MPr801 Project
Report

Face Recognition using Python

Submitted by
Aron G.
M013704

Under the guidance of
Dr. Ajit Kumar
Institute of Chemical Technology, Mumbai

Centre for Excellence in Basic Sciences
University of Mumbai
Kalina, Mumbai, India – 400098

Summer Semester 2017

Acknowledgement

I would like to express my profound sense of gratitude to Dr. Ajit Kumar for providing me this opportunity to learn under his invaluable guidance. I would also like to thank Dr. Sujit Tandel for providing me with computational assistance and Dr. Balwant Singh for the arrangements made for the project. I would also like to thank my classmates Maradana Sivakumar and Vijay Kumar Sharma for their prolonged support. I would like to thank UM-DAE Centre for Excellence in Basic Sciences for providing me this great opportunity. I also acknowledge the support provided by the Academic Board of the institute.

Abstract

Face recognition is one of the most challenging and fascinating pattern recognition problems. It's a constantly evolving field where new and efficient methods of recognition are discovered every now and then. Two of the most popular methods, namely eigenfaces and Fischerfaces are explored, studied, compared and contrasted as part of this project. Necessary programs for the same are written in python and are tested using webcam.

Contents

1 Introduction

Our aim is to use two of the most popular pattern recognition methods to recognize an unknown face from an image after training the computer with some known faces. For this, the computer should first of all be able to detect the faces in an image, and let us see how it is done.

1.1 Detecting the faces

Before going into reading faces directly, let us have a look at how a computer understands images.

1.1.1 Visualizing images as matrices

The images are nothing but figures with $N \times M$ pixels. Hence, they can be visualized as $N \times M$ dimensional matrices, where each element takes a hexadecimal value corresponding to the colour intensity on the respective cell in the image (*Figure 1*).

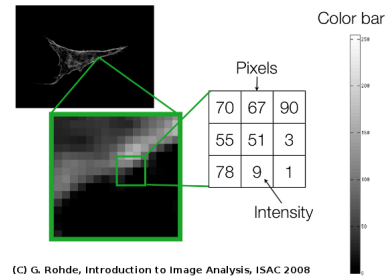

Figure 1: Image as a matrix.

Now that we know how to read the images, our next aim is to detect faces. This can be achieved by using Haar-cascade classifier or eigenfaces. The eigenfaces method will be explained in section 3.3.1. For now, let us look at how Haar-cascade classifiers work.

1.1.2 Haar-cascade classifiers

Initially, the algorithm needs a lot of positive images (images with faces) and negative images (images without faces) to train the classifier. Then we need to extract features from it. For this, haar features shown in *Figure 2* are used. Each feature is a single value obtained by subtracting sum of pixels under white rectangle from sum of pixels under black rectangle.

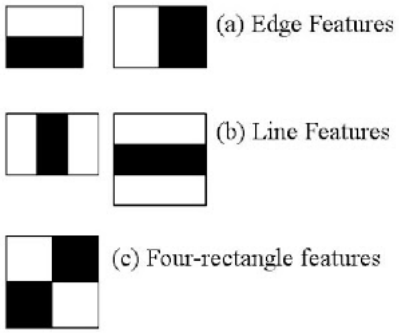

Figure 2: The five main Haar features [1].

For each feature calculation, we need to find sum of pixels under white and black rectangles. But among all these features we calculate, most of them are irrelevant. For example, consider the image below. Top row shows two good features. The first feature selected seems to focus on the property that the region of the eyes is often darker than the region of the nose and cheeks. The second feature selected relies on the property that the eyes are darker than the bridge of the nose. But the same windows applying on cheeks or any other place is irrelevant. We select the best features out of the total number of features by AdaBoost [1].

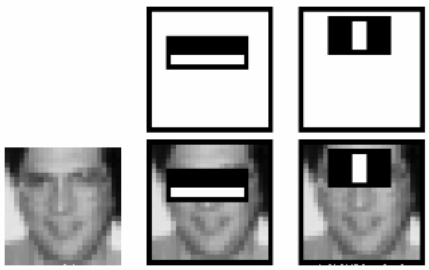

Figure 3: Haar features scanning a face [1].

AdaBoost, short for "Adaptive Boosting", is a machine learning meta-algorithm formulated by Yoav Freund and Robert Schapire who won the Gödel Prize in 2003 for their work. It can be used in conjunction with many other types of learning algorithms to improve their performance. The output of the other learning algorithms ('weak learners') is combined into a weighted sum that represents the final output of the boosted classifier. AdaBoost is adaptive in the sense that subsequent weak learners are tweaked in favor of those instances misclassified by previous classifiers.

Final classifier is a weighted sum of these weak classifiers. It is called weak because it alone cannot classify the image, but together with others forms a strong classifier.
Now we take an image and for each 24x24 window, apply 6000 features to it. Check if it is face or not. There is a way to improve the efficiency of this long computation, which is explained below.
In an image, most of the image region is non-face region. So it is a better idea to have a simple method to check if a window is not a face region. If it is not, discard it in a single shot. Don't process it again. Instead focus on region where there can be a face. This way, we can find more time to check a possible face region.

For this we introduce the concept of Cascade of Classifiers. Instead of applying all the 6000 features on a window, group the features into different stages of classifiers and apply one-by-one. (Normally first few stages will contain very less number of features). If a window fails the first stage, discard it. We don't consider remaining features on it. If it passes, apply the second stage of features and continue the process. The window which passes all stages is a face region.

