Eigenfaces

A Mathematical Overview and Implementation

Ismail Mustafa

Courant Institute of Mathematical Sciences
New York University
United States
December 17, 2015

Abstract

This paper looks at eigenfaces as described by the 1991 paper by Turk and Pentland (3). This paper begins with a mathematical background on principal component analysis as well as eigenvectors and eigenvalues. This is done in order to ensure the reader develops an intuition as to how and why eigenfaces work the way they do. An implementation of eigenfaces in Matlab will then be covered and code samples will be provided. Finally, sample output will be shown on two different test image sets. The first image set is the "labeled faces in the wild" corpus (6). The second is a collection of 36 faces as well as images with multiple pages from Washington University's computer science department (7).

Introduction

The understanding and interpretation of human faces, otherwise known as facial recognition is a task that comes easily to humans, but presents a mathematical challenge to researchers. Nevertheless, great advances have been made in this field. Beginning in the 1960s, semi-automated systems were developed that required operators to input specific facial features such as the eyes, nose and mouth. Distances and ratios were then calculated and compared to reference data that was measured before hand. In the 1970s, Goldstein, Harmon, and Lesk used 21 specific facial markers to automate recognition. Unfortunately, these approaches required a lot of manual labor (1). It wasn't until 1988 when Sirovich and Kirby used principal component analysis (PCA) for the task of facial recognition that the field was spurred again (2). Shorty after that in 1991, Turk and Pentland used PCA to calculate what they called "eigenfaces" and used the residual error to detect faces in images (3). In 1993, the Face Recognition Technology Evaluation (FERET) which was sponsored by DARPA was created. The FERET program lasted until 1997 and had one major goal, which was to spur the development of face recognition algorithms. In 2002, the Face Recognition Vendor Tests (FRVT) was created by NIST as a method of evaluating currently available commercial methods of facial recognition. In 2006, the Face Recognition Grand Challenge (FRGC) was created by NIST in order to further evaluate current standards in face recognition algorithms (1). Nowadays, current research into face recognition involves novel techniques such as deep convolutional neural networks. An example of this would be Google's FaceNet (4). In this paper, we'll look more closely at Turk and Pentland's eigenfaces approach to facial recognition and detection and present an implementation of this algorithm in Matlab. This paper also serves as an introduction to principal component analysis and an explanation of eigenvalues and eigenvectors in order to understand the intuition behind this method.

Mathematical Background

Eigenvectors and Eigenvalues

An eigenvector is a vector that does not change direction when undergoing a linear transformation. While its direction cannot change, it's length can. The multiple by which the eigenvector is scaled is called the eigenvalue. An example eigenvector and associated eigenvalue can be seen below:

$$\begin{pmatrix} 2 & 3 \\ 2 & 1 \end{pmatrix} \times \begin{pmatrix} 3 \\ 2 \end{pmatrix} = 4 \times \begin{pmatrix} 3 \\ 2 \end{pmatrix}$$

Here we see that when we multiply the vector $(3\ \ 2)^T$ by the transformation matrix on the left hand side, we get a vector with the same direction, but scaled up by a multiple of four. In this case, the eigenvector is $(3\ \ 2)^T$ and the eigenvalue is 4. It is important to know that eigenvectors can only be calculated for square matrices, and that not all square matrices have eigenvectors. In addition to this, if an $n \times n$ matrix has eigenvectors, it will have n eigenvectors. Another interesting property which will come in handy when discussing principal component analysis is that all eigenvectors of a matrix are orthogonal to one another. This way, we can "plot" data on the axes of these eigenvectors in much the same way that we plot data on x, y, and z axes due to their orthogonality. Given that a matrix has eigenvectors, it is fairly easy to calculate them for very small matrices using the following equation:

$$Ax = \lambda x$$

where A is the matrix whose eigenvectors we wish to calculate, λ is the eigenvalue we want to find the eigenvector for, and x is the eigenvector we are trying to calculate.

