Principal Components Analysis

- PCA -

Buzatu Rareş Tudor
323AC, ACS, UPB

4 March 2018

Contents

1 Abstract

This article is supposed to bring some light in terms of acknowledge the statistical procedure of Principal Component Analysis, or PCA. After I have completed a course of Numerical Methods, I thought about making an application of facial recognition based on PCA, so in this paper I try to describe and formulate this mathematical solution for this kind of problems. The basic idea on which the technique is based consist in reducing the number of dimensions of a problem to a smaller number from which we can draw some results. These problems are met in various fields, from medical, to sport and engineering and by solving them we can improve different aspects related to statistical matters or optimization.

2 Introduction

Principal Components Analysis is defined as a mathematical solution for statistical problems. As stated in the previous paragraph, it uses some mathematical " formulas " and " properties ", which we are going to approach a little later in this paper, to simplify the dimensions of a given problem to a form of fewer linear combinations between them. For the moment lets have a look on some practical problem, met nowadays in medical field. By solving these problems we can optimize the process or we can better understand it.

3 Real life problem

3.1 Problem description

In medical field this was a real problem that doctors had to answer it.
The genetic code is the set of rules by which information encoded within genetic material : deoxyribonucleic acid(DNA) and ribonucleic acid(RNA) is translated into proteins by living cells. Researchers found out that in one single-cell RNA sequence there are more than 10,000 transcribed genes and each cell in our body contains information about our genes. The problem was: Does related cells (200 types) contains the same genes?
To answer the question you would have to plot on a 200 dimensional space each gene and then observe any similarity between them. Or you could use PCA and reduce your problem on a 2 or 3 dimensional space that you could actually understand. Let's see how to do that:
To make the problem easier, we will leave the premise that there are only 2 cells, each containing different quantities of genes:

Gene	Cell 1	Cell 2
A	10	8
B	0	2
C	14	10
...	...	...

Now we have to plot the system:

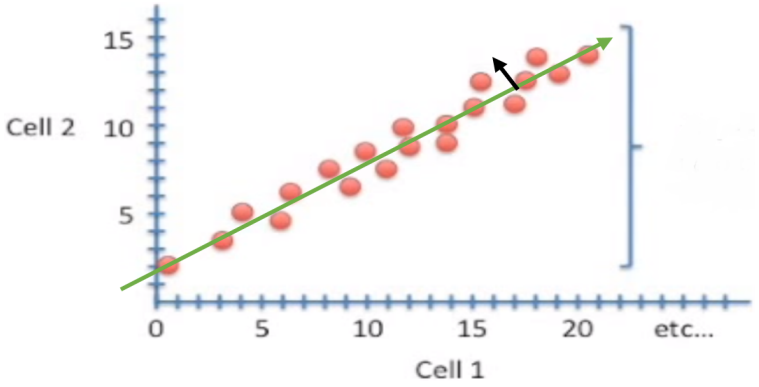

Figure 1: Positive linear distribution

3.2 Linear distribution

We can observe that the distribution in which the data-points(cells) are placed, forms a positive linear distribution. By rotating the axis anti-clockwise with an angle equal to the angle formed by the green arrow and the OX axis, we will observe that the points are almost placed on one axis, so we could eliminate the other one. However, by doing this we will lose some information about the graph, but that information is to small and it will not affect us(we will not be able to see which points are "upper" and which are "lower").
We can see that all of the points, or the majority of them, are placed along the green arrow. In this case the arrow defines one of the principal components of the system.
The maximum number of components of a system is determined by the number of dimensions: in this case there are two dimensions, so there are two principal components. These components are orthogonal and with a linear combination of them we can recreate each of the graph points.
For example, the other principal component could be the black arrow. Now we can see that the green arrow is bigger that the black arrow, that means that the green arrow contains more information about the problem compared to the green one. So we can eliminate the dimension corresponding to the black arrow and our information will be placed along one single axis.