Now that we know how to read faces, we need a large database to train the computer.

1.2 The Database

The main characteristics of a good training database is the enormous collection of faces and the variance in them, that is, different facial expressions of the same person, so as to increase the precision of recognition. Various face image data sets were used to study and analyze the program, however, the program was written to capture faces through the webcam of the computer. The primarily used data set was the Yale Face Database.

The Yale Face Database

The Yale Face Database (Figure 4) contains 165 grayscale images in GIF format of 15 individuals. There are 11 images per subject, one per different facial expression or configuration: center-light, with and without glasses, happy, left-light, normal, right-light, sad, sleepy, surprised, and wink.

Now that we know how to detect faces as well, our next target is to recognize them, that is, categorize them into a class, where a class is nothing but a collection of images of the same person. But, it is not as easy as it may

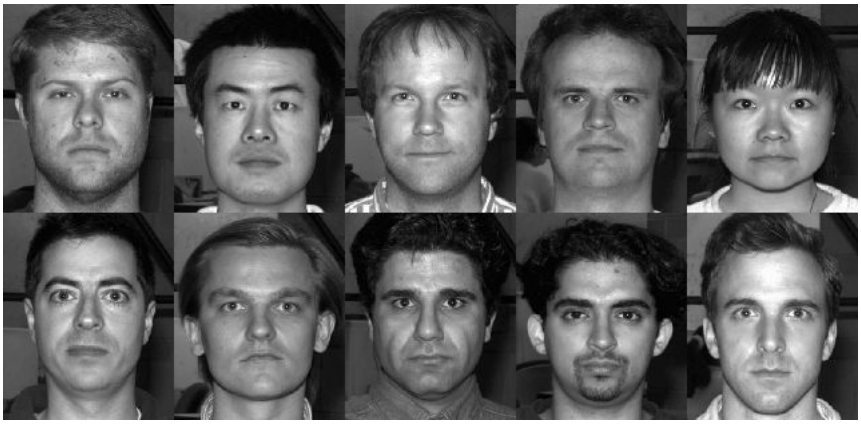

Figure 4: Sample faces from the Yale Face Database [3].

sound, since there are some pre-processing required because of the following problem.

Curse of dimensionality

Since the images are $N \times M$ dimensional, they become vectors of length NM. This which makes the data set very large and cumbersome to handle, since the matrices thus formed takes heavy chunks of RAM and processing power for computations. Hence we need to reduce the dimension of our input data without losing the key features of it. Thus comes to our rescue, image processing techniques like Principal Component Analysis (PCA) and Linear Discriminant Analysis (LDA).

2 Image processing techniques

These techniques ensure that the dimension of the images are subsequently reduced without much loss of any important features. The ones that we look into in this project are principal component analysis and linear discriminant analysis.

2.1 Principal Component Analysis

Principal component analysis (PCA) is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called

principal components (or sometimes, principal modes of variation) (Figure 5).

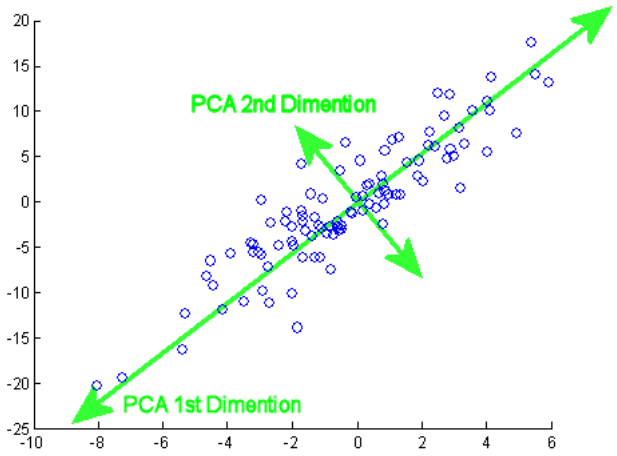

Figure 5: First two principal components of a sample data.

2.1.1 Features

The number of principal components is less than or equal to the smaller of the number of original variables or the number of observations. This transformation is defined in such a way that the first principal component has the largest possible variance (that is, accounts for as much of the variability in the data as possible), and each succeeding component in turn has the highest variance possible under the constraint that it is orthogonal to the preceding components. The resulting vectors are an uncorrelated orthogonal basis set. PCA is sensitive to the relative scaling of the original variables.

2.1.2 Procedure for PCA

PCA can be done by eigenvalue decomposition of a data co-variance (or correlation) matrix or singular value decomposition of a data matrix, and since the covariance method was used in the project only that is explained (Figure 6). The SVD method on the other hand, is not very different after obtaining the eigenvectors.

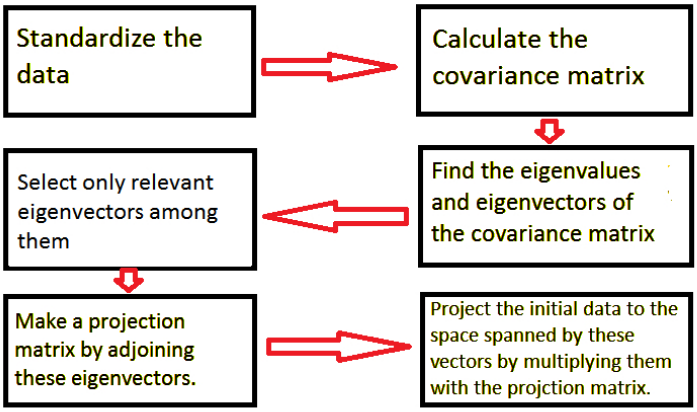

Figure 6: Basic outline of how PCA works .