Principal Component Analysis

Now that we have an intuitive understanding of eigenvalues and eigenvectors, it will be easier to grasp how PCA works. The main purpose of PCA is to be able extract the important parts of high dimensional data since it can be hard to extract patterns otherwise. Essentially, PCA will tell you what the most important variables in the data are. As a result, PCA is very useful for dimensionality reduction. If we find that a certain variable in the data-set contributes minimally to the variance of the data, we can simply dispose of its principal component thereby reducing the dimensionality of the data-set.

At a very high level, PCA takes observations with variables that could be correlated and turns them into linearly uncorrelated variables. These linearly uncorrelated variables are called principal components. The first principal component has the largest variance so that it accounts for most of the variability in the data. The second principal component has the second highest variance but with the constraint that it is orthogonal to the first principal component. This pattern continues for all the remaining principal components. The principal components are all orthogonal to one another because they are the eigenvectors of the covariance matrix which was used to calculate them. Now that we know what eigenvectors are, we can view PCA as simply "shifting" the data from one set of axes to another, except that this new set of axes is ordered by most important in terms of variance to least important.

How to Use Principal Component Analysis

In order show the steps of how to use PCA, we look at a simple data-set:

$$x = 1, 2$$

$$y = 2, 8$$

First, we normalize each variable's data by subtracting each data point by its respective mean:

$$x = \frac{1}{\frac{1+2}{2}}, \frac{2}{\frac{1+2}{2}}$$

$$y = \frac{2}{\frac{2+8}{2}}, \frac{8}{\frac{2+8}{2}}$$

Then we place the data into a matrix where each column represents an axes:

$$\begin{matrix} x & y \end{matrix}$$
$$A = \begin{pmatrix} 0.667 & 0.4 \\ 1.333 & 1.6 \end{pmatrix}$$

We can then calculate the covariance matrix using the following equation:

$$C = \frac{1}{M}\sum_{n=1}^{M} v_n v_n^T = \begin{pmatrix} 0.2218 & 0.3996 \\ 0.3996 & 0.72 \end{pmatrix}$$

We use Matlab to calculate the eigenvectors and eigenvalues. For small matrices it can be done by hand, but for larger matrices, it starts to get difficult. Calculating the eigenvalues and eigenvectors in Matlab gives us:

$$eigenvalues = \begin{pmatrix} 0 \\ 0.9418 \end{pmatrix}, \quad eigenvectors = \begin{pmatrix} -0.8744 & 0.4853 \\ 0.4853 & 0.8744 \end{pmatrix}$$

where the columns of the eigenvector matrix are the two eigenvectors of the covariance matrix. The next step is to create a feature vector in which the columns are the eigenvectors we wish to keep. In order to determine this, we first sort the eigenvectors by largest eigenvalue. The second column corresponds to eigenvalue 0.9418 so we reorder the eigenvector matrix to the following:

$$\begin{pmatrix} 0.4853 & -0.8744 \\ 0.8744 & 0.4853 \end{pmatrix}$$

Looking at the data qualitatively, we can decide which eigenvectors actually have an impact on the data. We could decide to then reduce the dimensionality of our data set by keeping the first k eigenvectors. In this case, we would create a feature vector containing the largest eigenvector:

$$F = \begin{pmatrix} 0.4853 \\ 0.8744 \end{pmatrix}$$

Finally, we can get a new more "revealing" data-set by multiplying the transpose of the feature vector by the transpose of our normalized data matrix:

$$CD = F^T \times A^T = \begin{pmatrix} 0.6735 & 2.0459 \end{pmatrix}$$

Where each row represents a different dimension, namely the dimension corresponding to the eigenvector we have decided to keep. In this case, CD is the compressed data. We have effectively reduced the dimensionality of the data from 2 dimensions to 1 dimension. When dealing with actual data that gives rise to certain insights, it is important to play around with the number of dimensions that you are reducing the dimensionality to. If too few eigenvectors are kept, a lot of information can be lost, however if too many are kept, then redundant dimensions are kept which could lead to inefficiencies and a failure to bring about reasonable insights from the reduced dimensionality data. What if we need to get the original data back? This is important for applications such as data compression. We simply do the following:

$$A^T = F^{-1} \times CD$$

However, the reconstructed matrix A will not contain all the data if we have chosen to only represent the data of a few principal components rather than all of them (5).