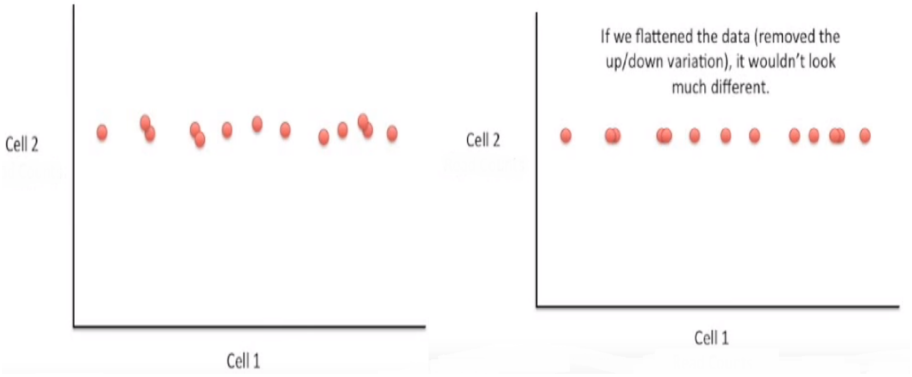

Figure 2: 2D plot vs 1D plot

In the above example, the distribution was very obvious, but in real problems the points could be more spread along multiple axis and we can not spot them.

3.3 What give the direction for the biggest principal component?

The answer is the most "extreme" values. Imagine that each value is a combination of colors and the dimensions are the colors. The values that are average are made of equal quantities of colors, so they have a dirty green-brown color, but the values from the ends are made of bright red and of bright blue. Which one will you notice? Which one will determine the system? The middle section made of all kind of colors, or the ends that have just a few colors? Of course, that the ends.
Just like this example, the values that have the biggest average square deviation define our system.
So now lets get back to our original problem: we have a lot of cells, each of them containing 10,000 types of genes. By incorporating this information in our PCA solving technique, the information will be placed in a 200 directions space, each direction corresponding to a principal component.
As we acknowledged, some of the axis give us just a little information about the system, while others give us more than 50% of the information we need. So we can reduce the system to a 2 or 3 dimensional one.
By this point we have learned how to reduce the number of variables to 2 or 3. But what about the numbers of genes that we have to plot? Do we need to plot 10,000 points on a graph?
The answer is no.
Let's consider PC1 the biggest principal component and PC2 the second biggest principal component.

After we determine all the principal components, we can assume that:

Original System

Gene	Cell1	Cell2
a	10	8
b	0	2
c	14	10
d	33	45
e	50	42
f	80	72
g	95	90
h	44	50
i	60	50
etc	etc	etc

PC 1

Gene	Influence on PC1	In numbers
a	high	10
b	low	0.5
c	low	0.2
d	low	-0.2
e	high	13
f	high	-14
...	...	

PC2

Gene	Influence on PC2	In numbers
a	medium	3
b	high	10
c	high	8
d	high	-12
e	low	0.2
f	low	-0.1
...	...	

Cell1 PC1 score = (10 * 10) + (0 * 0.5) + ... etc... = 12

Figure 3: Genes influences

Each gene influences each principal component by a number. And each cell contains few or more information from each gene. We want to plot cells in the orthogonal system (PC1;PC2), so we can state that for example Cell1 will be equal to $x * PC1 + y * PC2$, where x is each gene present in cell 1 in original system multiplied by the same gene present in cell 1 in PC1. Similar for the PC2.
Finally we can plot 200 points on a 2 dimensional graph to observe each cell with its genes:

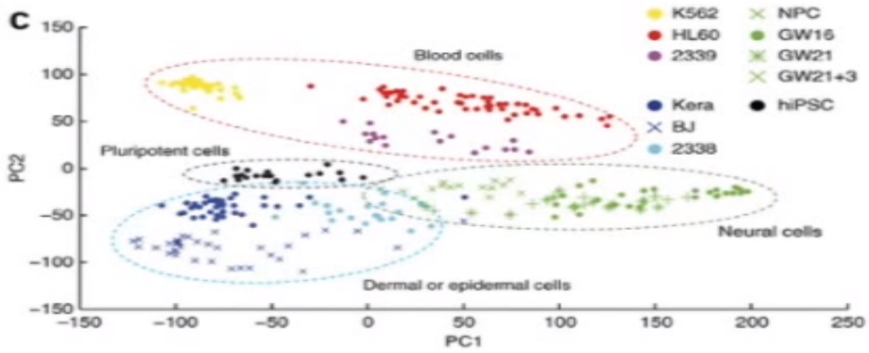

Figure 4: Cells plotted

In conclusion, we reduced our multidimensional problem to a smaller, understandable one using nothing more than PCA.

4 Mathematical approach

The first step in doing PCA is to find the vectors corresponding to each principal component.

4.1 How do we find the vectors?

We noticed that the direction of the biggest principal component is given by the way of variables are influencing each other. For example let's analyze the figure 1 from right to left, close to the second value there are 2 values(one on the right and one on the left). The one on the right is slightly higher than the second one, and the one on the left is slightly smaller. What does that tells us? It tells us this will always be in this order(from left to right small – average – big) no matter what average means. So they are insensitive to the mean value, actually their form will always be a positive linear form and that is very important. Having a linear form tells us that they vary in a linear way and they will always influence each other to respect this form.

4.2 How can we determine how they influence one another?

It is time we introduce a new term in statistic problem: the covariance matrix. The covariance matrix tells us if two variables do influence each other positively / negatively or they don't.

4.3 Example

We have the table below. There was an experiment that had 4 subjects and measured 4 qualities (random variables) of them.

	Var 1	Var 2	Var 3	Var 4
Subject 1	1	4	6	2
Subject 2	4	7	5	9
Subject 3	12	6	4	19
Subject 4	3	7	20	9

The formula for the covariance matrix is the following:

$$\Sigma_{ij} = cov(X_i, X_j) = E[(X_i - \mu_i) * (X_j - \mu_j)] = E[X_i, X_j] - \mu_i * \mu_j \quad (1)$$

So, to find out the covariance for Var 1 related to Var 2 we need to:

1. Find the mean of the first and second column
2. Subtract the first mean from the first column and the second mean from the second column
3. Multiply on each line the elements of Var 1 and Var 2
4. Add the results found after the multiplication.

23.3333	1.6667	-13.6667	32.6667
1.6667	2.0000	4.3333	4.6667
-13.6667	4.3333	56.9167	-9.4167
32.6667	4.6667	-9.4167	48.9167

Figure 5: Covariance Matrix

This is the result. We can see that the matrix is symmetric, so when we use it we should only use nearly half the space to store it. Another thing we can notice is the diagonal of the matrix. The diagonal represents the variance of the first variable, then the variance of the second one, etc. The variance of a discrete random variable measure the spread, or variability of the distribution.

4.4 Observation

Concidering the purpose of the data analysis, some formulas can be slightly changed:

1. How we calculate the covariance
   - Sample covariance
     It uses the algebraic mean applied just on a sample from the whole data
     Formula:

$$M = \frac{1}{n-1} * \sum_{i}^{n} (x_i - \bar{x}) * (y_i - \bar{y}) \quad (2)$$

- Population covariance
  It uses the algebraic mean applied on the whole data
  Formula:
$$M = \frac{1}{N} * \sum_{i}^{N} (x_i - \mu_x) * (y_i - \mu_y) \tag{3}$$

2. The covariance matrix vs. the correlation matrix

   - The covariance:
     - Gives us the way of the influence(positive/negative/0)
     - Does not give us the strength of the influence
     - Its values are spread in an interval defined by the original scale size
   - The correlation
     - Gives us the way of the influence(positive/negative/0)
     - Tells us the strength of the influence
     - Its values are spread in the inveraval (-1 ; 1)

The correlation matrix resembles the covariance matrix, but tells us more information.
The formula for this matrix is:

$$correlation(x_i, x_j) = \frac{cov(x_i, x_j)}{cov(x_i, x_i) * cov(x_j, x_j)} \tag{4}$$

The benefits for using the correlation matrix is that we know what its values mean. If we find the value 320, for example, on the covariance matrix we don't know if its strength is high or low, but if we find the value 0.9 on the correlation matrix we can be sure that its strength is high.