2.1.3 Algorithm for PCA

Let $X_1, X_2, \ldots, X_M$ be the input data set and each of them has N features, that is, each X_i is of dimension N, where each component represents a feature.The mean of the data, Ψ is defined as

$$\Psi = \frac{1}{M} \sum_{i=1}^{M} X_i \,.$$

Now, update X_i's as

$$\tilde{X}_i = X_i - \Psi.$$

We need to compute the covariance matrix of A, which is defined as, $A = [\tilde{X_1}\tilde{X_2} \ldots \tilde{X_M}]$. The covariance matrix, C is defined as

$$C = \frac{1}{M} \sum_{i=1}^{M} \tilde{X_i} \tilde{X_i}^T = AA^T.$$

Now, we need to compute the eigenvectors u_i's of the covariance matrix, C. They are nothing but the values that satisfy the following relation

$$Cu_i = \lambda_i u_i,$$

where λ_i's are the eigenvalues of C.

The typical goal of a PCA is to reduce the dimensionality of the original feature space by projecting it onto a smaller subspace, where the eigenvectors will form the axes. In order to decide which eigenvector(s) can be

dropped without losing too much information for the construction of lower-dimensional subspace, we need to inspect the corresponding eigenvalues. The eigenvectors with the lowest eigenvalues bear the least information about the distribution of the data, those are the ones can be dropped. In order to do so, the common approach is to rank the eigenvalues from highest to lowest. But the question still remains on how do we decide the k, such that we have enough information to do the analysis, or in other words, how many principal components are we going to choose for our new feature subspace? To answer this, we look at the *explained variance*, which can be calculated from the eigenvalues. The explained variance tells us how much information (variance) can be attributed to each of the principal components (Figure 7). Explained variance of an eigenvalue λ_i is calculated as

$$EV(\lambda_i) = \frac{\lambda_i}{\sum_{j=1}^{N} \lambda_j} \times 100\,.$$

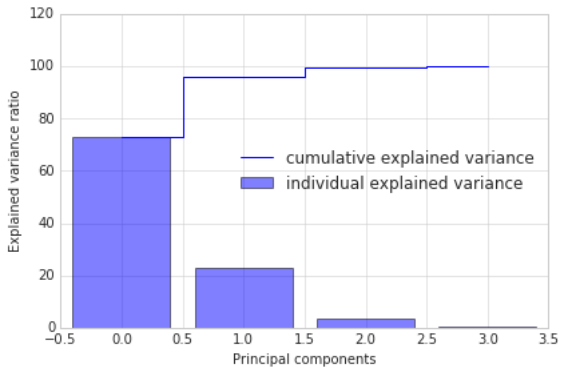

Figure 7: Explained variance of a sample data

Once the k eigenvalues are picked, their corresponding eigenvectors, u_i's are taken, and the data is projected into the feature space spanned by these u_i's. For this we define a projection matrix, W, which is obtained by concatenating the selected k eigenvectors, which is nothing but placing the k eigenvectors next to each other, to form an $N \times k$ matrix,

$$W = [u_1 u_2 \dots u_k]_{N \times k}\,.$$

Now, project the data vectors, by multiplying them by W, on the right, that is,

$$Y_i = X_i W,$$

where Y_i is the projection of X_i. Observe that Y_i's are k dimensional vectors.

2.2 Linear Discriminant Analysis

Another important image processing technique, where the inter class separation and within class separation are given more significance (Figure 8).

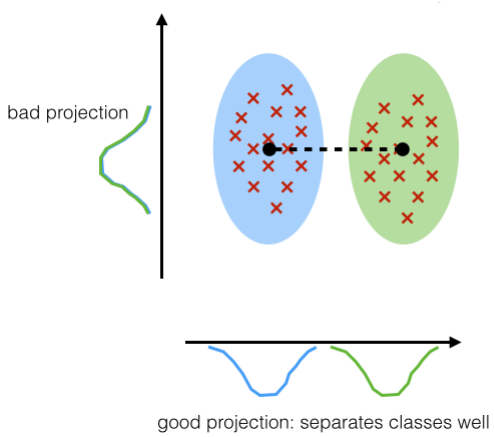

Figure 8: LDA optimizing the within class separation.

2.2.1 Procedure for LDA

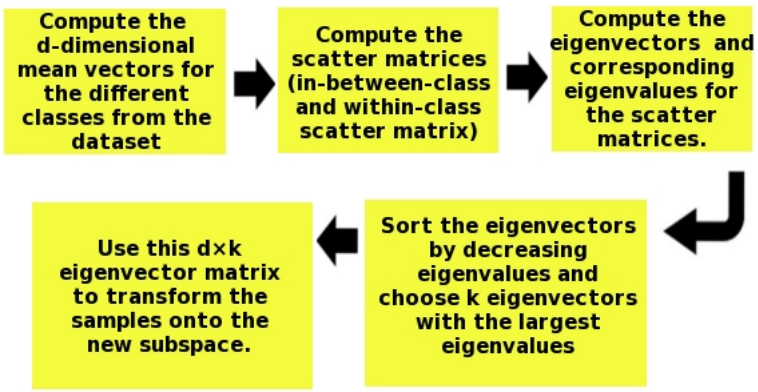

Figure 9: How LDA works.

