EIGEN-FACE RECOGNITION AND IMPROVEMENTS

FERNANDO DEL CASTILLO, JOSEPH RUBIN

1. INTRODUCTION

We implement "eigenfaces", as described in our references (see the end of this paper), and introduce a Gaussian Blur filter to attempt to improve the method. The goal of our project program is as follows: The program is given a collection of images from 15 different people. Then, the program is given a new image of each person (15 total), and it must classify those new images by determining which person is depicted in each new image. To measure success, we count the number of correct classifications, expressed as a percentage of the total 15. We'll state our conclusion here, before exploring the math – our program is able to correctly classify $12/15 = 80\%$ of the new images, where each new image contains a neutral facial expression of its subject. Even better, however, is that our program is able to identify $15/15 = 100\%$ of the images for some of the other facial expressions. We consider our program to be quite successful, given the small size of its data-set. In a real-world application, the data set is likely to be larger, and each newly classified image would be added to the data set. We implement the technique of "eigenfaces" using numpy, and show that our final result is able to achieve satisfactory accuracy, even among small changes in lighting and position, and even with a very small data set. Then, we go further, showing that applying a Gaussian Blur filter to the test faces before classification improves the classification rate, and each expression is classified better or at the same rate than without the filter. (e.g. our $12/15$ ratio is improved to $13/15 = 86.66\%$). Finally, we note that leaving out some of the training data can often improve the classification rate – while we make no attempt at implementing it, an algorithm that weighs each face in the training data according to its relevance may have an optimal classification rate.

2. METHOD

2.1. **Key Concept.** A simple algorithm that would allow for face recognition, and more broadly, for pattern recognition, is one of template matching: each pixel in an image is represented by an 8-bit gray-scale intensity value (0-255), so a basic matching algorithm would compare the intensity values of every pixel in two images. However, this method is very inefficient (and probably not very effective anyway). If we consider two images A and B of the same pixel dimension (say, $x \times y$), both transformed to be 8-bit gray-scale intensity values, we can convert each image into a single column vector of dimension $x \cdot y \times 1$, which allows for element-wise comparison of the two images. For small images, this method is feasible, but for large images, efficiency problems arise. If we let $x = y = N$ so that the image is square (we do this for clarity, images do not need to be square for algorithm to work), we see that the size of our column vector will be N^2. Essentially, our column vector becomes a point in N^2 space. An individual image maps to an individual point in this space. For very large images, element-wise comparison becomes slow and inefficient, and thus we requires a new method. One means of accomplishing this goal is to map our data onto a lower-dimensionionality space, which leads to the idea of principal component analysis. In our

FIGURE 1. First 5 eigen-faces of data set

context of face images, principal component analysis attempts to find a collection of vectors (of length N^2) that best describes the known face images. These special vectors are called eigenfaces, and the subspace that they span is correspondingly named the face space. For our training set, the first five of these eigenfaces are shown in figure 1 – we'll talk about their computation in a moment.

2.2. **Calculation of Eigen-faces.** To calculate the eigen-faces, we must first start with a training set of images, a part of which is presented in figure 2. For our project, we used 10 images each of 15 different people for training.
Let the set of training images be $I = \{I_1, I_2...I_M\}$. Then we can easily perceive each of these these images as a column vector size equal to the number of entries in the image matrix (for clarity we once again assume that the size of our images is $N \times N$ such that the size of the column vector is N^2. However, this is not necessary, the images can be of any aspect ratio). To create these vectors, we'll flatten them in row-major order. Thus we have:

$$\{I_1, I_2...I_M\} \Rightarrow \{\vec{L_1}, \vec{L_2}...\vec{L_M}\} \tag{1}$$

Where $\vec{L_i}$ is the corresponding column vector of each image. Now we define the "average face" vector, which is the element-wise average of all image vectors from the training set:

$$\vec{L_{avg}} = \frac{1}{M}\sum_{i=1}^{M}\vec{L_i} \tag{2}$$

Figure 3 shows the average face vector, again formatted as an image.