4.5 Rule of thumb for correlation strength

If the strength's absolute value is higher than the absolute value of r, than a relation is present between the elements of the correlation.

$$|r| >= \frac{2}{\sqrt{n}}, n = number\ of\ dimensions \tag{5}$$

4.6 So we have calculated the covariance matrix. But how can we use it?

Its importance is essential in finding the principal component, because we don't need anything else to find it. What we need is a random vector whose size is

equal to covariance column size. If we multiply the vector repeatedly by the covariance matrix we will see that the vector becomes bigger and rotates in the direction of the biggest principal component. Of course, the process is iterative and it will converge to the direction of the principal component.
So one way to find this direction is to take a random vector and to multiply it several times with the covariance matrix.
One clever way to find it is to take a vector that multiplied with the covariance it does not change its original direction. These vectors are called eigenvectors and they are representative for every single matrix.

4.7 How to find eigenvectors?

To find them, we must first find the eigenvalues associated with them. If the eigenvectors are some unit vectors placed on one direction, then the eigenvalues are their length.

$$\det(\lambda * I - \Sigma) = 0 \tag{6}$$

That formula gives us the eigenvalues (λ), where sigma is our covariance matrix
The eigenvectors are found solving the equation :

$$\Sigma * v = \lambda * v, where\ v = eigenvectors \tag{7}$$

4.8 Is there any easier way to find the eigenvectors? Do we really need to find the covariance matrix?

In 1970 Gene Golub published an algorithm that is still the one most-used even today. That algorithm is named SVD (Singular Value Decomposition). Basically what it does, it replaces one matrix by a product of three matrix: 2 are orthogonal and 1 is diagonal using the Householders transformation. The diagonal matrix contains the eigenvalues and the other 2 matrix contains the left and right eigenvectors.

$$A = U * \Sigma * V', \Sigma = diagonal \tag{8}$$

BUT, the covariance matrix is defined by:

$$\Sigma_{ij} = cov(X_i, X_j) = E[(X_i - \mu_i) * (X_j - \mu_j)] = E[X_i, X_j] - \mu_i * \mu_j \tag{9}$$

If we subtract the mean from each column of the matrix, we resume our problem to a matrix product, more specifically column i * column j, or line I of matrix A' and column j of matrix A

$$cov(A) = (A * A')/N \tag{10}$$

From SVD it results:

$$A = U*\Sigma*V' => A*A' = U*\Sigma*\Sigma'*U' = U*\Lambda*U'; \Lambda = eigenvalues\ of\ A*A' \tag{11}$$

U=eigenvectors of A*A',but at the same time the left eigenvectors of A => we do not need to compute the covariance matrix.

Why don't we use this method? Because when we find the left eigenvectors (U matrix) for the original matrix, we simultaneously compute the right eigenvectors too, so we are wasting time and memory on this approach.

4.9 Best solution

The best solution for this problem is a combination between the two possible ways of doing it.

1. Calculate the meaning of the matrix
2. Subtract the meaning from the matrix
3. Compute the covariance matrix
4. Find the eigenvectors of the covariance matrix
5. Eliminate some of them to reduce time and memory

5 Algorithm complexity

The most time consuming operation in this whole algorithm is to calculate the eigenvalues/eigenvectors. Let's see the complexity of each operation and let's calculate their sum:

- Compute the mean: O(N*M), where N is the number of pixels for each image and M is the number of images
- Subtract the mean: O(N*M)
- Compute the covariance matrix: $< O(N^3)$, because the matrix is symmetric
- Find the eigenvectors: $O(N^3)$, where N is the size of the covariance matrix

So the final result is $O(N^3)$.

