FACE RECOGNITION USING OPENCV

**1. THEORY OF OPENCV FACE RECOGNIZERS**

Thanks to OpenCV, coding facial recognition is now easier than ever. There are three easy steps to computer coding facial recognition, which are similar to the steps that our brains use for recognizing faces. These steps are:

1. **Data Gathering:** Gather face data (face images in this case) of the persons you want to identify.
2. **Train the Recognizer:** Feed that face data and respective names of each face to the recognizer so that it can learn.
3. **Recognition:** Feed new faces of that people and see if the face recognizer you just trained recognizes them.



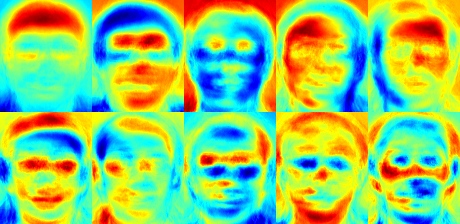
OpenCV has three built-in face recognizers and thanks to its clean coding, you can use any of them just by changing a single line of code. Here are the names of those face recognizers and their OpenCV calls:

* **EigenFaces** – cv2.face.createEigenFaceRecognizer()
* **FisherFaces** – cv2.face.createFisherFaceRecognizer()
* **Local Binary Patterns Histograms (LBPH)** – cv2.face.createLBPHFaceRecognizer()

### 1.1 EIGENFACES FACE RECOGNIZER

This algorithm considers the fact that **not all parts of a face are equally important or useful for face recognition**. Indeed, when you look at someone, you recognize that person by his distinct features, like the eyes, nose, cheeks or forehead; and how they vary respect to each other.

For example, from the eyes to the nose there is a significant change, and same applies from the nose to the mouth. When you look at multiple faces, you compare them by looking at these areas, because by catching the maximum variation among faces, they help you differentiate one face from the other.



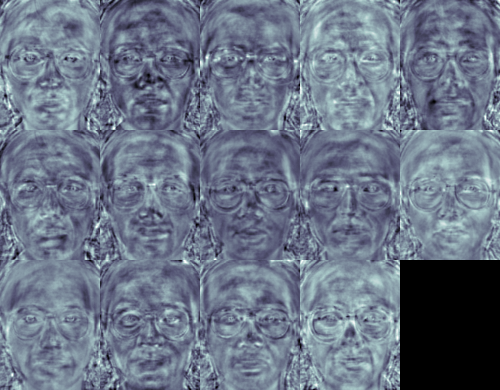
EigenFaces Face Recognizer Principal Components.

Source: [**docs.opencv.org**](http://docs.opencv.org/2.4/modules/contrib/doc/facerec/facerec_tutorial.html)

However, one thing to note in above image is that **EigenFaces algorithm also considers illumination as an important feature**. In consequence, lights and shadows are picked up by EigenFaces, which classifies them as representing a ‘face.'

### 1.2 FISHERFACES FACE RECOGNIZER

This algorithm is an improved version of the last one. As we just saw, **EigenFaces** looks at all the training faces of all the people at once and finds principal components from all of them combined. By doing that, it doesn't focus on the features that discriminate one individual from another. Instead, it concentrates on the ones that represent all the faces of all the people in the training data, *as a whole*.



* Since EigenFaces also finds illumination as a useful component, it will find this variation very relevant for face recognition and may discard the features of the other people's faces, considering them less useful. In the end, the variance that EigenFaces has extracted represents *just one individual's facial features*.