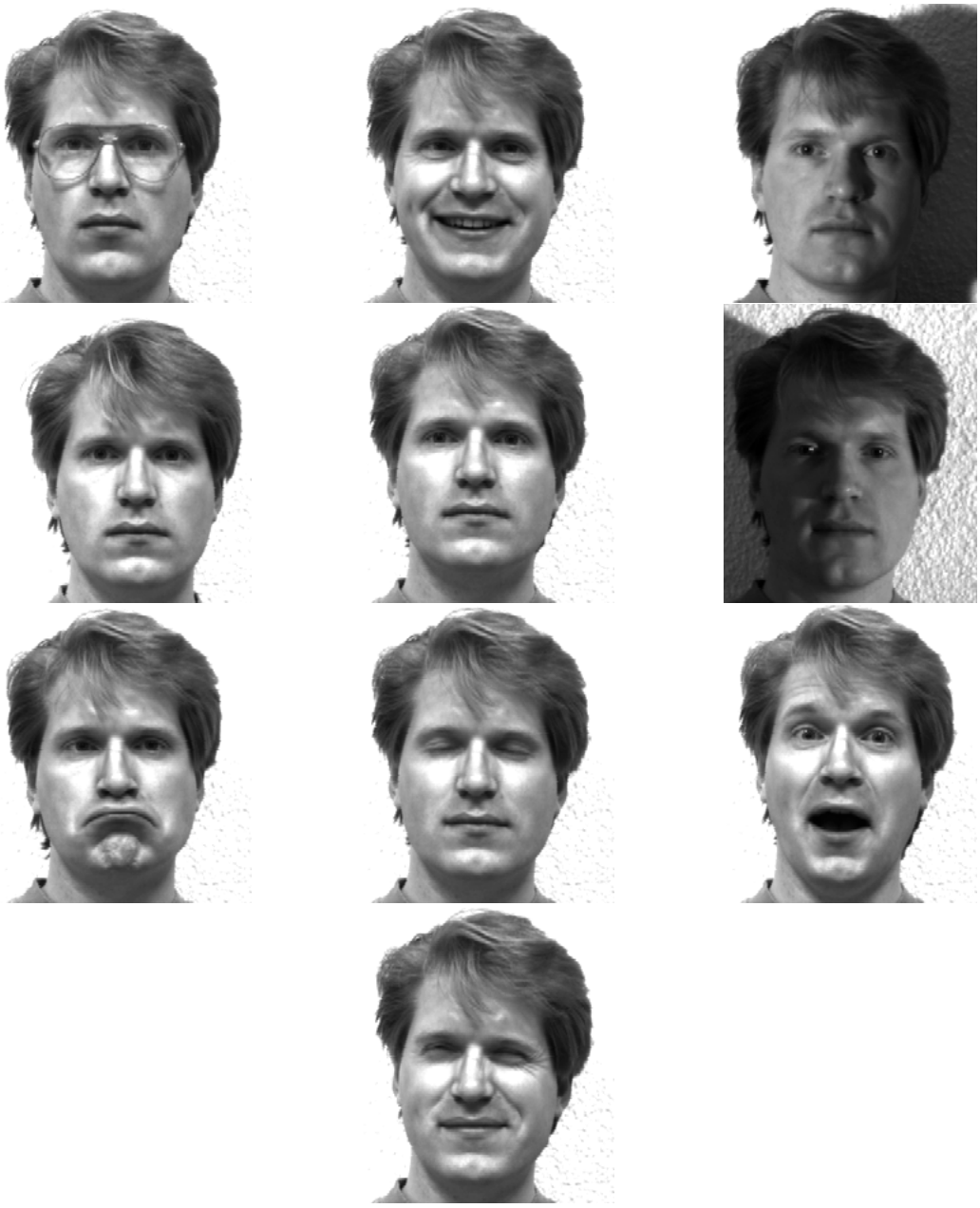


FIGURE 2. Set of images used for the training, all obtained from one individual in varying conditions (lighting, facial expression). This is one of 15 sets.

