Lab 11

Facial Recognition Using Eigenfaces

Lab Objective: *Use the singular value decomposition to implement a simple facial recognition system.*

Suppose we have a large database containing images of human faces. We would like to identify people by comparing their pictures to those in the database. This task is called *facial recognition*.

One way to automate the comparison process is know as the *eigenfaces* method. As the name suggests, this method uses eigenvectors of matrices related to the collection of face images. The method essentially projects face images to a low-dimensional subspace, in a way that preserves their distinguishing characteristics. Comparing the images in fewer dimensions is faster and allows us to store the images using less data.

The idea of projecting to fewer dimensions is not unique to the eigenfaces method. This method is an example of *principal component analysis*, where data is compared based on its principal components in a lower-dimensional vector space. Principal component analysis can be applied to many computing problems besides facial recognition.

Load the Data

The first step is to obtain a dataset of face images. Recall that a digital image may be stored as an $m \times n$ array of pixels. In this lab, we will store the images as mn-vectors by concatenating the rows of the $m \times n$ arrays.

Problem 1. In this lab we will use the `faces94` face image dataset found at `http://cswww.essex.ac.uk/mv/allfaces/faces94.html` This problem will make sure that you can load and display the images from the dataset.

1. Download the `faces94` dataset from the link above and extract the files. You should now have a directory named "faces94" which contains photographs of 153 people, organized into folders by person.

Figure 11.1: The mean face.

2. The function `getFaces()` is given in the appendix. It constructs a set of face images by selecting exactly one face image for each person in the directory. It should return an array whose columns are flattened face images. Feel free to modify the given code. You may have to replace the parameter `"./faces94"` with the location of the directory `faces94` on your machine.

   Test this function to make sure it runs without errors. Check that the return value `F` is a 36000×153 array. The columns of this array are 153 flattened face images of 153 different people.

3. Use `plt.imshow()` to display one of the faces. The original image dimensions are 200×180. You may find it useful to write a helper function that accepts a flattened image and displays it.

Shift By the Mean

The facial recognition algorithm is more robust if we first *shift by the mean*. When we shift a set of data by the mean, the distinguishing features are exaggerated. Therefore, in the context of facial recognition, shifting by the mean accentuates the unique features of the face. Suppose we have a collection of k face images represented as vectors $\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_k$ of length mn. Define the *mean face* $\boldsymbol{\mu}$ to be the average of the $\mathbf{f}_i$:

$$\boldsymbol{\mu} = \frac{1}{k}\sum_{i=1}^{k} \mathbf{f}_i.$$

Problem 2.

The facial recognition method you will write in this lab will be structured as a class. An outline of the `FacialRec` class is provided in the appendix. You will write the methods of the class.

When initialized, the `FacialRec` object first loads the face images using

Figure 11.2: Three mean-shifted faces from the dataset.

`getFaces` and stores the result. The next step is to compute the mean face.

1. In your class definition, implement the method `FacialRec.initMeanImage()`. Compute the mean face and store it as `self.mu`. This can be done in one line of code using NumPy.
2. Display the mean face. Your result should match Figure 11.1.

For each $i = 1, \ldots, k$, define $\bar{\mathbf{f}}_i := \mathbf{f}_i - \boldsymbol{\mu}$. The mean-shifted face vector $\bar{\mathbf{f}}_i$ is the deviation of the i-th face from the mean, and thus captures the unique features of the face. Now form the $mn \times k$ matrix $\bar{F}$ whose columns are given by the mean-shifted face vectors, i.e.

$$\bar{F} = \begin{bmatrix} \bar{\mathbf{f}}_1 & \bar{\mathbf{f}}_2 & \cdots & \bar{\mathbf{f}}_k \end{bmatrix}.$$

Problem 3.

1. In your class definition, implement the method `FacialRec.initDifferences()`. Compute $\bar{F}$ and store it as `self.Fbar`. This can be done in one line using array broadcasting.
2. Display one of the mean-shifted faces. The output should be similar to the faces in Figure 11.2.