So now we know the principal directions on which the information is spread and we can reduce the numbers of principal components by eliminating those that are the smallest (meaning those with the smallest eigenvalues). By doing this we reduce our problem and we keep just the essential information. To better understand what is happening we are going to see an example what is very intuitive.

6 Eigenfaces

6.1 Database

We have a database with 20 images of faces in it, each image having the size 300x300 pixels. Each pixel is a variable that influences every single image. So we have 20 eigenvectors with 90,000 elements, but we need to keep just the eigenvectors that provide us useful information, not all of them.
Image$\in R^{M\times N}$ => a column of M*N variables

Figure 6: Database of faces

Imagine that this is your database. One column of matrix A represents the pixels from one image placed one under another by column: on the top is the first column of pixels, under it is the second one and so on.
So we will have a matrix that is 90,000x20. For this matrix A we can compute the covariance matrix or the SVD for A'*A (because is smaller than A*A') and we will find the eigenvectors. From that list of eigenvectors we will keep just the eigenvectors that have the biggest eigenvalues(we keep enough of them to have the information we need, but not the redundant ones).

$$L = A' * A \ (L = covariance\ matrix;\ dimension : 20x20) \tag{12}$$

If we calculate the eigenvectors we will get 20 eigenvectors with 20 elements each. But we need 20 eigenvectors with the dimension 90,000.
If we compute L=A*A' instead of L=A'*A, we will get 90,000 eigenvectors with 90,000 elements each and from this 90,000 eigenvectors we need to identify the most important ones.

By calculating the covariance matrix as a small one we get the exact most important 20 eigenvectors, but with a small dimension. All we need to do now is to multiply the U matrix by the A matrix to get the length of eigenvectors 90,000.

$$U = A * U \tag{13}$$

6.2 Difference between A*A' and A'*A

We previously said that the complexity of finding the eigenvectors is $O(n^3)$. So, how that influences our runtime?
Let's try to approximate the running time for each version:
Our average computer's speed is 2.6 GHz. So that means it can make 2,600,000,000 operations per second.
If the covariance matrix has the dimensions $90,000 \times 90,000$, number of operations would be $90,000^3 = 729,000,000,000,000$ operations. This would take 280,384 seconds.
But if the covariance matrix is 20×20, then the number of operations would be $20^3 = 8000$, This would take 10^{-11} seconds.
The rule for computing is:

$$\begin{gathered} Size(A) = N * M; \\ \text{If}(N <= M) \\ Size(L) = N * N; \\ \text{else} \\ Size(L) = M * M; \end{gathered}$$

If we don't respect this simple rule we could waste entire minutes for nothing.

6.3 Eigenfaces obtained by PCA

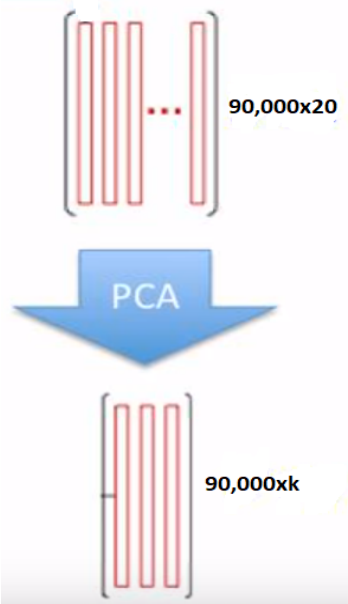

Figure 7: Computing the eigenfaces

After the transformation we will get something like this:

Figure 8: Eigenfaces

So each eigenvector represents a disturbed face like the ones above. They are just a combination of the previous faces. I like this example because we can actually see what PCA does to a certain set of values. Now imagine that this happens also when we work with every kind of problem and random variable, but we will not be able to "see" the result. It is just a combination of the initial data.
To better understand the images above, I will give you what I can deduce from them: For example from the second row – first column picture I can tell that the persons have a very similar nose and moustache, but they have different chins and eyes/glasses.