Eigenfaces

Calculating Eigenfaces is not much different from the steps we have taken above. There are two main differences. The first is in how we pre-process the image data in order to calculate the covariance matrix, and the second is in the actual calculation of the covariance matrix that will prevent the computation from being intractable. The notation used to explain the calculation of eigenfaces will be the same as in the 1991 Turk and Pentland paper (3).

For each image, we compress the image to a small size to save computational time. In our case, this is 50×50 pixels. This dimension is denoted as N. Each image is then represented as an $N \times N$ matrix and converted to gray-scale. We then convert each $N \times N$ matrix into an $N^2 \times 1$ column vector where each row is reshaped into the column vector as seen in Figure 1.

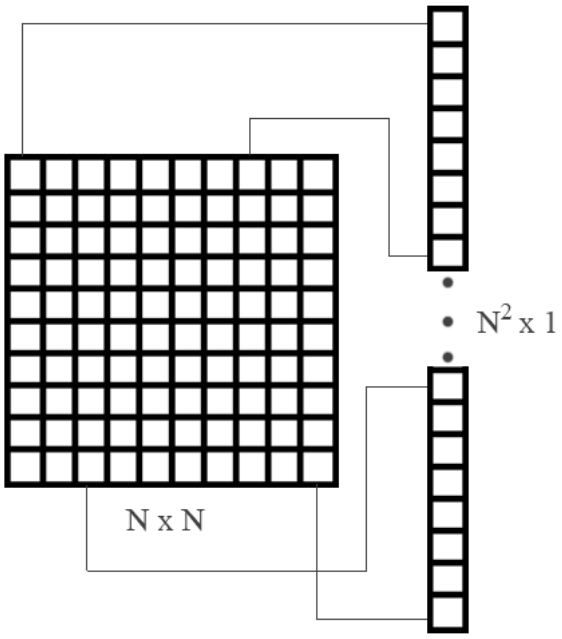

Figure 1: Converting a matrix representation of an image to a vector

For convenience, we label each $N^2 \times 1$ image vector as Γ_n. In order to normalize the data, we calculate the mean of the images in the training set to get the mean face μ. This can simply be done by adding up all the image vectors and dividing by the total number of image vectors in the training set. If we take the number of images in the training set to be M:

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

Finally, we subtract the mean from each image vector:

$$\Phi = \Gamma - \mu$$

where Φ is the normalized face. We then construct a matrix A of size $N^2 \times M$ where each column of A is the normalized face vector. Now that we have our data normalized in matrix A, we perform principal component analysis by first calculating the covariance matrix. Using the same formula as before:

$$C = \frac{1}{M} \sum_{n=1}^{M} \Phi_n \Phi_n^T = AA^T$$

The covariance matrix we calculated above would yield a matrix of size $N^2 \times N^2$. We know that if we calculate the eigenvectors of this matrix, we would get N^2 eigenvectors. Even for small images of 50×50 pixels, this computation quickly becomes intractable. We get around this problem by considering the matrix $A^T A$. Let us plug the matrix $A^T A$ into the equation mentioned in the mathematical background section for calculating eigenvectors:

$$A^T A x_i = \lambda_i x_i$$

We can multiply both sides by A to get:

$$AA^T A x_i = \lambda_i A x_i$$

Realizing that $C = AA^T$ in this equation, we simplify to the following:

$$CAx_i = \lambda_i Ax_i$$

From this we realize that $y_i = Ax_i$ is the relationship between the y eigenvectors of AA^T and the x eigenvectors of A^TA. Therefore, calculating the eigenvectors of the much more tractable matrix A^TA of which the dimensions are $M \times M$ gives us the M largest eigenvectors of the AA^T matrix. This is great because the computational complexity of calculating the eigenvectors of these eigenfaces increases linearly with the number of training samples used, regardless of the size of the image. It may be easy to think that a lot of information is lost by only considering the M best eigenvectors since $M << N^2$, but in practice, only k eigenvectors are used to represent the facial data where typically $k < M$. Now that we have selected the k best eigenvectors where k is heuristically selected based on qualitative analysis of the training set, we can represent any face as a linear combination of these k eigenvectors using the following equation:

$$\Phi_i - \mu = \sum_{j=1}^{k} x_j^T \Phi_i x_j$$

If we take $w_j = x_j^T \Phi_i$, we can represent each normalized face as a column vector of weights:

$$\Omega_i = \begin{pmatrix} w_1^i \\ ... \\ w_k^i \end{pmatrix}$$

Now that we can represent faces as a linear combination of the k largest eigenvectors, we can use these weights to recognize and detect faces in images. Let's say we have a new image of a person that we know we have in the training data-set, and we want to be able to identify this person. We would first normalize the input image as we have done before by creating an image vector. We would then project it onto the eigenspace using the equation above. From this, we get the weights of the image and we calculate the euclidean distance between these sets of weights and the weights for all the images in the training set. We then choose the image that gave us the smallest euclidean distance to the recognized face. In order to detect faces in images with multiple faces, we take a portion of the image and compute the normalized face as well as the set of weights corresponding to this face. We then find the euclidean distance between this face the rest of the weights in the data-set and compare this distance to a threshold determined by the training set as well as heuristically. If the distance is less than this threshold, we take the image to be a face.

Matlab Implementation and Results

Eigenface Calculation

For facial recognition, we use 16 images of celebrities from the "labeled faces in the wild" dataset. We first preprocess the images by resizing them to a standarized size of 50×50 pixels. We also make them grayscale and normalize each pixel to a value between 0 and 1. The code and resulting image is shown below:

```
img = rgb2gray(img);
img = imresize(img,N,N);
img = double(img)/255.0;
```

Figure 2: Resized and grayscale images before PCA

We then calculate the mean image of the faces as follows:

```
sumImage = zeros(N);
for i = 1:M
    sumImage = sumImage + images.data{i};
end
meanImage = sumImage/M;
```

Figure 3: Mean face of all faces in training set

We then normalize each face by subtracting the mean:

```
for i = 1:M
    images.dataMean{i} = images.data{i} - meanImage;
end
```

Figure 4: Normalized faces

The next step is to generate matrix A where each column is the normalized image vector. From this we calculate the eigenvectors and eigenvalues. We then sort all M eigenvectors in terms of the largest eigenvalues:

```
% Generate matrix A (covert dataMean into N^2 vectors)
A = zeros(N*N,M);
for i = 1:M
    A(:,i) = images.dataMean{i}(:);
end

% Covariance matrix
C = transpose(A)*A;

% Calculate eigenvectors and eigenvalues
[V,D] = eig(C);

% Compute M best eigenvectors
Mbest = A*V;

% covert back into NxN eigenfaces
for i = 1:M
    column = Mbest(:,i);
    eigenface = reshape(column, N,N);
    images.eigenface{i} = eigenface;
end

% Sort eigenfaces in terms of largest eigenvalue
eigenvalues = diag(D);
[B,I] = sort(eigenvalues,'descend');
eigenfacesOrdered = cell(1,M);
for i = 1:M
    eigenfacesOrdered{i} = images.eigenface{I(i)};
end
```

Figure 5: Eigenfaces sorted in order of largest eigenvalue starting from the top left

For each eigenface in the above figure, we can see the most relevant features being captured. The light features in the images are what have the largest covariance from face to face. From these images, we can see that the hair, eyes, eyebrows, nose, and mouth are the features that vary the most. Intuitively, this makes sense since these are the features we tend to focus on, as opposed to the cheeks, chin, or forehead. Now that we have these eigenfaces, we can calculate the corresponding weights for each image:

```
for i = 1:M
    for j = 1:k
        weights(i,j) = sum(transpose(eigenfacesOrdered{j}(:)) * A(:,i));
    end
end
```

In this case, we have picked the value of k to be 4 since we saw no improvement selecting more than this number of eigenvectors. This generates a matrix of size 16×4 where each row are the corresponding four weights for each image required to project the original image onto the eigenface space that we created.