Linear Discriminant Analysis (LDA) is most commonly used as dimensionality reduction technique in the pre-processing step for pattern-classification

and machine learning applications. The goal is to project a dataset onto a lower-dimensional space with good class-separability in order avoid overfitting and also reduce computational costs. Figure 9 shows how LDA works.

2.2.2 Algorithm for LDA

We are given with a certain amount of samples belonging to different classes. Let C be the collection of all classes and let N_i be the number of samples in the i^{th} class .Each sample has M features, hence they can be visualized as an M dimensional vector and therefore can be written as $\vec{X} = \{x_i\}_{1 \leq i \leq M}$. Then, we compute the average of all samples.

$$\vec{m} = \frac{1}{M} \sum_{i=1}^{M} \vec{X}_i.$$

Later, we compute the average of the samples in each class. Let the average of each class be denoted by $\tilde{C}_i$, that is,

$$\vec{\tilde{C}}_i = \frac{1}{N_i} \sum_{X_j \in C_i} \vec{X}_j.$$

Now, the difference of each image from the mean image of the class to which it belong and it is updated as follows,

$$\vec{X}_i = \vec{\tilde{C}}_i - \vec{X}_i.$$

2.2.3 Building Scatter Matrices

We build the scatter matrices for each class C_i , as follows

$$S_i = \sum_{X_j \in Ci} \vec{X}_j \vec{X}_j^{\,T}.$$

The within-class scatter matrix, S_W, is defined as

$$S_W = \sum_{i=1}^{N_i} S_i.$$

The between class scatter matrix, S_B, is defined as

$$S_B = \sum_{i=1}^{N_i} (\vec{\tilde{C}}_i - \vec{m}_i)(\vec{\tilde{C}}_i - \vec{m}_i)^T.$$

We should keep in mind that our aim is to minimize the within class scatter and maximize the between class scatter.
This is the same as maximizing the matrix $J(W)$, which is defined as follows

$$J(W) = \frac{W^T S_B W}{W^T S_W W},$$

where columns of W are eigenvectors of $S_W^{-1} S_B$.

Maximizing $J(W)$

Let $N_m = max\{N_i | i \in C\}$.
Case(i) **If S_W is non-singular, then $N_m \geq NM$.**
Compute the eigenvectors of $S_W^{-1} S_B$ and proceed.

Case(ii) **If S_W is singular, then $N_m \lneq NM$.**
Apply PCA first and reduce the dimension of faces from NM to N_m, thus making the training set, a N_m number of N_m dimensional vectors.
Now, compute the eigenvectors of $S_W^{-1} S_B$ and proceed.

Now that we are familiar with the image processing techniques, let us go ahead and see how to make use of them in the face recognition process.

3 Eigenfaces

Eigenfaces is the name given to a set of eigenvectors when they are used in the computer vision problem of human face recognition. The approach of using eigenfaces for recognition was developed by Sirovich and Kirby (1987) and used by Matthew Turk and Alex Pentland in face classification. The eigenvectors are derived from the covariance matrix of the probability distribution over the high-dimensional vector space of face images. The eigenfaces themselves form a basis set of all images used to construct the covariance matrix. This produces dimension reduction by allowing the smaller set of basis images to represent the original training images. Classification can be achieved by comparing how faces are represented by the basis set.

3.1 Procedure for eigenfaces

As mentioned earlier, the curse of dimensionality was haunting us in the operations in high dimensional space. So we use PCA to reduce the dimensionality of the data while retaining as much information as possible in the

original dataset.

Given an image database of faces, PCA is applied on it, from which a space spanned by the principal components is obtained. These principal components are nothing but the eigenvectors of the dominant eigenvalues of the correlation matrix, are hence called the eigenfaces and the space spanned by them is called the face space. The initial faces and unknown face is projected onto this space and the distance between them is measured using eucidean norm (Figure 10).

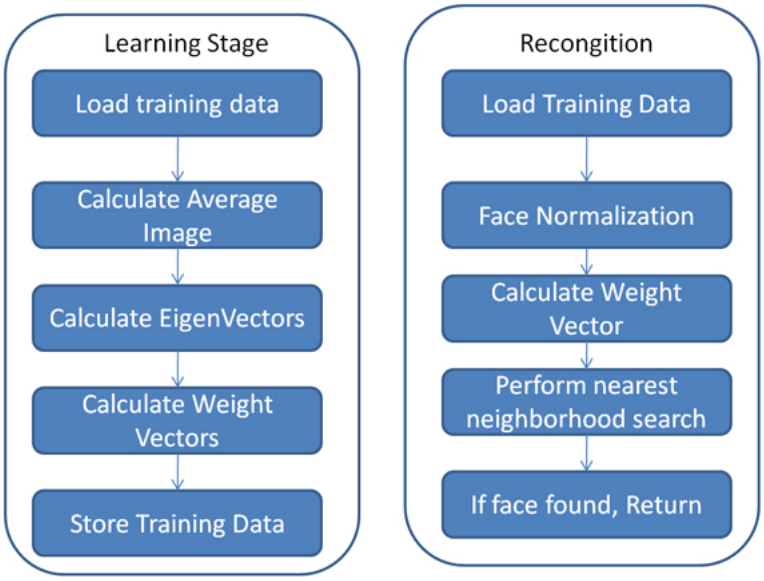

Figure 10: How eigenfaces work.

