7 Facial Recognition

Lab Objective: *Facial recognition algorithms attempt to match a person's portrait to a database of many portraits. Facial recognition is becoming increasingly important in security, law enforcement, artificial intelligence, and other areas. Though humans can easily match pictures to people, computers are beginning to surpass humans at facial recognition. In this lab, we implement a basic facial recognition system that relies on eigenvectors and the SVD to efficiently determine the difference between faces.*

Preparing an Image Database

The `faces94` face image dataset[1] contains several photographs of 153 people, organized into folders by person. To perform facial recognition on this dataset, select one image per person and convert these images into a database. For this particular facial recognition algorithm, the entire database can be stored in just a few NumPy arrays.

Digital images are stored on computers as arrays of pixels. Therefore, an $m \times n$ image can be stored in memory as an $m \times n$ matrix or, equivalently, as an mn-vector by concatenating the rows of the matrix. Then a collection of k images can be stored as a single $mn \times k$ matrix F, where each column of F represents a single image. That is, if

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

then each $\mathbf{f}_i$ is a mn-vector representing a single image.

The following function obtains one image for each person in the `faces94` dataset and converts the collection of images into an $mn \times k$ matrix F described above.

```
import os
import numpy as np
from imageio import imread

def get_faces(path="./faces94"):
    # Traverse the directory and get one image per subdirectory.
```

---

[1]See http://cswww.essex.ac.uk/mv/allfaces/faces94.html.

```
    faces = []
    for (dirpath, dirnames, filenames) in os.walk(path):
        for fname in filenames:
            if fname[-3:]=="jpg":          # Only get jpg images.
                # Load the image, convert it to grayscale,
                # and flatten it into a vector.
                faces.append(np.ravel(imread(dirpath+"/"+fname, as_gray=True)))
                break
    # Put all the face vectors column-wise into a matrix.
    return np.transpose(faces)
```

Problem 1. Write a function that accepts an image as a flattened mn-vector, along with its original dimensions m and n. Use `np.reshape()` to convert the flattened image into its original $m \times n$ shape and display the result with `plt.imshow()`.
(Hint: use `cmap="gray"` in `plt.imshow()` to display images in grayscale.)

Unzip the `faces94.zip` archive and use `get_faces()` to construct F. Each `faces94` image is 200×180, and there are 153 people in the dataset, so F should be 36000×153. Use your function to display one of the images stored in F.

The Eigenfaces Method

With the image database F, we could construct a simple facial recognition system with the following strategy. Let $\mathbf{g}$ be an mn-vector representing an unknown face that is not part of the database F. Then the $\mathbf{f}_i$ that minimizes $\|\mathbf{g}-\mathbf{f}_i\|_2$ is the matching face. Unfortunately, computing $\|\mathbf{g}-\mathbf{f}_i\|_2$ for each i is very computationally expensive, especially if the images are high-resolution and/or the database contains a large number of images. The *eigenfaces method* is a way to reduce the computational cost of finding the closest matching face by focusing on only the most important features of each face. Because the method ignores less significant facial features, it is also usually more accurate than the naïve method.

The first step of the algorithm is to shift the images by the *mean face*. Shifting a set of data by the mean exaggerates the distinguishing features of each entry. In the context of facial recognition, shifting by the mean accentuates the unique features of each face. For the images vectors stored in F, the mean face $\boldsymbol{\mu}$ is the defined to be the element-wise average of the $\mathbf{f}_i$:

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

Hence, the ith mean-shifted face vector $\bar{\mathbf{f}}_i$ is given by

$$\bar{\mathbf{f}}_i = \mathbf{f}_i - \boldsymbol{\mu}.$$

Next, define $\bar{F}$ as the $mn \times k$ matrix whose columns are given by the mean-shifted face vectors,

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

(a) The mean face. (b) An original face. (c) A mean-shifted face.

Figure 7.1

Problem 2. Write a class called `FacialRec` whose constructor accepts a path to a directory of images. In the constructor, use `get_faces()` to construct F, then compute the mean face $\boldsymbol{\mu}$ and the shifted faces $\bar{F}$. Store each array as an attribute.

(Hint: Both $\boldsymbol{\mu}$ and $\bar{F}$ can be computed in a single line of code by using NumPy functions and/or array broadcasting.)

Use your function from Problem 1 to visualize the mean face, and compare it to Figure 7.1a. Also display an original face and its corresponding mean-shifted face. Compare your results with Figures 7.1b and 7.1c.