Face Recognition

We can now use the following code to pass in an new image of a person contained within the data set to check if they can be recognized:

```
% process image
img = imread(filePath, 'ppm');
img = rgb2gray(img);
img = img - meanImage;
img = double(img)/255.0;
img = imresize(img, N, N);
img = img(:);

% project image onto eigenface space and get weights
for i=1:k
  imgWeights(i)  = sum(transpose(eigenfacesOrdered{i}(:)) * img);
end

% calculate euclidean norm for input image weights and training set weights
euc = zeros(M,1);
for i=1:M
    diff = imgWeights(1,:) - weights(i,:);
    euc(i,:) = norm(diff);
    weights(i,:);
end

% sort names in order of least euclidean distance
[sortedEuc,eucIndexOrder] = sort(euc,'ascend');
orderedNames = cell(1,M);
for i = 1:M
    splitName = strsplit(images.name{eucIndexOrder(i)},'_');
    name = [splitName{1}, ' ', splitName{2}];
    orderedNames{i} = name;
end
splitAnswer = strsplit(images.name{eucIndexOrder(1)},'_');

% solution is the guessed name
solution = [splitAnswer{1}, ' ', splitAnswer{2}];
```

As a test, we pass in a different image of Jack Straw who is already present in the data set, and as output, we get the following:

```
sortedEuc = 39.4143
            47.6421
            52.2216 ...

 solution = Jack Straw
```

The sortedEuc vector represents a sorted vector of euclidean distances of the weights of the input image to each of the weights in the training set. In this case, the lowest euclidean distance was found to be 39.4143. This value is the distance between the input weights and the weights of Jack Straw's training image.

Face Detection

A subjectively more interesting problem is detecting faces in an image. For this problem, we use a data-set of 36 faces from the University of Washington computer science department (7). Face detection is implemented in two steps. The first step is to calculate a threshold value from the 36 images to determine if an image is indeed a face. The following code iterates through all 36 images, calculates the euclidean norm between its weights and the weights of all the other images in the training set. It then does this for each image, and takes the average value of all of these euclidean norm averages. The threshold is then heuristically tuned by a multiplicative coefficient to achieve reasonable results:

```
 % Calculate threshold to determine if face is an image
averageOfAverageNorm = 0;
for i = 1:36
  filePath = ['faces/face', num2str(i), '.jpg'];

  % euclidean distance average between image weights and every other image
  averageNorm = isFace(filePath, ...
                       eigenfacesOrdered, ...
                       weights, ...
                       images, ...
                       meanImage, ...
                       k, ...
                       M, ...
                       N);
  averageOfAverageNorm = averageOfAverageNorm + averageNorm;
end

% Threshold with multiplicative coefficient heuristically tuned
threshold = (averageOfAverageNorm/36)*0.74;
```

The second step is to take an input image with multiple faces and iterate through it using a heuristically determined scale. For each portion of the image, the average euclidean norm is calculated between the input image and the weights of the faces in the training set. A face is then detected if the average norm of the input image is less than the threshold value calculated in the previous step. once a face is found, a square outline is drawn around the location of the face and the image is displayed. The face detection code shown below:

```
% process input image
img = imread(filepath, 'jpg');
img = double(img)/255.0;
img = rgb2gray(img);
[height, width] = size(img);

% Heuristically set scale
scale = 1.3*N;
widthLimit = width-scale;
heightLimit = height-scale;

% iterate through every possible sub image of large image using scale
allFaces = [];
for i = 1:heightLimit
  for j = 1:20:widthLimit
      c = img(i:i+scale-1,j:j+scale-1);
      averageNorm = isFaceTest(c, eigenfacesOrdered, weights, ...
                     images, meanImage, k, M, N);
    % Test condition to find face
    if averageNorm < threshold
         foundFace.y = i;
         foundFace.x = j;
         foundFace.width = scale;
         foundFace.height = scale;
         allFaces = [allFaces ; foundFace];
         break
    end
  end
end

% Draw face locations on image
figure(24), imshow(img,'Initialmagnification','fit'); title('before')
hold on
[sizeAllFaces, ~] = size(allFaces)
for i = 1:sizeAllFaces
    foundFace = allFaces(i);
    rectangle('Position', [foundFace.x foundFace.y foundFace.width foundFace.height],
    'EdgeColor', 'r', 'LineWidth', 3);
end
```