Project to a Subspace

Now suppose we are given a new face vector $\mathbf{g}$. We first shift $\mathbf{g}$ by the mean of the dataset, giving us $\bar{\mathbf{g}} = \mathbf{g} - \boldsymbol{\mu}$. The closest match to $\bar{\mathbf{g}}$ is the vector $\bar{\mathbf{f}}_i$ that minimizes $\|\bar{\mathbf{g}} - \bar{\mathbf{f}}_i\|_2$. If there are k images in the dataset, we find this match by comparing $\bar{\mathbf{g}}$ to every element of $\{\bar{\mathbf{f}}_1, \ldots \bar{\mathbf{f}}_k\}$.

However, comparing the original face images pixel by pixel is computationally expensive and inefficient. The vectors $\bar{\mathbf{f}}_i$ and $\bar{\mathbf{g}}$ are length mn, which in our case

equals 36000 and in practice may be many times larger. Computing the difference between face vectors of this length is time consuming, especially when the dataset is very large. It also requires us to use mn values to store each face, which is an inefficient use of space. In addition, pixel by pixel comparison is not very robust to small changes in individual pixels.

Instead, in order to store and compare our face vectors, we would like to represent each one with fewer than mn values. We can do this by projecting to a subspace. Mathematically, we want to choose s, $s \ll mn$, and project the face vectors to an s-dimensional subspace of the original mn-dimensional space of images. We can then use just s values to store each face in terms of the basis vectors of the new subspace.

The "best" subspace to project to is the one that is closest in the least squares sense (i.e., such that the sum of the squared errors between $\{\bar{\mathbf{f}}_1, \dots \bar{\mathbf{f}}_k\}$ and their projections is minimized). Let $U\Sigma V^T$ be an SVD of $\bar{F}$, with $\mathbf{u}_i$ the columns of U. As we will prove below, the s-dimensional subspace that minimizes the squared error is the span of $\mathbf{u}_1, \dots, \mathbf{u}_s$. Note that $\mathbf{u}_1, \dots, \mathbf{u}_s$ is an orthonormal basis for this subspace.

The projection matrix is $P_s = U_s U_s^T$ where $U_s = [\mathbf{u}_1 \ \dots \ \mathbf{u}_s]$. This matrix projects the original face vectors into the optimal s-dimensional subspace.

The Proof: SVD as a Least Squares Solution

Theorem 11.1. *Let $\mathbf{f}_1, \dots, \mathbf{f}_k$ be vectors on $\mathbb{R}^{mn}$, and let $\bar{F} = [\bar{f}_1 \ \dots \ \bar{f}_k]$. Suppose $U\Sigma V^T$ is an SVD for $\bar{F}$. Then the s-dimensional subspace that solves the least squares problem for $\mathbf{f}_1, \dots, \mathbf{f}_k$ is the span of the first s columns of U. If U_s is the first s columns of U, then the matrix $U_s U_s^T$ is the projection onto this subspace.*

Proof. We seek a rank-s projection matrix P_s so that $\sum_{i=1}^k \|P_s\bar{\mathbf{f}}_i - \bar{\mathbf{f}}_i\|_2^2$ is minimized—i.e., the sum of the squares of the "errors" is minimal when we project $\bar{\mathbf{f}}_i$ via P_s. But minimizing this quantity is the same as minimizing its square, which happens to equal the Frobenius norm of $P_s\bar{F} - \bar{F}$. Written mathematically,

$$\begin{aligned}\inf_{\text{rank}(P_s)=s} \sum_{i=1}^k \|P_s\bar{\mathbf{f}}_i - \bar{\mathbf{f}}_i\|_2^2 &= \inf_{\text{rank}(P_s)=s} \left(\sum_{i=1}^k \|P_s\bar{\mathbf{f}}_i - \bar{\mathbf{f}}_i\|_2^2\right)^2 \\ &= \inf_{\text{rank}(P_s)=s} \|P_s\bar{F} - \bar{F}\|_F.\end{aligned}$$