To increase computational efficiency and minimize storage, the face vectors can be represented with fewer values by projecting $\bar{F}$ onto a lower-dimensional subspace. Let s be a natural number such that $s < r$, where r is the rank of $\bar{F}$. By projecting $\bar{F}$ onto an s-dimensional subspace, each face can be stored with only s values.

Specifically, let $U\Sigma V^{\mathsf{H}}$ be the compact SVD of $\bar{F}$ with rank r, which can also be represented by

$$\bar{F} = \sum_{i=1}^{r} \sigma_i \mathbf{u}_i \mathbf{v}_i^{\mathsf{H}}.$$

The first r columns of U form a basis for the range of $\bar{F}$. Recall that the Schmidt, Mirsky, Eckart-Young Theorem states that the matrix

$$\bar{F}_s = \sum_{i=1}^{s} \sigma_i \mathbf{u}_i \mathbf{v}_i^{\mathsf{H}}$$

is the best rank-s approximation of $\bar{F}$ for each $s < r$. This means that $\|\bar{F} - \bar{F}_s\|$ is minimized against all other $\|\bar{F} - B\|$ where B has rank s. As a consequence of this theorem, the first s columns of U form a basis that provides the "best" s-dimensional subspace for approximating $\bar{F}$.

The s basis vectors $\mathbf{u}_1, \ldots, \mathbf{u}_s$ are are commonly called the *eigenfaces* because they are eigenvectors of $\bar{F}\bar{F}^{\mathsf{T}}$ and because they resemble face images. Each original face image can be efficiently represented in terms of these eigenfaces. See Figure 7.2 for visualizations of the some of the eigenfaces for the `facesd94` data set.
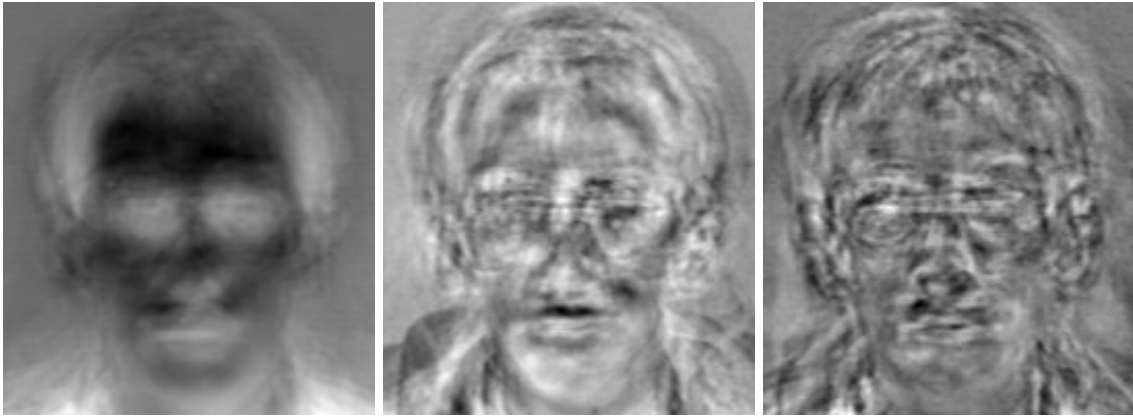
Figure 7.2: The first, 50th, and 100th eigenfaces.

In general, the lower eigenfaces provide a more general information of a face and higher-ordered eigenfaces provide the details necessary to distinguish particular faces [MMH04]. These eigenfaces will be used to construct the face images in the dataset. The more eigenfaces used, the more detailed the resulting image will be.

Next, let U_s be the matrix with the first s eigenfaces as columns. Since the eigenfaces $\{\mathbf{u}_i\}_{i=1}^s$ form an orthonormal set, U_s is an orthonormal matrix (independent of s) and hence $U_s^\mathsf{T} U_s = I$. The matrix $P_s = U_s U_s^\mathsf{T}$ projects vectors in $\mathbb{R}^{mn}$ to the subspace spanned by the orthonormal basis $\{\mathbf{u}_i\}_{i=1}^s$, and the change of basis matrix U_s^T puts the projection in terms of the basis of eigenfaces. Thus the projection $\widehat{\mathbf{f}}_i$ of $\bar{\mathbf{f}}_i$ in terms of the basis of eigenfaces is given by

$$\widehat{\mathbf{f}}_i = U_s^\mathsf{T} P_s \bar{\mathbf{f}}_i = U_s^\mathsf{T} U_s U_s^\mathsf{T} \bar{\mathbf{f}}_i = U_s^\mathsf{T} \bar{\mathbf{f}}_i. \tag{7.1}$$

