Lab 1

Facial Recognition Using Eigenfaces

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

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

Facial recognition is important in law enforcement, as well as other situations. For example, facial recognition can be combined with video surveillance to identify when a person is not authorized to be in a location.

Humans can easily compare two face images and determine whether they belong to the same person, but automating this process is more challenging. One technique for automated facial recognition uses *eigenfaces*.

Load the Data

Eigenfaces are an efficient way to store and query a database of face images. As the name suggests, this method uses eigenvectors of matrices related to the collection of face images. Essentially, the method of eigenfaces projects face images to a lower-dimensional subspace in a way that preserves their distinguishing characteristics. In the lower-dimensional subspace, comparing face images is much faster.

Recall that a digital image may be stored as an $m \times n$ array of pixel intensities. In this lab, we will store the images as mn-vectors by concatenating the rows of the $m \times n$ arrays.

Problem 1. The problems in this lab will help you write a class `FacialRec` to perform facial recognition. First we need to get a database of face images.

1. Download the `faces94` face image database found at `http://cswww.essex.ac.uk/mv/allfaces/faces94.html` and extract the files. You should now have a directory named "faces94" which contains photographs of many people, organized into folders by person.

With this directory we can begin to write our `FacialRec` class. This class is outlined below.

```python
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
import matplotlib.cm as cm
from random import sample

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
```

The function `getFaces()` should construct a database of face images by selecting exactly one face image for each person in the directory. It should return an array whose columns are the selected face images. One implementation of this function is found at the end of this lab.

resume Initialize a `FacialRec` instance with the command `facialRec = FacialRec("./faces94")`. You may have to replace the parameter `"./faces94"` with the location of the directory `faces94` on your machine. Check that `facialRec.F` is a 360000×153 array. The columns of this array are 153 face images of 153 different people.

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

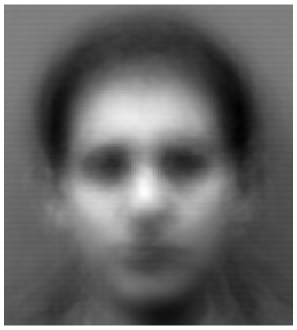


Figure 1.1: The mean face.

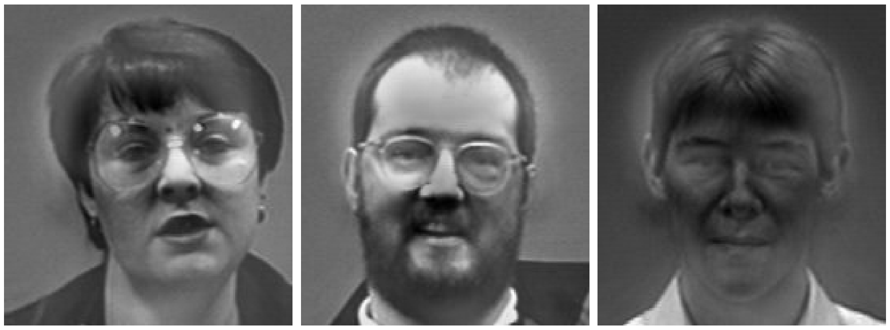


Figure 1.2: Three mean-shifted faces from the dataset.

average of the $\mathbf{f}_i$:

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

Problem 2.

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

```
def initMeanImage(self):
    self.mu = # Compute the mean face of the images in self.F
```

   This can be done in one line using `np.mean` and specifying the correct axis.

2. Plot the mean face. The function `show()` at the end of this lab will plot a flattened grayscale image. Your mean face should match Figure 1.1.

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. Implement the method `FacialRec.initDifferences()` to compute $\bar{F}$ as follows.

```
def initDifferences(self):
    self.Fbar = # Compute the mean-shifted face vectors Fbar
```

   This can be done in one line using array broadcasting.