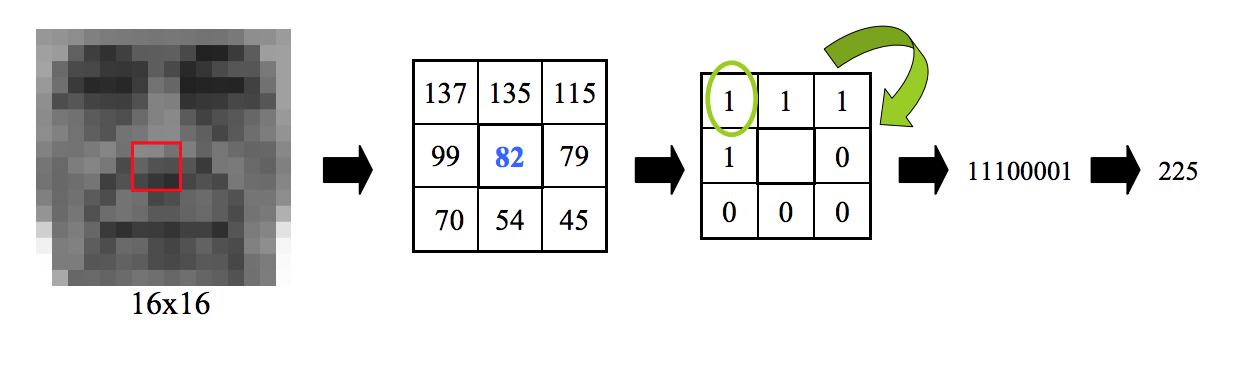
### 1.3 LBPH FACE RECOGNIZER

The idea with **LBPH** is not to look at the image as a whole, but instead, try to find its local structure by comparing each pixel to the neighboring pixels.

#### THE LBPH FACE RECOGNIZER PROCESS

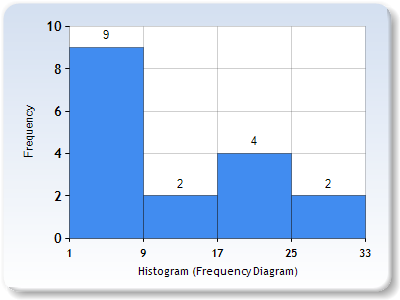
Take a 3×3 window and move it across one image. At each move (each local part of the picture), compare the pixel at the center, with its surrounding pixels. Denote the neighbors with intensity value less than or equal to the center pixel by 1 and the rest by 0.

After you read these 0/1 values under the 3×3 window in a clockwise order, you will have a binary pattern like 11100011 that is local to a particular area of the picture. When you finish doing this on the whole image, you will have a list of **local binary patterns**.



LBP conversion to binary. Source: López & Ruiz; Local Binary Patterns applied to Face Detection and Recognition.

Now, after you get a list of local binary patterns, you convert each one into a decimal number using [**binary to decimal conversion**](https://www.mathsisfun.com/binary-number-system.html) (as shown in above image) and then you make a [**histogram**](https://www.mathsisfun.com/data/histograms.html) of all of those decimal values. A sample histogram looks like this:



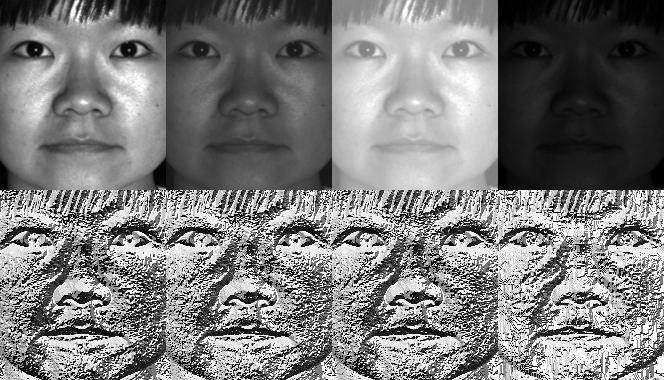
Histogram Sample.

In the end, you will have one histogram for each face in the training data set. That means that if there were 100 images in the training data set then LBPH will extract 100 histograms after training and store them for later recognition. Remember, the **algorithm also keeps track of which histogram belongs to which person**.

Later during recognition, the process is as follows:

1. Feed a new image to the recognizer for face recognition.
2. The recognizer generates a histogram for that new picture.
3. It then compares that histogram with the histograms it already has.
4. Finally, it finds the best match and returns the person label associated with that best match.

Below is a group of faces and their respective local binary patterns images. You can see that the **LBP faces are not affected by changes in light conditions**:



LBPH Face Recognizer Principal Components. Source: [**docs.opencv.org**](http://docs.opencv.org/2.4/modules/contrib/doc/facerec/facerec_tutorial.html)

.