3.2 Algorithm for eigenfaces

Let $I_1, I_2, \ldots, I_M$ be the training face images. Each image has a resolution of $N \times N$ pixels and hence can be written as an $N \times N$ matrix. The entries of the matrix will be the intensity of the corresponding pixel in the image. Now, let Γ_i be an $N^2 \times 1$ vector obtained by vectorising the $N \times N$ image I_i by stacking the columns one above the other.
Compute the average face vector Ψ,

$$\Psi = \frac{1}{M} \sum_{i=1}^{M} \Gamma_i \, .$$

We will now normalize each image, for that we take their difference from the mean face as follows,

$$\Phi_i = \Gamma_i - \Psi \, .$$

After that, will compute the covariance matrix, C, which is given by,

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

where $A = [\Phi_1 \Phi_2 \dots \Phi_M]$. We need to compute the eigenvectors u_i of AA^T but since AA^T is very large, its not practical to compute its eigenvalues. But we know $AA^T and A^T A$ have the same eigenvalues (Theorem 1), and $A^T A$ is smaller in dimension than AA^T, so it's preferable to calculate the eigenvalues of $A^T A$.

Theorem 1. *Let A be an $n \times n$ dimensional matrix, then, AA^T and $A^T A$ have the same eigenvalues.*

Proof. Let $AA^T v = \lambda v$, where λ is an eigenvalue of $AA^T$and v, the corresponding eigenvector of AA^T. Multiplying A^T on both sides in the left yields

$$A^T AA^T v = A^T \lambda v.$$

Since λ is a scalar, this can be rearranged as

$$A^T A(A^T v) = \lambda(A^T v).$$

□

Note that λ is the eigenvalue for $A^T A$ as well but the eigenvector is different and as our aim is to find the eigenvectors of $A^T A$, and we do so using this λ. Given an eigenvalue λ_i of $A^T A$, we know,

$$A^T A v_i = \lambda_i v_i \, , \tag{1}$$

where $v_i' s$ are eigenvectors of $A^T A$ corresponding to eigenvalues $\lambda_i' s$ and

$$AA^T u_i = \lambda_i u_i, \tag{2}$$

where $u_i' s$ are eigenvectors of AA^T corresponding to eigenvalues $\lambda_i' s$. Now, we need to establish a relation between $u_i' s$ and $v_i' s$. Multiplying A on both sides of equation (1), we get,

$$AA^T A v_i = A \lambda_i v_i \, ,$$

where we know $AA^T = C$ and λ_i can commute, since it is a scalar. Hence,

$$CAv_i = \lambda_i Av_i.$$

Comparing this with equation (2), which is,

$$Cu_i = \lambda_i u_i \,,$$

we can conclude that $u_i = Av_i$.
Now that we have eigenvectors of AA^T, our next aim is to choose the relevant ones. For this, we arrange the eigenvalues of AA^T in descending order and choose K dominant ones among them and normalize $u_i's$ corresponding to those $\mu_i's$, such that, $\|u_i\| = 1$.
Now observe that these K $u_i's$ form an orthonormal basis for the space of the faces and are hence called the eigenfaces.

3.2.1 Representation of eigenfaces

Each face in the training set can be represented as a linear combination of the best K eigenvectors as follows,

$$\Gamma_i = \sum_{j=1}^{K} w_j u_j \,, \quad (w_j = u_j^T \Phi_i).$$

The weight vector w_i is unique for each image Γ_i and hence can be used as a representative for each of them.
For a given image Γ_i , the weight vector Ω_i is given by

$$\Omega_i = \begin{pmatrix} w_1^i \\ w_2^i \\ \vdots \\ w_K^i \end{pmatrix}.$$

Now that we have completed the task of creating a "face-space", it is time to recognise an unknown image by projecting it onto this space.

3.3 Face Recognition Using Eigenfaces

Given an unknown image Γ of someone's face, we need to exploit the above method (3.2.1) to recognise their face. For that, all we need to do is express the image as a linear combination of the eigenfaces, and then compare the coefficients with that of the vector. Firstly, we normalize the unknown image, Γ as follows,

$$\Phi = \Gamma - \Psi.$$

Project it onto the eigenspace as

$$\hat{\Phi} = \sum_{i=1}^{K} w_i u_i, (w_i = u_i^T \Phi),$$

where $\hat{\Phi}$ is the projection of Φ and w_i's are the weight vectors. Hence, Φ can now be represented as

$$\Omega = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_K \end{pmatrix}.$$

This Ω is now compared with all the representatives Ω_i's of the images in the set of training face images. Let us take the nearest neighbour distance to compute the difference between them, that is, define ϵ as,

$$\epsilon = min_i \left\| \Omega - \Omega_i \right\|.$$

If ϵ is less than some cutoff value, say t, for some Ω_i, then the unknown face is said to be Γ_i.

3.3.1 Face Detection Using Eigenfaces

As mentioned before, face detection can also be done using eigenfaces. Given an image Γ, our aim is see if it is a face image or not. We start with normalizing Γ as,

$$\Phi = \Gamma - \Psi.$$

Now, it is projected on the eigenspace as

$$\hat{\Phi} = \sum_{i=1}^{K} w_i u_i, (w_i = u_i^T \Phi).$$

The difference of the projection of unknown image from the normalized image determines how close the unknown image is as of a face. Define an error term ϵ as before,

$$\epsilon = \left\| \hat{\Phi} - \Phi \right\|.$$

If ϵ is less than some cutoff value, say t, then the unknown image has a face in it.