Now let $U\Sigma V^T$ be an SVD of $\bar{F}$ with $\mathbf{u}_i$ the columns of U, $\mathbf{v}_i$ the columns of V, and σ_i the singular values of $\bar{F}$. If $P_s = \sum_{i=1}^s \mathbf{u}_i\mathbf{u}_i^T$, then

$$\begin{aligned}P_s\bar{F} &= \left(\sum_{i=1}^s \mathbf{u}_i\mathbf{u}_i^T\right)\left(\sum_{j=1}^k \sigma_j\mathbf{u}_i\mathbf{v}_i^T\right) = \sum_{i=1}^s\sum_{j=1}^k \sigma_j\mathbf{u}_i\mathbf{u}_i^T\mathbf{u}_j\mathbf{v}_j^T \\ &= \sum_{i=1}^s\sum_{j=1}^k \sigma_j\mathbf{u}_i\delta_{ij}\mathbf{v}_j^T = \sum_{i=1}^s \sigma_i\mathbf{u}_i\mathbf{v}_i^T.\end{aligned}$$

Figure 11.3: The top three eigenfaces.

In fact, the Schmidt-Eckart-Young-Mirsky Theorem from Lab 10 tells us that $X = \sum_{i=1}^{s} \sigma_i \mathbf{u}_i \mathbf{v}_i^T$ is exactly the rank-s matrix that minimizes $\|X - \bar{F}\|_F$. Since $P_s \bar{F}$ will always have rank s or less, the projection $P_s = \sum_{i=1}^{s} \mathbf{u}_i \mathbf{u}_i^T$ is the one we seek. If we let $U_s = [\mathbf{u}_1 \ \dots \ \mathbf{u}_s]$, then we may write $P_s = U_s U_s^T$. Notice that P_s is projection onto the subspace spanned by the columns of U_s. □

The s basis vectors $\mathbf{u}_1, \dots, \mathbf{u}_s$ are eigenvectors of $\bar{F}\bar{F}^T$. They also resemble face images. Hence, they are commonly called the "eigenfaces."

Problem 4.

1. In your class definition, implement the method `FacialRec.initEigenfaces()`.

   Compute the SVD (`scipy.linalg.svd()` is a good implementation to use) and store the array `U` containing the eigenfaces as its columns. Because we will only use the first few columns of `U`, specify the keyword parameter `full_matrices=False` to compute only the compact SVD.

2. Plot the first eigenface (i.e. the first column of `U`). It should resemble the first eigenface shown in Figure 11.3.

Change Basis

The projection matrix $P_s = U_s U_s^T$ projects a face vector into the s-dimensional subspace spanned by the eigenfaces, but still keeps it as a vector in $\mathbb{R}^{mn}$. The change-of-basis matrix U_s^T both projects the face vector and and changes the basis. The resulting vector has length s and represents the face in terms of eigenfaces.

To represent any face vector in terms of the first s eigenfaces, multiply by U_s^T. To change back to a full length-mn projection, multiply again by U_s.

Problem 5.

1. Implement the method `FacialRec.project()` in your class definition. This should accept a flattened image or an array with flattened images as its columns. It should also accept a value for s. The function should project the image or images into the appropriate s-dimensional subspace and change basis, then return the result.
2. Let `face` be the first mean-shifted face (the first column of `facialRec.Fbar`). Do the following:
   (a) Project `face` so that it is represented in terms of the first 19 eigenfaces.
   (b) Change basis again back to the standard basis on $\mathbb{R}^{mn}$.
   (c) Add back the mean face `facialRec.mu`.
   (d) Plot the resulting image.

   Your image should match Figure 11.4e.

Match Faces

Finally, we are ready to identify which mean-shifted image $\bar{\mathbf{f}}_i$ is closest to an input image, $\bar{\mathbf{g}}$. We begin by projecting all vectors to some s-dimensional subspace and writing them in terms of the basis vectors, which are the eigenfaces. This is done by multiplying by the change-of-basis matrix:

$$\widehat{\mathbf{f}}_i = U_s^T(\mathbf{f}_i - \boldsymbol{\mu}) \qquad \widehat{\mathbf{g}} = U_s^T(\mathbf{g} - \boldsymbol{\mu}).$$

Next, we compute which $\widehat{\mathbf{f}}_i$ is closest to $\widehat{\mathbf{g}}$. Since the columns of U_s are an orthonormal basis, we get the same result doing the computation in this basis as we would in the standard Euclidean basis. Define

$$i^* = \text{argmin}_i \|\widehat{\mathbf{f}}_i - \widehat{\mathbf{g}}\|_2.$$

Then the i^*-th face image is the best match for $\mathbf{g}$.