The algorithm is optimized so that it only searches the specified scale every 20 pixels as opposed to incrementing by only 1 pixel. The scale was also set to 1.3 times the pixel size of the image because this was found to produce the best results. The following images show the reasonable performance of this facial detection algorithm:

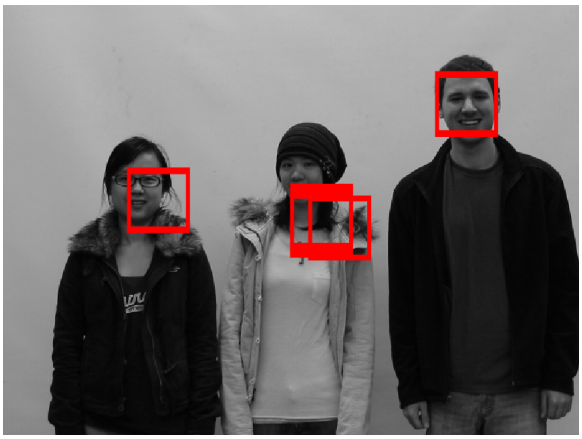

Figure 6: Face detection test in which the detection is slightly off

Figure 7: Face detection test in which one face goes unnoticed

In Figure 6, we see the person's face on the right identified perfectly whereas the person on the left wearing glasses has the wrong portion of their face identified. This could be due the the glasses being worn by the person as it is probably not a feature captured by the eigenvectors. The person in the middle interestingly has their right shoulder identified yet it is quite close to their face. An explanation of this could be due to the beanie this person was wearing. Since hair is what varies the most with eigenfaces, the beanie could have thrown the algorithm off. In Figure 7, we see the left two faces identified fairly well whereas the face on the right is completely missed. We could assume this was due to the glasses but the person on the left of Figure 6 was identified even while wearing glasses. Another explanation could be due the angle the person has their head tilted at. In order to effectively detect faces uses PCA, every face must be perfectly straight and facing the camera.

Conclusion

The purpose of this paper was to show a Matlab implementation of eigenfaces as introduced by Turk and Pentland (3), as well as provide a mathematical overview of the topics involved in calculating these eigenfaces. The topics that were explained were principal component analysis, eigenvectors and eigenvalues as well as their relationships. These topics provided an intuition for how eigenfaces worked, and why they were effective at capturing the most relevant features of a human face. The Matlab implementation covered two interesting applications of eigenfaces, the recognition of faces as well as the detection of faces from an image containing multiple faces.

References

References

[1] "Facial Recognition." FBI. FBI, 29 Jan. 2013. Web. 17 Dec. 2015.

[2] Sirovich, L., and M. Kirby. "Low-dimensional Procedure for the Characterization of Human Faces." *Journal of the Optical Society of America A J. Opt. Soc. Am. A* 4.3 (1987): 519. Web.

[3] Turk, M.a., and A.p. Pentland. "Face Recognition Using Eigenfaces." Proceedings. 1991 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (1991): n. pag. Web.

[4] Schroff, Florian, Dmitry Kalenichenko, and James Philbin. "FaceNet: A Unified Embedding for Face Recognition and Clustering." 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2015): n. pag. Web.

[5] Smith, Lindsay I. A Tutorial on Principal Components Analysis. N.p.: n.p., n.d. University of Otago: Computer Science. 26 Feb. 2002. Web. 10 Dec. 2015.

[6] "Labeled Faces in the Wild Home." LFW Face Database : Main. University of Massachusetts Amherst, n.d. Web. 7 Dec. 2015.

[7] Simon, Ian. "CSE 455 Project 2: Eigenfaces." CSE 455 Project 2. University of Washington, 22 Jan. 2014. Web. 13 Dec. 2015.