Figure 3 presents the average face from our data set. Now we normalize all images by computing the difference between all image vectors and the average vector, which we define to be $\vec{a_i}$:

$$\vec{a_i} = \vec{L_i} - \vec{L_{avg}} \tag{3}$$

We can compile each of these column vectors into a matrix A, in which the i^{th} column corresponds to A_i:

$$A = \{\vec{a_1} \ \ \vec{a_2} \ \ ... \ \ \vec{a_M}\} \tag{4}$$

FIGURE 3. The average face from our training set.

Now to this set of vectors, we apply principal component analysis, that is, we find a set of M orthonormal vectors $U = \{u_1 \ u_2 \ ... \ u_M\}$, such that u_k maximizes the value of λ_k in the following condition:

$$\lambda_k = \frac{1}{M}\sum_{n=1}^{M}(u_k^t a_n)^2 \tag{5}$$

where orthonormality is held by the following condition:

$$u_l \cdot u_k = \begin{cases} 1, & \text{if } l = k \\ 0, & \text{if } l \neq k \end{cases} \tag{6}$$

The set U and corresponding values λ_i are the eigenvectors and eigenvalues of the covariance matrix of the original images:

$$C = \frac{1}{M}\sum_{n=1}^{M} a_n a_n^t = AA^t \tag{7}$$

Now all that is left it to actually compute the eigenvectors of our covariance matrix, that is, to find the u_i's. A direct calculation of this, however, is impractical, because AA^t is of size $N^2 \times N^2$. But we don't need to find every eigenvector anyway. We would just like to find the most important ones, the ones whose corrosponding eigenvalues are the largest. For our purposes, we only need about 150, although as we will see, even 20 or so is enough to guarantee good results. Therefore, we choose to work with the matrix A^tA instead of the covariance matrix. Most notably, the eigenvectors of this matrix are related to those of the covariance matrix in the following fashion. Let $\vec{v}$ be a λ - eigenvector of A^tA. Then:

$$A^tA\vec{v} = \lambda\vec{v} \Rightarrow AA^tAv_i = A\lambda\vec{v} = \lambda A\vec{v} \tag{8}$$

So $A\vec{v}$ is a λ - eigenvector of AA^t. So all we need to do is calculate the eigenvectors of A^tA (which is computationally easy, as this matrix is only $M \times M$), then transform each one by A, and we have some eigenvectors of the covariance matrix. Because A^tA is real and symmetric, we are guaranteed M real, orthogonal eigenvectors by the Spectral Theorem.

