Wolfy Fiorini
MTH 3130
12/07/22

Facial Recognition Using Eigenfaces

1. **Abstract -** Facial recognition is a relevant and useful tool in today's world. However, constructing software that can reliably distinguish individuals faces efficiently is difficult and is the subject of ongoing research. One technique, called Principal Component Analysis, provides a simple yet powerful method for implementing facial recognition through "eigenfaces" that capture variations in features between faces.

   This paper develops the mathematical concepts needed to implement the method and examines its practical application through software. Through the images and results produced by the software, it is shown that eigenfaces can correctly identify 90-95% of faces and does so with exceeding efficiency.

2. **Introduction** – Facial recognition software is used for a variety of applications today, including security, law enforcement, multimedia, video transmission and more (Nayeem and Ravi, 59). This paper aims to replicate a method of facial recognition used in the paper written by Matthew Turk and Alex Pentland (1991), using a dataset of 400 faces of 40 individuals (FACES), in software. Using Principal Component Analysis, hereafter referred to as PCA, to reduce the dimension of the face images and speed up computation, and eigenfaces that capture prominent features and variations in the training faces, the software will efficiently process an image of a face and determine whether the face matches any the software was previously trained on.

   This paper will first define and develop the mathematical tools, symbols, and methods used, providing an in-depth explanation of each and how they will be applied. Following this will be a summary of the processes used to implement the technique, and then a showcase of the images used in and produced by the software implementation. Next, there will be a detailed examination of the data used in the application, using specific examples, as well as an assessment of its performance and results. Finally, there will be a discussion of the results and discoveries made, finishing with the possibilities of how parts of this technique may be explored further in the future.

3. Mathematical Formulation

3.1 Representing images - In the method of facial recognition used in this paper, black and white images of faces that are N pixels tall and N pixels wide are used as training faces with the intention that any future face images can be compared against these faces to determine whether there are any authorized faces that are similar enough to the new image to be considered a match. We can treat these images as M square matrices of size $N \times N$, where each entry in each matrix represents a single pixel with intensity values ranging from 0 (black) to 255 (white). As outlined by Turk and Pentland, to use PCA the matrices need to be flattened

into vectors by concatenating each row together into *M* vectors of dimension N^2, represented as $\Gamma_1, \Gamma_2, \Gamma_3, \ldots, \Gamma_M$.

With these vectors, we can define a vector, Ψ, that represents the average of all the face vectors, calculated as follows:

$$\Psi = \frac{1}{M}\sum_{i=1}^{M}\Gamma_i$$

With Ψ, we can calculate how much each face differs from the average with M vectors ϕ, called difference vectors, calculated by $\phi_i = \Gamma_i - \Psi$ for *i = 1, 2,…,M*. We arrange these vectors into an $N^2 \times M$ matrix *A* whose columns are the vectors $(\phi_1, \phi_2, \ldots, \phi_M)$. Next, the variation of each value in any given difference vector ϕ in relation to the other values in the other vectors must be calculated. To do this, we find the covariance of each vector.

3.2 Covariance – In the context of statistics, covariance measures how strongly two random variables are related to one another by measuring how much one differs from the mean when the other changes. Relating this back to our application, we can view each pixel value as a discrete random variable, with the difference vectors ϕ representing how much each image differs from the mean. As shown in an article by Jeremy Kun, we can calculate how strongly the values in all the difference vectors vary in relation to one another by finding the covariance matrix C. This is done simply by multiplying each difference vector by its transpose:

$$C = \frac{1}{M}\sum_{i=1}^{M}\phi_i{\phi_i}^T = AA^T$$

The covariance matrix gives us an idea of which pixels are most closely related to each other. example, a high positive value at $C_{i,j}$ and $C_{j,i}$ tells us that the pixel i and pixel j in every image tend to vary from their mean by a similar amount. This also highlights the fact that C is a symmetrical matrix.

3.3 PCA - According to Slavković and Jevtić, PCA reduces the dimensionality of data from one dimension to a lower dimension (in the case of this application, from dimension N^2 to dimension M) by finding projection vectors that retain the most information about the original data (121). Furthermore, Henderson posits that we can find these projection vectors by finding the eigenvalues λ and eigenvectors *v* of C (4). To do this, we solve the following equation:

$$Cv = AA^Tv = \lambda v$$

However, AA^T is of size $N^2 \times N^2$, and thus this will be very computationally expensive to solve for any N larger than some small value, so we need to find a way to reduce the number of calculations needed.