Note carefully that though the shifted image $\bar{\mathbf{f}}_i$ has mn entries, the projection $\widehat{\mathbf{f}}_i$ has only s entries since U_s is $mn \times s$. Likewise, the matrix $\widehat{F}$ that has the projections $\widehat{\mathbf{f}}_i$ as columns is $s \times k$, and

$$\widehat{F} = U_s^\mathsf{T} F. \tag{7.2}$$

> **Problem 3.** In the constructor of `FacialRec`, calculate the compact SVD of $\bar{F}$ and save the matrix U as an attribute. Compare the computed eigenfaces (the columns of U) to Figure 7.2.
>
> Also write a method that accepts a vector of length mn or an $mn \times \ell$ matrix, as well as an integer $s < mn$. Construct U_s by taking the first s columns of U, then use (7.1) or (7.2) to calculate the projection of the input vector or matrix onto the span of the first s eigenfaces. (Hint: this method should be implemented with a single line of code.)

Reducing the mean-shifted face image $\bar{\mathbf{f}}_i$ to the lower-dimensional projection $\widehat{\mathbf{f}}_i$ drastically reduces the computational cost of the facial recognition algorithm, but this efficiency gain comes at a price. A projection image only approximates the corresponding original image, but as long as s isn't too small, the approximation is usually good enough for the algorithm to work well. Before completing the facial recognition system, we reconstruct some of these projections to visualize the amount of information lost.

From (7.1), since U_s^T projects $\bar{\mathbf{f}}_i$ and performs a change of basis to get $\widehat{\mathbf{f}}_i$, its transpose U_s puts $\widehat{\mathbf{f}}_i$ back into the original basis with as little error as possible. That is,

$$U_s \widehat{\mathbf{f}}_i \approx \bar{\mathbf{f}}_i = \mathbf{f}_i - \boldsymbol{\mu},$$

so that we have the approximation

$$\widetilde{\mathbf{f}}_i = U_s\widehat{\mathbf{f}}_i + \boldsymbol{\mu} \approx \mathbf{f}_i. \tag{7.3}$$

This $\widetilde{\mathbf{f}}_i$ is called the *reconstruction* of $\mathbf{f}_i$.

(a) A reconstruction with $s = 5$.   (b) A reconstruction with $s = 19$.   (c) A reconstruction with $s = 75$.

Figure 7.3: An image rebuilt with various numbers of eigenfaces. The image is already recognizable when it is reconstructed with only 19 eigenfaces, less than an eighth of the 153 eigenfaces corresponding to nonzero eigenvalues or $\overline{F}\overline{F}^\mathsf{T}$. Note the similarities between this method and regular image compression via the truncated SVD.

Problem 4. Instantiate a `FacialRec` object that draws from the `faces94` dataset. Select one of the shifted images $\overline{\mathbf{f}}_i$. For at least 4 values of s, use your method from Problem 3 to compute the corresponding s-projection $\widehat{\mathbf{f}}_i$, then use (7.3) to compute the reconstruction $\widetilde{\mathbf{f}}_i$. Display the various reconstructions and the original image. Compare your results to Figure 7.3

Matching Faces

Let $\mathbf{g}$ be a vector representing an unknown face that is not part of the database. We determine which image in the database is most like $\mathbf{g}$ by comparing $\widehat{\mathbf{g}}$ to each of the $\widehat{\mathbf{f}}_i$. First, shift $\mathbf{g}$ by the mean to obtain $\overline{\mathbf{g}}$, then project $\overline{\mathbf{g}}$ using a given number of eigenfaces:

$$\widehat{\mathbf{g}} = U_s^\mathsf{T}\overline{\mathbf{g}} = U_s^\mathsf{T}(\mathbf{g} - \boldsymbol{\mu}) \tag{7.4}$$

Next, determine which $\widehat{\mathbf{f}}_i$ is closest to $\widehat{\mathbf{g}}$. Since the columns of U_s are an orthonormal basis, the computation in this basis yields the same result as computing in the standard Euclidean basis would. Then setting

$$j = \underset{i}{\operatorname{argmin}}\|\widehat{\mathbf{f}}_i - \widehat{\mathbf{g}}\|_2, \tag{7.5}$$

we have that the jth face image $\mathbf{f}_j$ is the best match for $\mathbf{g}$. Again, since $\widehat{\mathbf{f}}_i$ and $\widehat{\mathbf{g}}_i$ only have s entries, the computation in (7.5) is much cheaper than comparing the raw $\mathbf{f}_i$ to $\mathbf{g}$.