With a linear combination of these eigenvectors we can reproduce every single initial face, but if we keep just the eigenfaces with the biggest eigenvalues, we can recreate a certain percent of them. The percent is direct proportional with the eigenvalues that we decide to keep.

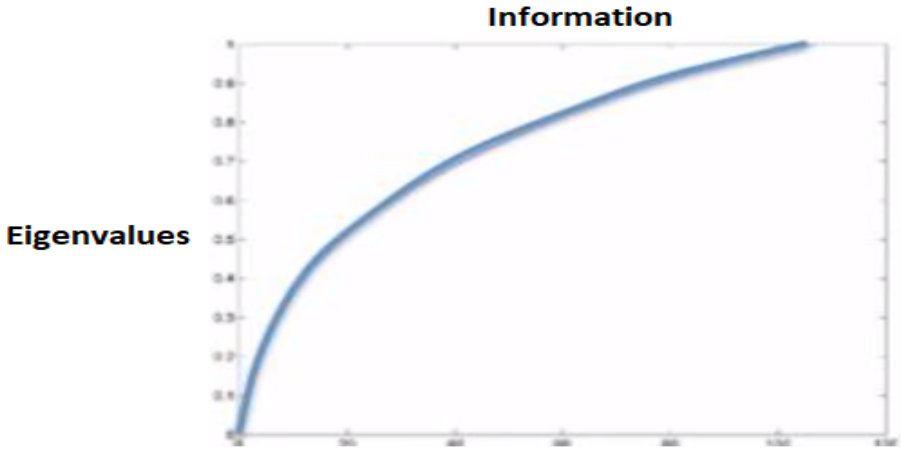

Figure 9: Eigenvalues - Percent of information kept

Below it is a face that is reconstructed from just a few eigenfaces to the majority of them. We can see that the more eigenfaces we add, the clearer is the final face.

Figure 10: Face recreation

6.4 Projection space

We had created our 20 eigenvectors U each with 90,000 elements and we said that the original images are a linear combination of these eigenfaces.
Now we need to compute the distance from each original image to each eigenface. To do that we multiply matrix U' by matrix A.

$$Projections = U' * A \tag{14}$$

Imagine we had a vector [3;5]' and the orthogonal directions [1;1]' and [1;-1]'. [3;5]' is equal to

$$\begin{aligned} dot([3;5]', [1;1]') * [1;1]' + dot([3;5]', [-1;1]') * [-1;1]' = \\ = [3;5]' * norm([1;1]') * norm([-1;1]') \end{aligned} \tag{15}$$

While the norm 2 of each direction is 1, the result is equal to the initial vector. Using this concept, we can calculate these coefficients by $U' * A$, because the coefficients are scalar products between each direction and the image.

6.5 Finding the most alike image

We take one new image that we want to compare and we store it in a 1x90,000 vector named "newim" and we find the coefficients from each linear combination between "newim" and all eigenfaces:

$$NewProjection = U' * newim \tag{16}$$

After that, we subtract these coefficients from the previous ones that we have found and we calculate the smallest distance, corresponding to the most alike image from the database.

$$min(norm(Projection_i - NewProjection)), i = \overline{1, M} \tag{17}$$

6.6 Algorithm deficiencies:

This algorithm is very sensitive to any light influence and also to facial expressions and micro-expressions.
To work properly, the database images have to be taken in identical conditions. A good example for what identical conditions mean are the mugshots done by police, where the suspect can't smile or grin and the background is always the same.
An innovative method for reducing these factors is Robust PCA which eliminates their contributions to the images.