As outlined by Turk and Pentland, there are only *M-1* nonzero eigenvalues associated with any $N \times N$ symmetrical matrix (74), and thus the same applies to the covariance matrix. In the case of the covariance matrix, an eigenvalue of zero represents a feature of the training faces that doesn't vary meaningfully from the average, and thus can be ignored. We can

solve for the nonzero eigenvalues by first looking at the eigenvectors v_i of the $M \times M$ matrix, $L = A^T A$, and applying the properties of linear algebra as follows:

$$A^T A v_i = \lambda v_i$$
$$A(A^T A)v_i = \lambda A v_i$$
$$(AA^T)Av_i = \lambda A v_i$$
$$CAv_i = \lambda A v_i$$

As shown above, we can find the eigenvalues λ and eigenvectors Av_i of C by finding the M eigenvalues and eigenvectors, v_i, of L. Finally, we can find *M'* orthonormal vectors $f_i,\ i = 1,2,\dots,M'$, by calculating v_i, keeping the *M'* eigenvectors with the highest eigenvalues, calculating Av_i, and normalizing the result. We want the eigenvectors with the highest eigenvalues, as a higher eigenvalue indicates the vector captures a feature that heavily varies between the faces in the training set. Fortunately, around 90% of the variance in the faces is captured by the first 5% to 10% of eigenvalues (Slavković and Jevtić, 121).

Calculating Av_i is equivalent to using the eigenvectors' values as coefficients to form a linear combination of the training faces as outlined by Kun:

$$f_i = v_{i1}\phi_1 + v_{i2}\phi_2 + \dots + v_{iM}\phi_M,\ \ i = 1,2,\dots,M'$$

3.4 Eigenfaces – We call each eigenvector in the set f an eigenface. These vectors form an orthonormal basis for a subspace of $\mathbb{R}^M$ in $\mathbb{R}^{N^2}$, providing a way to project any face image, Γ, from a vector in $\mathbb{R}^{N^2}$ to a vector in the eigenface space in $\mathbb{R}^M$. We can do this by calculating the difference vector, ϕ, of Γ, and then projecting ϕ onto each of the *M'* eigenfaces that forms the eigenface space F as follows:

$$proj_F \phi = w_1 f_1 + w_2 f_2 + \cdots + w_{M'} f_{M'}$$

We will call the coefficients in the equation above the weights, w_i, of each eigenface's contribution to forming the image Γ. Since the eigenfaces form an orthonormal basis, the weights are simply the dot product of ϕ and f_i, which is equivalent to the matrix multiplication product of ${f_i}^T$ and ϕ:

$$w_i = {f_i}^T \phi, \quad i = 1,2,\dots,M',$$

creating what's known as a pattern vector, Ω_i, defined as ${\Omega_i}^T = [w_1, w_2, \dots, w_{M'}]$. An added feature of this vector is that it allows us to reconstruct the initial face vector as follows:

$$\Gamma = \Psi + \sum_{i=1}^{M'} (w_i f_i)$$

3.5 Facial recognition – Following the process explained by Turk and Pentland (76 - 77), to recognize faces, we first need to project each of the training faces onto the eigenface space by calculating the pattern vector, Ω^T, for each of our M training faces. To allow for recognition of a face in different conditions, we can create a single pattern vector by adding together pattern vectors of several images of the same person, then averaging their values and storing them in a single vector called a face class. This allows for better recognition of an

authorized person's face with different expressions, clothing, or lighting. We create k face classes, one for each of the k unique individuals in our set of training faces.

To see if an input image, Γ, is recognized, we find the pattern vector, Ω_Γ, for the input and compare it with each of the face classes. Recall that each value in a pattern vector represents the weight of an individual eigenface's contribution towards the face image. If two pattern vectors are very similar, then this means that the faces share the feature captured by the eigenface. If they are similar enough, then there is a high chance that the input image matches one of the individuals in our training sample.