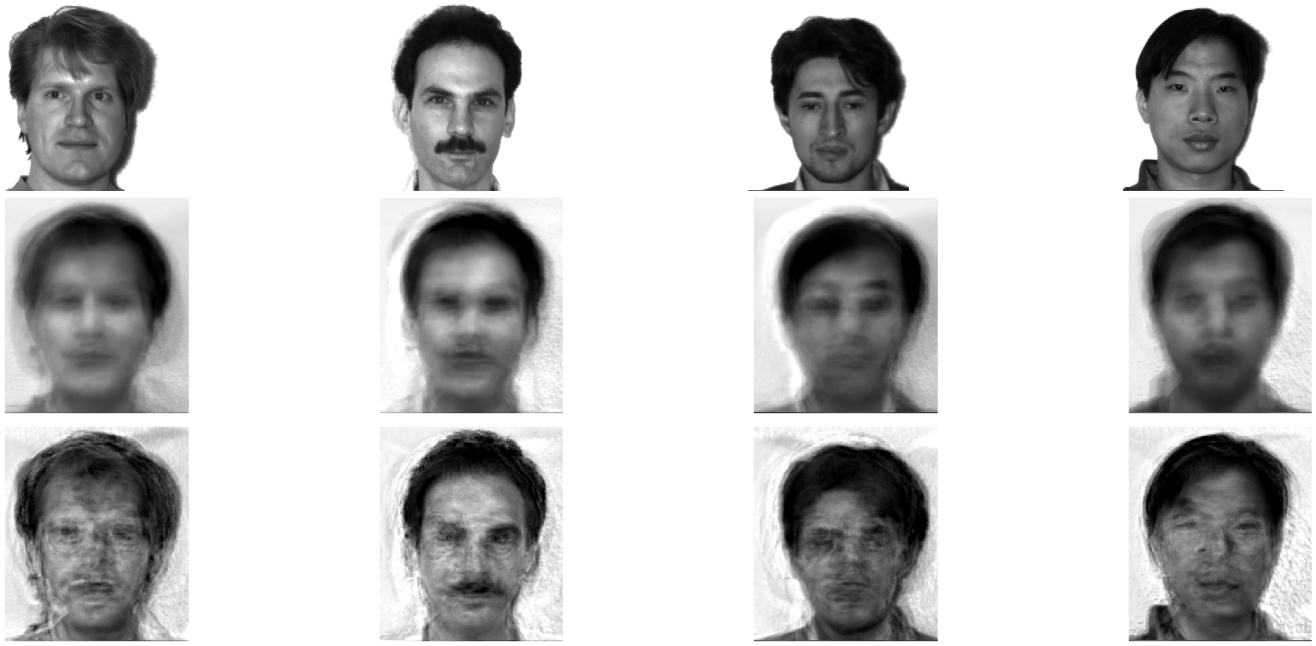
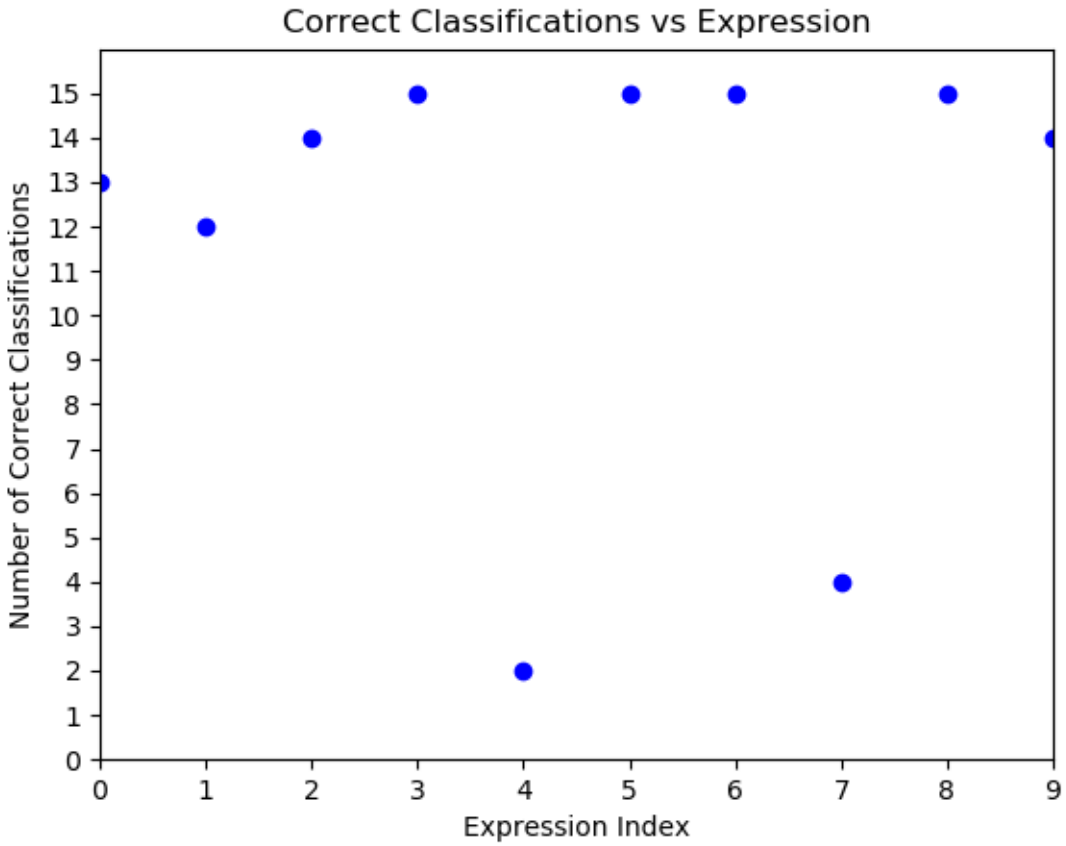