4 Fischerfaces

The Fisherface method is an enhancement of the eigenface method that it uses Fisher's Linear Discriminant Analysis (FLDA or LDA) for the dimensionality reduction. The LDA maximizes the ratio of between-class scatter to that of within-class scatter, therefore, it works better than PCA for purpose of discrimination. The Fisherfaces method is especially useful when facial images have large variations in illumination and facial expression.

4.1 Procedure for Fischerfaces

Given a certain number of images of some people, our aim is to make a classifier with maximum separability between classes, that is, between images of two different people.

4.2 Algorithm for Fischerfaces

We are given with certain number of images of people, of dimension $N \times M$. Let the total number of images be L and the number of people be C. Note that number of people is same as number of classes, since each class represents images of one person.
Each image can be visualized as an NM dimensional vector. Hence can be written as $\vec{X} = \{x_i\}_{1 \leq i \leq NM}$.
Then, we compute the average of all faces as

$$\vec{m} = \frac{1}{L} \sum_{i=1}^{L} \vec{X_i}.$$

Later, we compute the average face of each person. And since each person represents each class, it's the same as taking the average of faces in a class. Since each class have equal number of faces, the number of classes, $N_c = \frac{M}{N_m}$, where N_m =Number of images of each person. And let Π_i denote each class. Therefore the average face of each person is nothing but,

$$\vec{C_i} = \sum_{X_j \in \Pi_i} \vec{X_j}.$$

Now, the difference of each image from the mean image of the class to which it belong and it is updated as follows,

$$\vec{X_i} = \vec{C_i} - \vec{X_i}.$$

4.2.1 Building Scatter Matrices

We build the scatter matrices for each class Π_i , as follows

$$S_i = \sum_{X_j \in \Pi_i} \vec{X_j} \vec{X_j}^T .$$

The within-class scatter matrix S_W is defined as

$$S_W = \sum_{i=1}^{N_c} S_i \, .$$

The between class scatter matrix is defined as

$$S_B = \sum_{i=1}^{N_c} (\vec{C_i} - \vec{m_i})(\vec{C_i} - \vec{m_i})^T \, .$$

We should keep in mind that our aim is to minimize the within class scatter and maximize the between class scatter.
This is the same as maximizing the matrix $J(W)$, which is defined as follows

$$J(W) = \frac{W^T S_B W}{W^T S_W W} \, ,$$

where columns of W are eigenvectors of $S_W^{-1} S_B$. Maximize $J(W)$, as shown in section 2.2.3.

4.3 Classification using Fischerfaces

The mean images of each person should be projected onto the LDA space as follows

$$\vec{X_{LDA}} = W^T \vec{X} \, .$$

Given an unknown image $\vec{Y}$, project it into the LDA space as follows

$$\vec{Y_{LDA}} = W^T \vec{Y} \, .$$

Run a nearest neighbour classifier over the projected mean images,

$$dist = \left\| \vec{X} - \vec{Y} \right\|_2 \, .$$

If $dist \leq \epsilon$, for some cut off value ϵ, then classify $\vec{Y}$ into $\vec{X}$'s class.

5 Results

The program was written to capture user's face using webcam, which is later processed and a classifier is made out of it. While training the user's face, the program cuts just the face in a square frame (Figure 11).

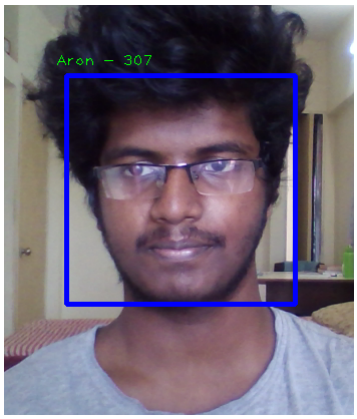

Figure 11: User's face being detected.

About 100 to 200 such photos with different angles and poses are taken in a row (Figure 12) as the test images and a classifier is generated out of it .

Figure 12: Diffrerent poses maximises the testing accuracy

5.1 Limitations of Eigenfaces

Eigenfaces being the first ever face recognition method is bound to have many limitations, they are,

- Change in background causes problems.
- Change in lighting conditions degrade performance.
- Performance decreases quickly with changes to face size.
- cannot accommodate changes in orientation.
- PCA assumes that the data follows a Gaussian distribution, which is not always true.

5.2 Fischerfaces

Fischerfaces on the other hand, were able to handle various lighting conditions, facial expression changes and even some props, including spectacles, though with limited or no variations in these parameters, eigenfaces performed better. Some of the advantages Fischerfaces have over eigenfaces are,

- Faster than eigenfaces, in some cases.
- Has lower error rates with changes in background.
- Works well even in different illumination conditions.
- Can accommodate changes in facial expression.

6 Conclusion

The task of recognizing faces was a success in overall, though the techniques used were not the most advanced or the latest. Eigenfaces, had a better success rate than Fischerfaces when images of the same lighting and background conditions were given with less changes in orientation, but with variations in all the mentioned parameters, Fischerfaces was clearly the winner among the two. This makes Fischerfaces a more dependable and stable technique for face recognition amongst the two.