Problem 5. Write a method for the `FacialRec` class that accepts an image vector **g** and an integer s. Use your method from Problem 3 to compute $\widehat{F}$ and $\widehat{\mathbf{g}}$ for the given s, then use (7.5) to determine the best matching face in the database. Return the index of the matching face. (Hint: `scipy.linalg.norm()` and `np.argmin()` may be useful.)

NOTE

This facial recognition system works by solving a *nearest neighbor search*, since the goal is to find the $\mathbf{f}_i$ that is "nearest" to the input image **g**. Nearest neighbor searches can be performed more efficiently with the use of a *k-d tree*, a binary search tree for storing vectors. The system could also be called a *k-neighbors classifier* with $k = 1$.

Problem 6. Write a method for the `FacialRec` class that accepts an flat image vector **g**, an integer s, and the original dimensions of **g**. Use your method from Problem 5 to find the index j of the best matching face, then display the original face **g** alongside the best match $\mathbf{f}_j$.

The following generator yields random faces from `faces94` that can be used as test cases.

```python
def sample_faces(num_faces, path="./faces94"):
    # Get the list of possible images.
    files = []
    for (dirpath, dirnames, filenames) in os.walk(path):
        for fname in filenames:
            if fname[-3:]=="jpg":        # Only get jpg images.
                files.append(dirpath+"/"+fname)

    # Get a subset of the image names and yield the images one at a time.
    test_files = np.random.choice(files, num_faces, replace=False)
    for fname in test_files:
        yield np.ravel(imread(fname, as_gray=True))
```

The `yield` keyword is like a `return` statement, but the next time the generator is called, it will resume immediately after the last `yield` statement.[a]

Use `sample_faces()` to get at least 5 random faces from `faces94`, and match each random face to the database with $s = 38$. Iterate through the random faces with the following syntax.

```python
for test_image in sample_faces(5):
    # 'test_image' is a now flattened face vector.
```

_____

[a]See the Python Essentials lab on Profiling for more on generators.

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

Additional Material

Improvements on the Facial Recognition System with Eigenfaces

The `FacialRec` class does its job well, but it could be improved in several ways. Here are a few ideas.

- The most computationally intensive part of the algorithm is computing $\widehat{F}$. Instead of recomputing $\widehat{F}$ every time the method from Problem 5 is called, store $\widehat{F}$ and s as attributes the first time the method is called. In subsequent calls, only recompute $\widehat{F}$ if the user specifies a different value for s.

- Load a `scipy.spatial.KDTree` object with $\widehat{F}$ and use its `query()` method to compute (7.5). Building a kd-tree is expensive, so be sure to only build a new tree when necessary (i.e., the user specifies a new value for s).

- Include an error tolerance ε in the method for Problem 5. If $\|\mathbf{f}_j - \mathbf{g}\| > \varepsilon$, print a message or raise an exception to indicate that there is no suitable match for $\mathbf{g}$ in the database. In this case, add $\mathbf{g}$ to the database for future reference.

- Generalize the system by turning it into a k-neighbors classifier. In the constructor, add several faces per person to the database (this requires modifying `get_faces()`). Assign each individual a unique ID so that the system knows which faces correspond to the same person. Modify the method from Problem 5 so that it also accepts an integer k, then use `scipy.spatial.KDTree` to find the k nearest images to $\mathbf{g}$. Choose the ID that belongs to the most nearest neighbors, then return an index that corresponds to an individual with that ID.

  In other words, choose the k faces $\mathbf{f}_i$ that give the smallest values of $\|\mathbf{f}_i - \widehat{\mathbf{g}}\|_2$. These faces then get to vote on which person $\mathbf{g}$ belongs to.

- Improve the user interface of the class by modifying the method from Problem 6 so that it accepts a file name to read from instead of an array. A few lines of code from `get_faces()` or `sample_faces()` might be helpful for this.

Other Methods for Facial Recognition

The method of facial recognition presented here is more formally called *principal component analysis (PCA) using eigenfaces*. Several other machine learning and optimization techniques, such as linear discriminant analysis (LDA), elastic matching, dynamic link matching, and hidden Markov models (HMMs) have also been applied to the facial recognition problem. Other techniques focus on getting better information about the faces in the first place, the most prevalent being 3-dimensional recognition and thermal imaging. See `https://en.wikipedia.org/wiki/Facial_recognition_system` for a good survey of different approaches to the facial recognition problem.