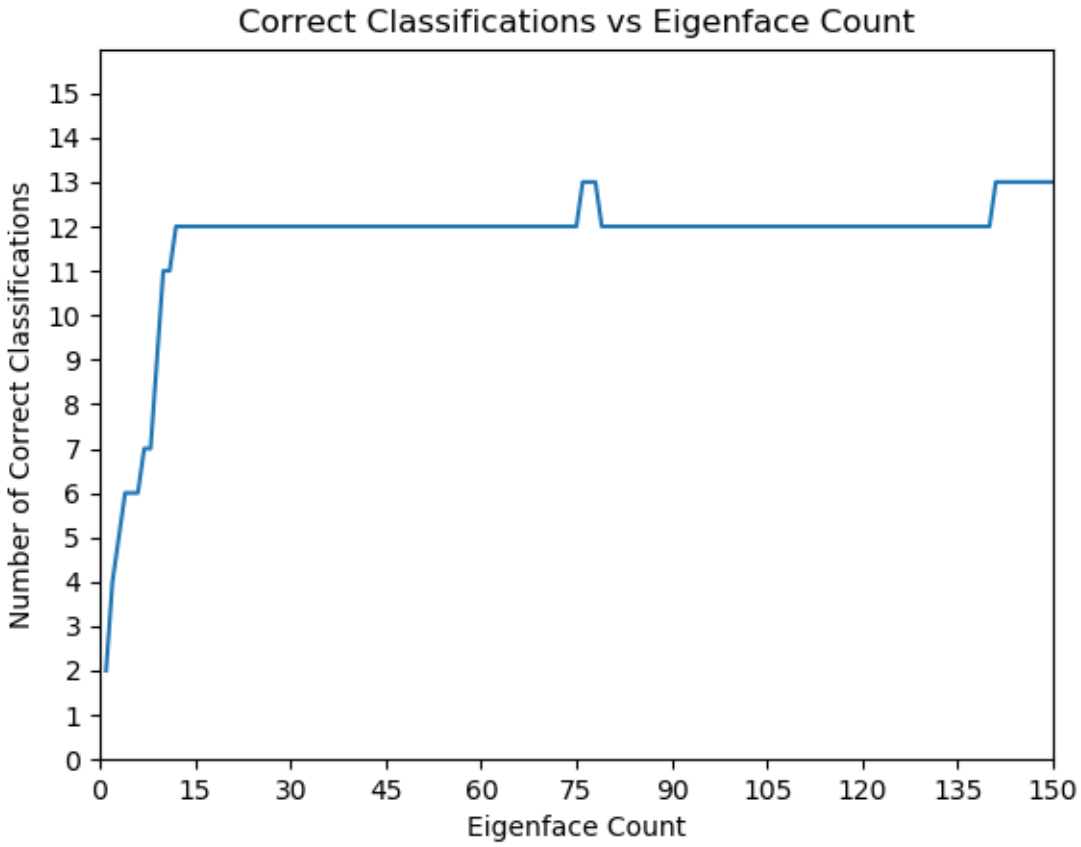
FIGURE 4. Original images (top row), low count (10) eigenvector reconstruction (middle row), and high count (150) eigenvector reconstruction (bottom row) for images not in the training set of images. Reconstructions represent projections on to the face space.

