or coefficients of the original face image with respect to the eigenfaces.

Step 7: Reconstruction

Given a weight vector w (representing coefficients), reconstruct a new face image by summing the weighted eigenfaces: reconstructed_face = F * w + a.

Varying the weights allows for generating different variations of the reconstructed face.

Mathematical Notations:

Matrix A: L×M matrix representing the dataset of face images.

Vector a: Mean vector obtained by averaging all columns of matrix A.

Matrix C: L×M matrix obtained by subtracting the mean vector a from each column of matrix A: C = A - a.

Matrix S: Covariance matrix computed as the product of matrix C and its transpose: S = C * C^T.

Matrix U: L×L matrix containing the eigenvectors of the covariance matrix S.

Matrix Λ: Diagonal matrix containing the eigenvalues of the covariance matrix S.

Matrix F: L×K matrix containing the selected K eigenvectors (eigenfaces) with the highest eigenvalues.

Vector x: Vector obtained by subtracting the mean face vector a from the original face image: x = x' - a.

Vector w: Weight vector obtained by calculating the dot product between the vector x and each eigenface in matrix F: w = F^T * x.

Reconstructed Face: New face image obtained by summing the weighted eigenfaces: reconstructed_face = F * w + a.

By following these steps and using the corresponding mathematical notations, eigenfaces can be used for various tasks in face recognition, including:

Face Identification:

Given a new face image, extract its feature vector by subtracting the mean face vector and projecting onto the eigenfaces: x = x' - a, w = F^T * x.

Compare the weight vector of the new face with the weight vectors of known faces in a database using similarity measures like Euclidean distance or cosine similarity.

Identify the face by finding the closest match based on the similarity measure.

Face Verification:

For face verification, a threshold is set to determine whether two face images belong to the same person or not.

Calculate the similarity between the weight vectors of the two face images and compare it with the threshold.

If the similarity exceeds the threshold, the images are considered a match; otherwise, they are deemed different individuals.

Face Reconstruction:

Given a weight vector w, reconstruct the original face image by adding up the weighted eigenfaces: reconstructed_face = F * w + a.

This allows for visualizing and generating new face images based on different combinations of eigenfaces and weights.

Face Expression Recognition:

Extend the eigenface approach to analyze and recognize facial expressions.

Train a separate set of eigenfaces for each facial expression, and use the weight vectors to classify and recognize different expressions.

Face Pose Estimation:

Extend the eigenface approach to estimate the pose (orientation) of a face in an image.

Incorporate additional eigenfaces that capture variations in face pose, such as tilt, rotation, and scale.

Face Reconstruction from Incomplete Data:

Eigenfaces can be used to reconstruct a complete face image from partial or noisy data.

By utilizing the weights associated with the eigenfaces, missing or corrupted pixels in a face image can be filled in or restored.

These are some of the applications where eigenfaces can be employed for face recognition. The linear algebra techniques involved, such as mean subtraction, covariance matrix calculation, eigenvector decomposition, and reconstruction, form the foundation of the eigenface method and enable its practical implementation.

References

- L. Sirovich and M. Kirby (1987). "Low-dimensional procedure for the characterization of human faces". *Journal of the Optical Society of America A*. 4(3): 519–524.
- M. Turk and A. Pentland (1991). "Eigenfaces for recognition". *Journal of Cognitive Neuroscience*. 3(1): 71–86.
- Introduction to Linear Algebra, Fifth Edition, 2016.
- sklearn.decomposition.PCA API
- matplotlib.pyplot.imshow API
- Eigenface on Wikipedia