We can leverage the fact that the pattern vector created from the input, as well each of the face classes, are vectors that represent points in $\mathbb{R}^M$. Due to this, we can calculate the Euclidean distance ϵ_k from the point described by the input pattern vector Ω_Γ to each of our face classes as follows:

$$\epsilon_k = \| \Omega_\Gamma - \Omega_k \|^2,$$

where Ω_k represents the *k*th face class. By choosing the face class that results in the minimum value of ϵ_k, we are choosing the face class that is the most similar in terms of important features as the input. Using a chosen threshold θ, we can then determine whether the input matches the face class *k* if ϵ_k is below the threshold, meaning that it is close enough to face class *k* to be considered recognizable as the individual whose face was used for that class. Slavković and Jevtić advise using a common method to calculate θ, which is to calculate the minimum distance of each individual face class from every other face class, storing these distances in a vector called *rast.* The threshold can then be calculated by multiplying the greatest distance from *rast* by 0.8 (124).

3.6 Assumptions and approximations – To simplify the implementation of the problem, every input image used is assumed to be an image of a face. As a result, the calculation of distance between the image and eigenface space was omitted. Additionally, there will be no preprocessing of images, and each image is assumed to be cropped to represent only the face. It will also be assumed that any of the eigenvalues, eigenvectors, weights, or matrix operations calculated via the Python NumPy library are accurate.

As for approximation, PCA and its implementation here introduces a small margin of error into the calculations. As explained before, projecting data from a higher dimension into a lower dimension means that some information may be lost, and as a result projecting the data back into the higher dimension will never recreate it entirely accurately. Additionally, the choice of how many eigenfaces will be used effects the accuracy in recognition and recreation of images. In other applications, this method is very reliable so this shouldn't make a large difference.

4. Examples and Numerical Results

4.1 Process – To summarize, the implementation of PCA and eigenfaces for face recognition is performed via the following steps:

1. **Load face images –** First, the faces from the dataset being used are loaded and into vectors. This is also where the faces are split by individual, and then individuals are split into training and testing sets.
2. **Generate mean face –** The *M* face vectors from the training set are averaged into a mean vector, Ψ.
3. **Calculate differences –** The difference from each face to the average is found and stored in a difference vector ϕ_i. These are as columns in the $N^2 \times M$ matrix, A.
4. **Find eigenfaces –** *M* eigenfaces are formed by finding the eigenvectors v of the matrix $L = A^T A$, taking the *M'* eigenvectors with the highest eigenvalues, and multiplying them by A. These vectors are then normalized.
5. **Classify training faces –** Each of the *M* faces in the training set are projected onto the subspace formed by the eigenfaces, and weights are for each eigenface. These weights are stored in a pattern vector, Ω^T, and a face class is formed for each individual by taking the average of all the pattern vectors for that individual.
6. **Face recognition –** An input image is turned into a vector and is projected into the eigenface space to form its pattern vector Ω_Γ. The distance ϵ_k from Ω_Γ to each face class is calculated. The minimum value of ϵ_k, is compared with a threshold value, θ. If the value is smaller than the threshold, the face is recognized. Otherwise, the face is unrecognized.

4.2 Dataset – For this application, the Olivetti dataset of 400 face images provided by Sheik Imran were used. Each image is 64 pixels by 64 pixels and represented in grayscale. In this implementation, the first 200 images were used as the set of training images and designated as authorized faces, while the last 200 were used as test images for unauthorized users. The dataset contains 40 individuals. Figure 3 shows some of these individuals.

Each individual has 10 different pictures of their face, each with slightly different lighting, expressions, accessories like glasses, or poses. Figure 4 shows images with slight variations for a single person.

(top) **Figure (3)** – Images of 4 different individuals from the dataset.
(bottom): **Figure (4)** – 6 images from the dataset of the same

4.3 Software implementation – The software implementing the eigenface facial detection was written in Python, using the NumPy library. The code used for the application is contained in an appendix at the end of this paper. NumPy provides everything needed to implement the steps in the process in nearly the exact same way as outlined before, minus the mathematical symbols used, so this paper will not summarize the code to avoid redundancy. Instead, the images from each step in the process will be shown and analyzed, and then data gathered from 60 trial runs will be summarized.

4.4 Face generations - To begin, Figure 5 shows the image Ψ produced by averaging all the faces in the training set. Note the defined eyes, eyebrows, and nose.

Figure (6) – Face images (top) and their mean adjusted counterparts (bottom)

Figure (5) – Mean face image formed from 200 training faces

Since every face has these features in nearly the same position, these are strongly represented in the average. Other features are not as defined since each different face introduces variation with expression or pose.