7 Python Programs

The following program was written for eigenfaces, however a functional program for Fischerfaces could not be written due to lack of computational power and complexity in making an efficient python module. Due to this reason, a python-applet that uses Fischerfaces for face recognition was downloaded from GitHub and used for study.

7.1 Eigenfaces

```
# Program to recognize face using eigenfaces
from os import listdir
from PIL import Image as PImage
import numpy as np
import matplotlib.pyplot as plt

# Module to load images from a file
def loadImages(path):
    imagesList = listdir(path)
    loadedImages = []
    for image in imagesList:
        img = PImage.open(path + image)
        loadedImages.append(img)
    return loadedImages
path = "C:/Users/Aron/Documents/Sem8 project/Fischerfaces/facespca/"

# loading images into an array
imgs = loadImages(path)
images=[]
for img in imgs:
        imggray = img.convert('LA')
        imgmat = np.array(list(imggray.getdata(band=0)), float)
        images.append(imgmat)

# Normalizing the images against their mean.
m=0
for img in images:
        m=m+img
mean=m/len(images)
for img in images:
        img=img-mean
```

```
#Calculating the eigenvaluea and eigenvectors of the covariance matrix
Q=np.matmul(np.transpose(images),images)
W=np.linalg.eigh(Q)
WV=W[0]

# Selecting relevant eigenvectors using explained variance
flg=0.0
for i in WV:
        flg=flg+i
eigindex=[]
for i in range(len(WV)):
        flg1=(WV[i]/flg)*100
        if (flg1 > 3):
                eigindex.append(i)

#Constructing the projection matrix
promat=[]
for num in eigindex:
        promat.append(W[1][num])
promat=np.mat(promat)

# Uploading the image to be recognized
from PIL import Image
img = Image.open('recognize.png')
imggray = img.convert('LA')
imgmat = np.array(list(imggray.getdata(band=0)), float)

#projecting the unknown image into eigenspace
test=np.dot(promat,imgmat)

#measuring the nearest neighbour distance from other faces
erlist=[]
for img in images:
        er=np.linalg.norm(test,imgpro)
        erlist.append(er)

#Classifying the unknown face
for i in range(len(WV)):
        if (erlist[i]==numpy.amax(erlist)):
            print "The unknown face is the",i,"th image from the training set"
```

7.2 Fischerfaces

This is the code used in the python-applet which was downloaded from *https://github.com/bytefish/facerec*.

7.2.1 Training Fischerfaces

```
import numpy as np
import cv2
import sys
import os

FREQ_DIV = 5   #frequency divider for capturing training images
RESIZE_FACTOR = 4
NUM_TRAINING = 100

class TrainFisherFaces:
    def __init__(self):
        cascPath = "haarcascade_frontalface_default.xml"
        self.face_cascade = cv2.CascadeClassifier(cascPath)
        self.face_dir = 'face_data'
        self.face_name = sys.argv[1]
        self.path = os.path.join(self.face_dir, self.face_name)
        if not os.path.isdir(self.path):
            os.mkdir(self.path)
        self.model = cv2.createFisherFaceRecognizer()
        self.count_captures = 0
        self.count_timer = 0

    def capture_training_images(self):
        video_capture = cv2.VideoCapture("female.avi")
        while True:
            self.count_timer += 1
            ret, frame = video_capture.read()
            inImg = np.array(frame)
            outImg = self.process_image(inImg)
            cv2.imshow('Video', outImg)

            # When everything is done, release the capture on pressing 'q'
            if cv2.waitKey(1) & 0xFF == ord('q'):
                video_capture.release()
```

```
                cv2.destroyAllWindows()
                return

    def process_image(self, inImg):
        frame = cv2.flip(inImg,1)
        resized_width, resized_height = (112, 92)
        if self.count_captures < NUM_TRAINING:
            gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
            gray_resized = cv2.resize(gray, (gray.shape[1]/RESIZE_FACTOR, gray.s
            faces = self.face_cascade.detectMultiScale(
                gray_resized,
                scaleFactor=1.1,
                minNeighbors=5,
                minSize=(30, 30),
                flags=cv2.cv.CV_HAAR_SCALE_IMAGE
                )
            if len(faces) > 0:
                areas = []
                for (x, y, w, h) in faces:
                    areas.append(w*h)
                max_area, idx = max([(val,idx) for idx,val in enumerate(areas)])
                face_sel = faces[idx]

                x = face_sel[0] * RESIZE_FACTOR
                y = face_sel[1] * RESIZE_FACTOR
                w = face_sel[2] * RESIZE_FACTOR
                h = face_sel[3] * RESIZE_FACTOR

                face = gray[y:y+h, x:x+w]
                face_resized = cv2.resize(face, (resized_width, resized_height))
                img_no = sorted([int(fn[:fn.find('.')]) for fn in os.listdir(sel

                if self.count_timer%FREQ_DIV == 0:
                    cv2.imwrite('%s/%s.png' % (self.path, img_no), face_resized)
                    self.count_captures += 1
                    print "Captured image: ", self.count_captures

                cv2.rectangle(frame, (x, y), (x+w, y+h), (0, 255, 0), 3)
                cv2.putText(frame, self.face_name, (x - 10, y - 10), cv2.FONT_HE
        elif self.count_captures == NUM_TRAINING:
            print "Training data captured. Press 'q' to exit."
```

```
            self.count_captures += 1

        return frame

    def are_enough_faces(self):
        existingFaces = 0
        for (subdirs, dirs, files) in os.walk(self.face_dir):
            for subdir in dirs:
                existingFaces += 1

        if existingFaces > 1:
            return True
        else:
            return False

    def fisher_train_data(self):
        imgs = []
        tags = []
        index = 0

        for (subdirs, dirs, files) in os.walk(self.face_dir):
            for subdir in dirs:
                img_path = os.path.join(self.face_dir, subdir)
                for fn in os.listdir(img_path):
                    path = img_path + '/' + fn
                    tag = index
                    imgs.append(cv2.imread(path, 0))
                    tags.append(int(tag))
                index += 1
        (imgs, tags) = [np.array(item) for item in [imgs, tags]]

        self.model.train(imgs, tags)
        self.model.save('fisher_trained_data.xml')
        print "Training completed successfully"
        return


if __name__ == '__main__':
    trainer = TrainFisherFaces()
    trainer.capture_training_images()
```

```
    if trainer.are_enough_faces():
        trainer.fisher_train_data()
        print "Type in next user to train, or press Recognize"
    else:
        print "Type in next user to train, or press Recognize"
```