Now we have the eigenvectors of our covariance matrix, and therefore can arbitrarily choose a certain number K of those eigenvectors, corresponding to the largest K eigenvalues (the higher K is, the less efficient the method is, but the greater accuracy of our program. Each of our original faces from the training set can be represented as a linear combination of the K eigenvectors chosen. In order to compute the coordinates of each face with respect to the eigenvectors, we form a matrix, B, where each row is an eigenvector. Then we can compute the coordinate vector $\vec{w}$ of a face, $\vec{a}$, which represents a projection of that face onto a lower dimension, by simply computing: $\vec{w} = B\vec{a}$. We can then reconstruct the face from its coordinates by computing $\vec{w^t}B$. This reconstruction will never look exactly like the original image, but using more eigenfaces will increase the authenticity of the reconstruction. Figure 4 shows the results.

2.3. **Face Recognition.** For our face recognition purposes, we wish to match new images to a person in the training set. We construct a face class for each individual in the training set, which is computed by finding the coordinate vector, $\vec{w}$, for each image of a particular individual, and then averaging them. This is the mean projection of an individual onto the face space.
Then, for each test image, we compute the same projection. Now, the face matching is easy. The coordinate vectors are of a low dimension, so we can compute the euclidean distance between the coordinate vector of the test image and each face class, in turn, and classify the new face based on which distance is the lowest.
Our results are summarized below. Expression 0 is neutral, and is thus the most important result to consider, at $13/15$. Some expressions achieved $15/15$, and only a couple achieved low rates of success, probably because of the wildly varied mouth position in expressions 4 and 7 for many subjects.



For interest, the following graph (for expression 0), shows the success as a function of the number of eigenfaces used. It can be seen that even using very few eigenfaces achieves quite a good result.



2.4. **Python Implementation.** To run the program, see the instructions in the readme.txt for installing the required packages. Then run python on main.py. It will run the eigenfaces program to classify expression 0. It should print out the following result:

```
Images loaded.
Eigenfaces computed.
Face classes computed.
Trial: 1, best class match: 1
Trial: 2, best class match: 2
Trial: 3, best class match: 3
Trial: 4, best class match: 4
Trial: 5, best class match: 5
Trial: 6, best class match: 1
Trial: 7, best class match: 4
Trial: 8, best class match: 8
Trial: 9, best class match: 9
Trial: 10, best class match: 10
Trial: 11, best class match: 11
Trial: 12, best class match: 12
Trial: 13, best class match: 13
```

```
Trial: 14, best class match: 14
Trial: 15, best class match: 15

Number correct: 13 / 15
Percentage: 86.66666666666667%
Eigenfaces used: 150

Process finished with exit code 0
```

We found that applying a Gaussian Blur to the test images before projecting them onto the face space and doing the classification improved the results. This is likely because the blur eliminates stark outlying data (noise), and reduces the image to a simpler form which preserves unique face characteristics. The intensity of the blur can be modified in const.py. The idea to add the blur is discussed sometimes in papers relating to eigenfaces, but our references do not test it. We are happy that it was a successful idea.

The code is straightforward – main.py drives the program, while compute.py is responsible for most of the linear algebra. const.py allows many constants to be toggled (how many eigenfaces to use, which expression to classify, how much blur to use, etc.). The code follows the mathematical theory outlined above. The repo may be found at https://github.com/josephrubin/eigenfaces, and it is also included in the .zip archive.

REFERENCES

[1] Turk, M. Pentland, A. (1991). Eigenfaces for recognition. J. Cognitive Neuroscience, 3, 71–86. doi: http://dx.doi.org/10.1162/jocn.1991.3.1.71

[2] Kutz, J. N., Kutz, J. N. (2013), *Data-driven modeling scientific computation: Methods for complex systems big data*. Oxford: Oxford University Press.

[3] http://www.vision.jhu.edu/teaching/vision08/Handouts/case_study_pca1.pdf

3. SUPPLEMENTAL INFORMATION

E-mail address: fadc@princeton.edu, jmrubin@princeton.edu