The next images produced by the software are the difference images ϕ, formed by subtracting the average face from each training face. Figure 6 shows 4 of these faces and their original counterpart. Very dark areas in the image are representative of features that are close to the average, and very light areas are parts that differ heavily from the average face. Interestingly, while the mouths differ quite a bit as expected from looking at the average face, the part that differs the most from the average are the eyes.

The next, and perhaps most important, images produced are the eigenfaces. Recall that each of the eigenfaces is represent different features of the training faces that vary from the average that can be added together to form any face in the training set. Figure 7 shows the 8 eigenfaces with the highest associated eigenvalues. These faces are referred to in several

Figure (7) – Eigenfaces.
(top) - Associated eigenvalues from left to right: 73068.44, 2749.29, 1,344.18, 792.12.
(bottom) – Associated eigenvalues from left to right: 656.15, 540.09, 445.31, 378.94.

papers as “ghost faces”, for reasons that should be clear. When an eigenface has a bright value, this is an area that varies greatly between different faces in the training set. For example, by looking at the face in the top left with the highest eigenvalue, we can see that the features with the most variation across all the faces are the background and eyes.

The final set of images that are of interest are the results of recreating a face from weighted eigenfaces. Since each face can be represented by taking the average face and adding each eigenface with varied weight to it, we can look at the changes that each eigenface makes to get closer to the original image. Figure 8 shows one such recreation at several steps, clearly showing how each eigenface dictates the addition or subtraction of key elements to achieve a result that looks nearly identical to the original image.

Figure 8 – Recreation process of a face from eigenfaces. From left to right: Average, after adding eigenface 1, 4, 5, 6, 10, 16, 21, and 50, original image.

4.5 Trials – 60 trials were run on the dataset using the program. Half of these were performed using the covariance matrix C to calculate the eigenfaces, and the others were run using the matrix L for these calculations. For each matrix, 30 trials were run – 10 each using 10 eigenfaces, 50 eigenfaces, and all 200 eigenfaces, and averaged to determine the most efficient and accurate method of face detection. Each trial consisted of performing the entire process from loading the faces to calculating eigenfaces and classifying the training set. Then, each of the 400 faces were shown to the program, and the program decided whether they were recognized or not. Data was compiled for every trial, shown in figure 8 and 9.

4.6 Trial results – Figure 9 is a table showing the time elapsed for each task in the program when performed with the different matrices and number of eigenfaces. It is obvious that process of PCA massively reduces the time spent performing calculations, as every trial done without PCA took over 37 seconds, while none of the PCA trials took more than 1 second. In fact, in every category besides calculation of the threshold, the PCA trials were much faster. Most of the time in the trials that used the covariance matrix C were spent calculating the eigenvectors for C, as there were $4096^2 = 16{,}777{,}216$ calculations that needed to be done versus only $200^2 = 40{,}000$ when using matrix L. This extra time was wasted, because as previously mentioned most eigenvectors in C have eigenvalues of zero and thus are useless for the intended application.

	Eigenfaces used	Calculating eigenfaces	Classifying training set	Calculating threshold	Detecting faces	Total
Covariance Matrix (4096x4096)	10	3696.03 ms	5.21 ms	0.29 ms	15.48 ms	3717.19 ms
	50	3731.20 ms	31.99 ms	0.29 ms	76.31 ms	3840.11 ms
	200	3818.06 ms	1148.70 ms	0.29 ms	226.72 ms	4160.05 ms
L Matrix (200x200)	10	3.11 ms	1.67 ms	0.29 ms	9.15 ms	14.43 ms
	50	3.39 ms	9.11 ms	0.29 ms	21.94 ms	34.95 ms
	200	3.79 ms	30.05 ms	0.29 ms	63.35 ms	97.52 ms

Figure (8) – Tabulation of average trial times based on matrix type and number of eigenfaces used.

Figure 9 is a table showing the detection results for every number of eigenfaces used in the trials. The data is not separated by matrix like in figure 8 as the eigenfaces calculated from both are identical. From the table we can see that the program correctly detects almost all the faces even when using a small number of eigenfaces. In addition, the accuracy climbs significantly by 2.75% to 92.25% when increasing from 5 to 10 eigenfaces, quickly plateaus, climbing only 2.25% with the next 190 eigenfaces added. This clearly shows how only a small amount of eigenfaces help significantly with detection.