7.2.2 Recognition using Fischerfaces

```
import numpy as np
import cv2
import sys
import os

RESIZE_FACTOR = 4

class RecogFisherFaces:
    def __init__(self):
        cascPath = "haarcascade_frontalface_default.xml"
        self.face_cascade = cv2.CascadeClassifier(cascPath)
        self.face_dir = 'face_data'
        self.model = cv2.createFisherFaceRecognizer()
        self.face_names = []

    def load_trained_data(self):
        names = {}
        key = 0
        for (subdirs, dirs, files) in os.walk(self.face_dir):
            for subdir in dirs:
                names[key] = subdir
                key += 1
        self.names = names
        self.model.load('fisher_trained_data.xml')
    def show_video(self):
        video_capture = cv2.VideoCapture("Untitle.avi")
        while True:
            ret, frame = video_capture.read()
            inImg = np.array(frame)
            outImg, self.face_names = self.process_image(inImg)
            cv2.imshow('Video', outImg)
            # When everything is done, release the capture on pressing 'q'
            if cv2.waitKey(1) & 0xFF == ord('q'):
```

```
                video_capture.release()
                cv2.destroyAllWindows()
                return
    def process_image(self, inImg):
        frame = cv2.flip(inImg,1)
        resized_width, resized_height = (112, 92)
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        gray_resized = cv2.resize(gray, (gray.shape[1]/RESIZE_FACTOR, gray.shape
        faces = self.face_cascade.detectMultiScale(
                gray_resized,
                scaleFactor=1.1,
                minNeighbors=5,
                minSize=(30, 30),
                flags=cv2.cv.CV_HAAR_SCALE_IMAGE
                )
        persons = []
        for i in range(len(faces)):
            face_i = faces[i]
            x = face_i[0] * RESIZE_FACTOR
            y = face_i[1] * RESIZE_FACTOR
            w = face_i[2] * RESIZE_FACTOR
            h = face_i[3] * RESIZE_FACTOR
            face = gray[y:y+h, x:x+w]
            face_resized = cv2.resize(face, (resized_width, resized_height))
            confidence = self.model.predict(face_resized)
            if confidence[1]<400:
                person = self.names[confidence[0]]
                cv2.rectangle(frame, (x,y), (x+w, y+h), (255, 0, 0), 3)
                cv2.putText(frame, '%s - %.0f' % (person, confidence[1]), (x-10,
            else:
                person = 'Unknown'
                cv2.rectangle(frame, (x,y), (x+w, y+h), (0, 0, 255), 3)
                cv2.putText(frame, person, (x-10, y-10), cv2.FONT_HERSHEY_PLAIN,
            persons.append(person)
        return (frame, persons)
if __name__ == '__main__':
    recognizer = RecogFisherFaces()
    recognizer.load_trained_data()
    print "Press 'q' to quit video"
    recognizer.show_video()
```

References

[1] P. Viola, M. Jones *Rapid object detection using a boosted cascade of simple features*. IEEE Conference on Computer Vision and Pattern Recognition, CVPR, Hawaii, 2001.

[2] http://cvc.yale.edu/projects/yalefaces/yalefaces.html *The Yale Face Database*

[3] M. A. Turk and A. P. Pentland *Face Recognition Using Eigenfaces*. Proc. of IEEE Conf. on Computer Vision and Pattern Recognition, pp. 586-591, June 1991.

[4] J. Ashbourn, Avanti, V. Bruce, A. Young *Face Recognition Based on Symmetrization and Eigenfaces*.

[5] Muller, N., Magaia, L. and Herbst B.M. *Singular value decomposition, eigenfaces, and 3D reconstructions*. SIAM Review, Vol. 46 Issue 3, pp. 518–545. Dec. 2004.

[6] Belhumeur, J.Hespanha, D.Kriegman *Eigenfaces vs. Fisherfaces: Recognition Using Class Specific Linear Projection*. IEEE PAMI, 1997.

[7] M.H.Yang, D.J.Kriegman, and N.Ahuja *Detecting faces in images: A survey* EEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI), Vol. 24, No.1,pp.34–58, 2002.

[8] Ö. Toygar, A. Acan *Face Recognition Using PCA, LDA and ICA, Approaches On Colored Images*. Journal Of Electrical & Electronics Engineering, Vol.3, No.1, pp.735-743, Turkey, 2003.