7 Matlab Code

The following code is used in a facial recognition application to:

7.1 Find the principal components

```
%3 dimenisional matrix allocation in which is stored each image from the
%database
    imagini=zeros(H,W,M);
    enter = waitforbuttonpress;
    disp(' ');
    disp('II. Matrix allocation');
%Matrix allocation that have on each column an entire image, its numbers of
%columns being equal to the number of images
    vec=zeros(H*W,M);

% Pun informatia din imaginile din "baza de date" in matricile create
% anterior
%The information from the database is put into the matrices
    for i=1:M
        imagini(:,:,i)=imread(pgm(i).name);
        vec(:,i)=reshape(imagini(:,:,i),H*W,1);
    end
        enter = waitforbuttonpress;
        disp(' ');
        disp('III. PCA');
        disp('Calculate the arithmetic mean of each (x,y) pixel from all images');
% Calculate the arithmetic mean of each (x,y) pixel from all images
    m=sum(vec,2)/M;
enter = waitforbuttonpress;
disp('The previous results are substracted from the pixels');
% The previous results are substracted from the pixels
    A=vec-repmat(m,1,M);

    enter = waitforbuttonpress;
disp('Covariance matrix: ');
%Calculate the covariance matrix for A.
    L=A'*A;
    disp(L);
    enter = waitforbuttonpress;
disp('Eigenvectors of covariance matrix: ');
%Calculate the eigenvectors for the covariance matrix
    [V,lambda]=eig(L);
    disp(V);
    enter = waitforbuttonpress;
disp('Calculate Eigenfaces ');
disp('*Having a dimension of 300x300 it is hard to plot the results');
% Calculate Eigenfaces
    U=A*V;
```

Figure 11: PCA

7.2 Find the most alike image

```
%Testing image
    testIm=imread(testImage);

%Store the image in a lx(H*W) vector
    newim=reshape(testIm,H*W,1);

%Converting to double
    imtest=double(newim);

%Subtract the mean from the new image
    imd=imtest-m;
    enter = waitforbuttonpress;
disp('IV. Projection of the new image on the eigenfaces space: ');
% Projection of the new image on the eigenfaces space.
    NewProjection=U'*imd;
    disp(NewProjection);

%Dublicate vector NewProjection M times and subtract from it the previous
%distances of the database images
    d=repmat(NewProjection,1,M)-Projection;
%Distances vector allocation"
    dist=zeros(M,1);

    for i=1:M
        dist(i,1)=norm(d(:,i));
    end

% Fin the index of the miminum and maximum element
    index=IndiceMin(dist);
    index2=IndiceMax(dist);

%Norm all distances to 1.
    dist(:)=dist(:)/dist(index2);

%Display the smallest distance.
disp(' ');
    disp('The distance is:');
    disp(dist(index));

% Display the image.
    figure
    imshow(uint8(imagini(:,:,index)))
    title('Recognised face')
end
```

Figure 12: Minimum distance

With just some more code lines with this code you can create a facial recognition application, the above code being the core of the whole app.

8 Conclusions

Principal components analysis is a mathematical technique for reducing the numbers of dimensions of a problem. This technique has many applications in various fields and it simplifies the problem to a point that they can be understandable for us to optimize or solve them.
Using just the matrix properties and some of the key elements of numerical methods like eigenvalues, Householders or covariance, this method is by far one of the best ones in optimization and statistical problems.

References

[1] Bogdan Dumitrescu,Corneliu Popeea, Boris Jora, *"Metode de calcul numeric matriceal. Algoritmi fundamentali"*

[2] Graham Currell, *"Scientific Data Analysis"*

[3] PCA explained
`https://statquest.org/2015/08/13/pca-clearly-explained/`

[4] Victor Lavrenko
Online course, `http://bit.ly/PCA-alg`

[5] CompX: Computational Thinking and Big Data
`http://bit.ly/2rfZXSz`

[6] Brandon Foltz
Statistics 101, `https://www.youtube.com/user/BCFoltz/videos?flow=list&view=1&live_view=500&sort=dd`