Problem 6.

1. Implement the method `FacialRec.findNearest()` as follows.

```
def findNearest(self, image, s=38):
    Fhat = # Project Fbar, producing a matrix whose columns are the f-hat defined above
    ghat = # Shift 'image' by the mean and project, producing g-hat as defined above
    # for both Fhat and ghat, use your project function from the previous problem
```

(a) 5 eigenfaces, about 1/32 of the total.

(b) 9 eigenfaces, or 1/16 of the total.

(c) 19 eigenfaces, about 1/8 of the total.

(d) 38 eigenfaces, about 1/4 of the total.

(e) 75 eigenfaces, about 1/2 of the total.

(f) All 153 of the eigenfaces.

Figure 11.4: Image rebuilt with various numbers of eigenfaces. The image is somewhat recognizable when it is reconstructed with only 1/8 of the eigenfaces.

```
    # Return the index that minimizes ||fhat_i - ghat||_2.
```

The functions `np.linalg.norm()` and `np.argmin()` will be useful for the last line. When using `np.linalg.norm`, make sure you indicate the correct axis.

2. Test your facial recognition system on faces selected randomly from the `faces94` dataset. The function `sampleFaces(n_tests, path)` at the end of this lab will build an array of `n_tests` random faces from the `faces94` dataset.

   Plot the random face beside the face returned by your facial recognition code to see if your system is accurately recognizing faces.

By this point, you have created a basic facial recognition system. We can extend the system to detect when a face doesn't match anything currently in the dataset, and then add this new face. We can also make the system more robust by including multiple pictures of the same face with different expressions and lighting conditions.

Although there are other approaches to facial recognition that utilize more complex techniques, the method of eigenfaces remains a wonderfully simple and effective solution, illustrating another application of the singular value decomposition.

Appendix: Helper Code

This section contains some functions to help you implement the facial recognition class outlined in the problems of this lab.

```
import numpy as np
from scipy import linalg as la
from os import walk
from scipy.ndimage import imread
from matplotlib import pyplot as plt
from random import sample

def getFaces(path="./faces94"):
    """Traverse the directory specified by 'path' and return an array containing
    one column vector per subdirectory.

    For the faces94 dataset, this gives an array with just one column for each
    face in the dataset. Each column corresponds to a flattened grayscale image.
    """

    # Traverse the directory and get one image per subdirectory.
    faces = []
    for (dirpath, dirnames, filenames) in walk(path):
        for f in filenames:
            if f[-3:]=="jpg": # only get jpg images
                # load image, convert to grayscale, flatten into vector
                face = imread(dirpath+"/"+f).mean(axis=2).ravel()
                faces.append(face)
                break

    # put all the face vectors column-wise into a matrix.
    F = np.array(faces).T
    return F


def sampleFaces(n_tests,path = "./faces94")
    """Return an array containing a sample of n_tests images contained
    in the path as flattened images in the columns of the output.
    """
    files = []
    for (dirpath, dirnames, filenames) in walk(path):
        for f in filenames:
            if f[-3:]=="jpg": # only get jpg images
            files.append(dirpath+"/"+f)

    #Get a sample of the images
    test_files = sample(files, n_tests)
    #Flatten and average the pixel values
    images = np.array([imread(f).mean(axis=2).ravel() for f in test_files]).T
    return images
```

The following is the outline of the Facial Recognition class.

```
class FacialRec:
    ##########Members##########
    #   F, mu, Fbar, and U
    ############################
    def __init__(self,path):
        self.initFaces(path)
        self.initMeanImage()
        self.initDifferences()
        self.initEigenfaces()

    def initFaces(self, path):
        self.F = getFaces(path)
    def initMeanImage(self):
        pass
    def initDifferences(self):
        pass
    def initEigenfaces(self):
        pass
    def project(self, A, s=38):
        pass
    def findNearest(self, image, s=38):
        pass
```