2. Plot a mean-shifted face. Plotting the 28th mean-shifted face, i.e. the 28th column of the matrix `Fbar`, should match the first face in Figure 1.2.

Project to a Subspace

Now suppose we have a new face vector $\mathbf{g}$. The closest face image to $\mathbf{g}$ should be the vector $\bar{\mathbf{f}}_i$ that minimizes $\|\bar{\mathbf{g}} - \bar{\mathbf{f}}_i\|_2$, where $\bar{\mathbf{g}} = \mathbf{g} - \boldsymbol{\mu}$. Unfortunately, computing $\|\bar{\mathbf{g}} - \bar{\mathbf{f}}_i\|_2$ for each i is computationally intractable when the length mn of the vectors is large. Since a low-resolution photo may easily have $100 \times 100 = 10,000$ pixels, in practice mn is very large indeed.

Here is the trick: instead of computing in the mn-dimensional space of all possible images, we will compute in a lower-dimensional subspace. We could start by using the subspace spanned by the vectors $\bar{\mathbf{f}}_1, \ldots \bar{\mathbf{f}}_k$, which is at most k-dimensional. Unfortunately, in practice k is still to large for this subspace to be computationally efficient.

Therefore, we want to project to a subspace of $\bar{\mathbf{f}}_1, \ldots \bar{\mathbf{f}}_k$ in a way that retains as much information about the $\bar{\mathbf{f}}_i$ as possible. Mathematically, we want to find the s-dimensional subspace of span$\{\bar{\mathbf{f}}_1, \ldots \bar{\mathbf{f}}_k\}$ that is closest to the $\bar{\mathbf{f}}_i$ in the least-squares sense. In a minute, we will prove that the SVD of $\bar{F}$ (whose columns are $\bar{\mathbf{f}}_1, \ldots \bar{\mathbf{f}}_k$) solves this problem. But first, let us summarize how the solution works.

Let $U\Sigma V^T$ be an SVD of $\bar{F}$ with $\mathbf{u}_i$ the columns of U. Then the "best" s-dimensional subspace for approximating span$\{\bar{\mathbf{f}}_1, \ldots \bar{\mathbf{f}}_k\}$ is the span of $\mathbf{u}_1, \ldots, \mathbf{u}_s$. The matrix for this projection is $P_s = U_s U_s^T$ where $U_s = [\mathbf{u}_1 \ \ldots \ \mathbf{u}_s]$.

Because the vectors $\mathbf{u}_i$ are eigenvectors of $\bar{F}\bar{F}^T$, we call them *eigenfaces*. Therefore, the best s-dimensional subspace for solving the facial recognition problem is exactly the span of the first s eigenfaces.

The Proof: SVD as a Least Squares Solution

Theorem 1.1. *Let $\mathbf{f}_1, \ldots, \mathbf{f}_k$ be vectors on $\mathbb{R}^{mn}$, and let $\bar{F} = [\bar{f}_1 \ \ldots \ \bar{f}_k]$. Suppose*

$U\Sigma V^T$ is an SVD for $\bar{F}$. Then the s-dimensional subspace that solves the least squares problem for $\mathbf{f}_1, \ldots, \mathbf{f}_k$ 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 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

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

In fact, the Schmidt-Eckart-Young-Mirsky Theorem from Lab **??** 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 $U_s = [\mathbf{u}_1 \; \ldots \; \mathbf{u}_s]$, then we may write $P_s = U_sU_s^T$. Notice that P_s is projection onto the subspace spanned by the columns of U_s. $\square$

Problem 4.

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

```
def initEigenfaces(self):
    self.U, s, Vt = # Compute the SVD of Fbar
```

   This can be done in one line with the function `linalg.svd()`. 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 match the first eigenface shown in Figure 1.3.