Eigenfaces used	5	10	50	100	200
False positives	25	16	15	10	10
False negatives	17	15	9	13	11
Correct detections	358	369	376	377	379
Accuracy	89.50%	92.25%	94.00%	94.25%	94.75%

Figure (9) – Table showing the results of detection from different amounts of eigenfaces

The overall accuracy of the detection is around 94%, matching the results found by Çarıkçı and Özen (123). This is remarkably accurate for a facial recognition method that does not implement any neural networks or pattern refinement algorithms. The system is relatively simple, efficient, easy to recreate, and accurate for images without too much variation in condition. It's also worth noting that the trials performed don't show any clear bias towards false positives or false negatives, as the prevalence of each changed from trial to trial.

Since it was shown in figure 8 that the computation time increases drastically the more faces are used, it can be concluded that the program runs most effectively with only the eigenfaces corresponding with the highest 5-10% of eigenvalues, due to the rapid drop off in eigenvalues. This decrease in magnitude of the eigenvalues is modeled in figure 10 on the right, which shows the size of the first 8 eigenvalues. As the graph shows, the eigenvalues decrease seemingly

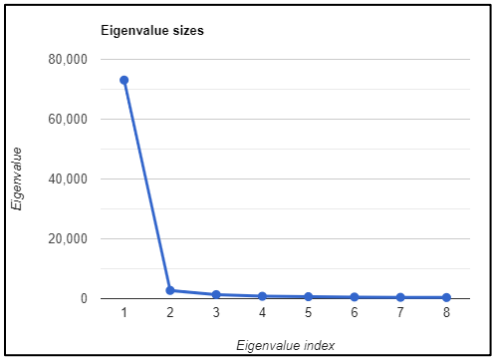

Figure (10) – Graph showing the size of the 8 highest eigenvalues

exponentially, with the biggest drop being from the highest eigenvalue at 73,068.44S to the second at 2,749.29.

5. Discussion and Conclusions

5.1 Discoveries – While not outlined in this paper, there were some interesting discoveries that were made in the process of writing the software. One is that the same exact process can be used to recognize any type of object, or in fact any patterns in any dataset if there is enough training data. This is due to the generality of PCA – the method is intended to efficiently analyze variations in any data, not just faces – and as such highlights the power of the method.

Another realization that was made is that the math and logic behind the process is quite simple and straightforward. Almost all the process just uses scalar and matrix multiplication, and the most advanced operations are projection, which is made simple due to the properties of orthonormal bases, and eigen calculations, which don't need to be manually calculated.

5.2 Relevance – In terms of linear algebra, many of the steps in the eigenface process provide fantastic distillations of complicated concepts. An example of this is the visualization provided by figure 8 of projecting a vector onto a subspace and how linear combinations work. The recreation of the face gives a great intuitive showcase of how a vector can be decomposed into several components that lie in a different subspace but that combine in a linear combination to form the original vector. Another good example of this is the practical application of Euclidean distances, by representing an image as a point in a subspace and showing that the distance between two points in that subspace represent how similar two images are.

5.3 Future work – While this project covers the topic in-depth, there is more that can be done to understand it fully. One area that could be looked at more closely is the issue of increasing accuracy in recognizing the same face in different conditions. The implementation in this paper is very simple and only uses images of faces taken in ideal conditions, so it would be worth looking at how to make the process more generally applicable in real-world situations whether that is through image processing or more advanced methods.

Another part of the project that would be interesting to examine more is the role of variability and methods of measuring it regarding data in general. This seems like an important part of the technique, and it is likely that it can be applied in any number of situations.

5.4 Conclusion – From analyzing this process and its results when applied, PCA is a very useful tool in analyzing large and complex datasets. Additionally, it provides a simple and effective way of categorizing data and highlights interesting features that otherwise would have been easily overlooked. When applied in eigenfaces, it facilitates very reliable face recognition with relatively little computing power. More than that, it gives ways to visualize some key concepts of linear algebra and statistics that are very intuitive.

6. References

Çarıkçı, Müge, and Figen Özen. “A Face Recognition System Based on Eigenfaces Method.” *Procedia Technology*, vol. 1, 2012, pp. 118–123., https://doi.org/10.1016/j.protcy.2012.02.023.

Henderson, Thomas. “A Geometric Interpretation of the Covariance Matrix.” *Digital Image Processing*, 23 July 2020, http://www.cs.utah.edu/~tch/CS6640F2020/resources/A%20geometric%20interpretation%20of%20the%20covariance%20matrix.pdf.