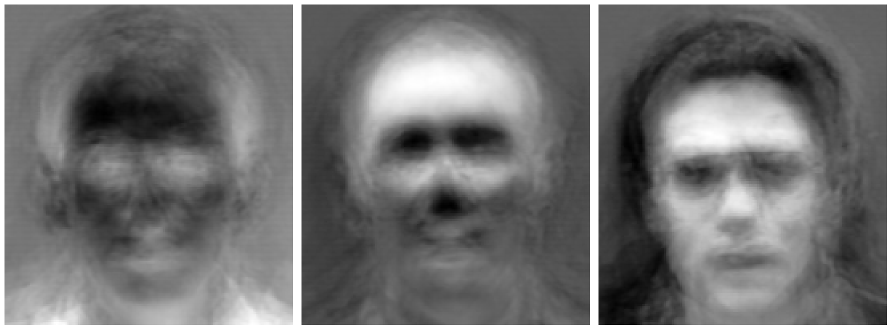


Figure 1.3: The top three eigenfaces.

Change Basis

It does us no good to project all our vectors into an s-dimensional space if we still store them as vectors in $\mathbb{R}^{nm}$. Instead, we must store our face vectors in terms of the columns of U_s. This way, each vector is a length-s array in NumPy, instead of a length-mn array.

The change-of-basis matrix is U_s^T, so $U_s^T \widehat{P}_s = U_s^T U_s U_s^T = U_s^T$. Thus we can project into the subspace and change basis by multiplying by U_s^T. To change back to the full mn-vector, multiply by U_s.

Problem 5.

1. Implement the method `FacialRec.project(s)` as follows.

```
def project(self,s):
    # project A onto s-dimensional subspace and return A_s.
```

2. (Optional) Let `face` be the first mean-shifted face from the database (the first column of `facialRec.Fbar`). Do the following:

   (a) Project `face` onto the subspace spanned by the first 75 eigenfaces.

   (b) Change basis 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 1.4e.

Recognizing Faces

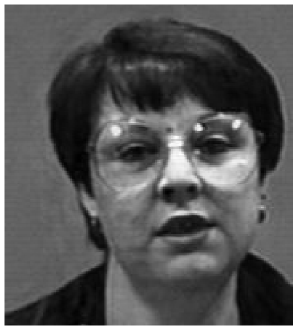
Finally, we are ready to identify which mean-shifted image $\bar{\mathbf{f}}_i$ is closest to an inputed image, $\bar{\mathbf{g}}$. We begin by projecting all vectors to some s-dimensional subspace and writing them in terms of an orthonormal basis for that subspace. This is accom-

(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.

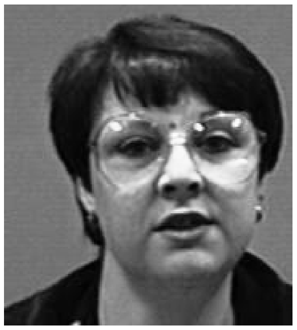

(f) All 153 of the eigenfaces.

Figure 1.4: Image rebuilt with various numbers of eigenfaces. The image is somewhat recognizable when it is reconstructed with only 1/8 of the eigenfaces.

plished with multiplication by U_s^T:

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

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

    # 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` database.

   Plot the random face beside the face returned by your facial recognition code to see if your system is accurately recognizing faces. The function `show2()` at the end of this lab will plot two face vectors side-by-side.

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 database, and then add this new face into the database. 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 code up the facial recognition class outlined in the problems of this lab.

```
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 database. 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 show(im, w=200, h=180):
    """Plot the flattened grayscale image 'im' of width 'w' and height 'h'."""

    plt.imshow(im.reshape((w,h)), cmap=cm.Greys_r)
    plt.show()



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


def show2(im1, im2, w=200, h=180):
    """Convenience function for plotting two flattened grayscale images of
    the specified width and height side by side.
    """
    plt.subplot(121)
    plt.imshow(im1.reshape((w,h)), cmap=cm.Greys_r)
    plt.subplot(122)
    plt.imshow(im2.reshape((w,h)), cmap=cm.Greys_r)
    plt.show()
```