Kun, Jeremy. “Eigenfaces, for Facial Recognition.” *Math ∩ Programming*, 27 July 2011, https://jeremykun.com/2011/07/27/eigenfaces/.

Mohamed, Sheik. *Olivetti Faces*. Kaggle.com, 2017. Web. 29 Nov 2022. https://www.kaggle.com/datasets/imrandude/olivetti

Nayeem, Sadique and S. Ravi. “A Study on Face Recognition Technique Based on Eigenface.” *International Journal of Applied Information Systems*, vol. 5, no. 4, Mar. 2013, pp. 57–62., www.ijais.org. Accessed 5 Dec. 2022.

“NumPy Documentation.” *NumPy*, NumPy, 2008, https://numpy.org/doc/.

Slavkovic, Marijeta, and Dubravka Jevtic. “Face Recognition Using Eigenface Approach.” *Serbian Journal of Electrical Engineering*, vol. 9, no. 1, Feb. 2012, pp. 121–130., https://doi.org/10.2298/sjee1201121s.

Turk, Matthew, and Alex Pentland. “Eigenfaces for Recognition.” *Journal of Cognitive Neuroscience*, vol. 3, no. 1, 1 Jan. 1991, pp. 71–86., https://doi.org/10.1162/jocn.1991.3.1.71.

Appendix

Python code implementing Eigenfaces – eigenfaces.py

```
import numpy as np
import matplotlib.pyplot as plt
import cv2
import time

# Change from a 1xN^2 vector to a NxN matrix image
def vector_to_image(vector, dimension):
  return np.reshape(vector, (dimension,dimension))

# Access the face images and split them by person
def load_face_images(facefile, targetfile):
  # Load faces and face IDs
  faces = np.load(facefile)
  target = np.load(targetfile)
  curr_id = target[0]
  training_ims = []
  person = []
  for face, face_id in zip(faces, target):
    # New individual, start new list
    if face_id != curr_id:
      curr_id = face_id
      training_ims.append(person)
      person = [face.flatten()]
    # Same individual, add to their list
    else:
      person.append(face.flatten())
  # Add last set of faces
  training_ims.append(person)
  # Transform into matrix
  training_ims = np.asarray(training_ims)
  return training_ims, target, faces

# Create a face that represents the average of every training face
def find_average_face(training_ims, target):
  # Initialize empty matrix
  av_face = np.zeros_like(training_ims[0][0])
  # Access every face vector
  for person in training_ims:
    for face in person:
      # Add the face vector to a running sum vector
      av_face = np.add(av_face, face)
```

```
  # Divide the sum vector by the total number of faces
  av_face = av_face / len(target)
  return av_face

# Calculate each training face's variance from the average
def find_difference_faces(training_ims):
  difference_matrix = []
  # Access every face vector
  for person in training_ims:
    for face in person:
      # Calculate how much the face differs from the mean
      diff_im = np.subtract(face, av_face)
      # Add the difference vector to a matrix
      difference_matrix.append(diff_im)

  # Turn into numpy array, take transpose so row vectors become columns
  return np.asarray(difference_matrix).T

# Find vectors representing most prominent features of the training faces
def find_eigenfaces(difference_matrix, num_significant):

  # Calculate A(T)A instead of Covariance matrix AA(T)
  eigenmatrix = np.matmul(difference_matrix.T, difference_matrix)
  # Find the eigenvectors vi of A(T)A, sorted from heighest to lowest eige
nvalue
  eigenvalues, eigenvectors = np.linalg.eig(eigenmatrix)

  # Calculate the eigenvectors of Covariance matrix AA(T), aka "eigenfaces
"
  # Only keep the M' eigenfaces with the highest eigenvalue for efficiency
  eigenfaces = np.matmul(difference_matrix,eigenvectors)[:,:num_significan
t]
  # Normalize each eigenface so ||Avi|| = 1
  for i in range(num_significant):
    # Find vector norm
    norm = np.linalg.norm(eigenfaces[:,i])
    # Divide values in vector by its norm
    eigenfaces[:,i] = (1/norm) * eigenfaces[:,i]
  return eigenfaces

# Find the contribution of each eigenface towards an image
def find_weights(face):
  # Create an empty vector, the pattern vector
  face_weights = []
  # Calculate difference of face from the mean
```

```
    diff = np.subtract(face, av_face)
    # Calculate the weight of each eigenface towards image, add to list
    for eface in eigenfaces.T:
      # Equivalent to doing the calculation eface(T)diff
      face_weights.append(np.matmul(eface, diff))
    # Turn into numpy array
    face_weights = np.asarray(face_weights)
    return face_weights

# Calculate the weights for every individual in the dataset
def classify_faces(authorized_faces):
    face_classes = []
    # Access every face vector
    for person in authorized_faces:
      weights = np.zeros_like(eigenfaces[0,:])
      for face in person:
        # Combine pattern vectors for several images of same person
        face_weights = find_weights(face)
        weights = np.add(weights, face_weights)
      # Take the average of the person's pattern vectors
      num_faces = len(person)
      weights = np.divide(weights, num_faces)
      # Add as a face classifier
      face_classes.append(weights)
    return face_classes

# Find the most similar face to an image
def distance_to_classes(image_pattern):
    distances = []
    # Look at every face class
    for classifier in face_classes:
      # Find how close image is to this class
      distance_to_class = np.linalg.norm(image_pattern - classifier)
      # Append nonzero distances
      if distance_to_class > 0:
        distances.append(distance_to_class)
    # Return index of and distance to closest face class
    closest_face = np.argmin(distances)
    min_distance = np.min(distances)
    return closest_face, min_distance

# Calculate a maximum distance threshold for
# an image to be considered a match
def calculate_threshold():
    class_distances = []
```

```
  # Look at every face class
  for classifier in face_classes:
    # Find smallest distance from this class to all others
    _, dist = distance_to_classes(classifier)
    class_distances.append(dist)
  # Set threshold as 0.8 times the maximum of the minimum distances betwee
n classes
  return 0.8*np.max(class_distances)

# Decide if image is of authorized face, and return index of matching face
def is_authorized_face(image_vector):
  # Calculate eigenface weights
  image_pattern = find_weights(image_vector)
  # Find the face most similar to image and its distance
  closest_face, min_distance = distance_to_classes(image_pattern)
  # Return index of matching face if it's similar enough, otherwise return
 -1
  if min_distance < threshold:
    return closest_face, min_distance
  return -1, min_distance

# Reconstruct an image from eigenfaces
def recreate_image(pattern_vector):
  # Start with the average face
  image = av_face.copy()
  process = []
  # Access every weight and it's corresponding eigenface
  for weight, eface in zip(pattern_vector, eigenfaces.T):
    # Add the scaled eigenface to the average image
    process.append(image)
    image = np.add(image, weight * eface)
  # Return the recreated image and all the steps towards it
  return image, process

# Load face images
facefile = "./olivetti_faces.npy"
targetfile = "./olivetti_faces_target.npy"
authorized_faces, face_ids, face_images = load_face_images(facefile, targe
tfile)

# Split faces into authorized and unauthorized categories
authorized_faces = authorized_faces[:20]

# Perform PCA
av_face = find_average_face(authorized_faces, face_ids)
```

```
difference_faces = find_difference_faces(authorized_faces)
eigenfaces = find_eigenfaces(difference_faces, num_significant=200)
# Classify the features of every authorized face
face_classes = classify_faces(authorized_faces)
# Decide the minimum allowable distance to a class for a face to be recogn
ized
threshold = calculate_threshold()

decisions = []
distances = []
for face in face_images:
  face_vector = face.flatten()
  # Look at every face and determine if it's one from the training set
  decision = is_authorized_face(face_vector)
  # Add the decision made to a list
  decisions.append(decision[0])
  distances.append(decision[1])

accepted = []
rejected = []
# Sort distances into ones that were rejected and accepted
for i in range(len(distances)):
  if decisions[i] == -1:
    accepted.append(-1)
    rejected.append(distances[i])
  else:
    accepted.append(distances[i])
    rejected.append(-1)

# Split decisions into the expected categories
auth = accepted[:199]
unauth = rejected[200:]

# Count how many authorized faces were seen as unauthorized
f_negative = auth.count(-1)

# Count how many unauthorized faces were seen as authorized
f_positive = unauth.count(-1)

# Calculate the recognition accuracy and print the values
accuracy = 100 - ((f_positive + f_negative) * 100 / 400)
print("False negatives: ", f_negative)
print("False positives: ", f_positive)
print("Accuracy: ", accuracy, "%")
```