2. ALGORITHM/CODING

### ****2.1 PREPARE TRAINING DATA.****

The premise here is simple:

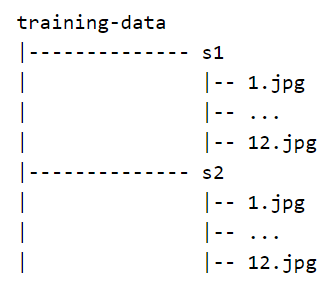
The more images used in training, the better.

Being thorough with this principle is important because it is the only way for training a face recognizer so it can learn the different ‘faces’ of the same person; for example: with glasses, without glasses, laughing, sad, happy, crying, with a beard, without a beard, etc.

So, our training data consists of total two people with 12 images of each one. All training data is inside the folder:*training-data.*

This folder contains one subfolder for every individual, named with the format:**sLabel (e.g. s1, s2)**where**the label is the integer assigned to that person**. For example, the subfolder called **s1** means that it contains images for **person 1**.

With that in mind, the directory structure tree for training data is as follows:



On the other hand, The folder*test-data* contains images that we will use to test our face recognition program after we have trained it successfully.

Considering that the OpenCV face recognizer only accepts labels as integers, we need to define a mapping between integer tags and the person’s actual name.

So, below I am defining the mapping of a person’s integer labels and their respective names.

I have a sneaking suspicion that my follower thief is none other than Elvis Presley. Why else do I keep listening to 1950s rock and roll?

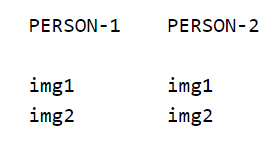
**Note:**As we have not assigned label 0 to anyone, the mapping for tag 0 is empty:

#### DATA PREPARATION FOR FACE RECOGNITION.

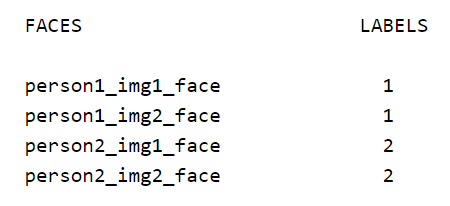
#### Well, to know which face belongs to which person, OpenCV face recognizer accepts information in a particular format. In fact, it receives two vectors:

* One is the faces of all the people.
* The second is the integer labels for each face.

For example, if we had two individuals and two images for each one:



Then, the data preparation step will produce following face and label vectors:



In detail, we can further divide this step into the following sub-steps:

1. Read all the sub folders names provided in the folder *training-data.* In this tutorial; we have folder names:s1, s2.
2. Extract label number. Remember that all the sub folders containing images of a person following the format:sLabel where Label is an integer representing each person. So for example, folder name: s1 means that the person has label 1, s2 means the person's label is 2, and so on. We will assign the integer extracted in this step to every face detected in the next one.
3. Read all the images of the person, and apply **face detection** to each one.
4. Add each face to **face vectors** with the corresponding person label (extracted in above step)

Face Detection Code

def detect\_face(img):

#convert the test image to gray scale as opencv face detector expects gray images

gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

#load OpenCV face detector, I am using LBP which is fast

#there is also a more accurate but slow: Haar classifier

face\_cascade = cv2.CascadeClassifier('opencv-files/lbpcascade\_frontalface.xml')

#let's detect multiscale images(some images may be closer to camera than others)

#result is a list of faces

faces = face\_cascade.detectMultiScale(gray, scaleFactor=1.2, minNeighbors=5);

#if no faces are detected then return original img

if (len(faces) == 0):

return None, None

#under the assumption that there will be only one face,

#extract the face area

x, y, w, h) = faces[0]

#return only the face part of the image

return gray[y:y+w, x:x+h], faces[0]

Training Data Code

#and will return two lists of exactly same size, one list

#of faces and another list of labels for each face

def prepare\_training\_data(data\_folder\_path):

#------STEP-1--------

#get the directories (one directory for each subject) in data folder

dirs = os.listdir(data\_folder\_path)

#list to hold all subject faces

faces = []

#list to hold labels for all subjects

labels = []

#let's go through each directory and read images within it

for dir\_name in dirs:

#our subject directories start with letter 's' so

#ignore any non-relevant directories if any

if not dir\_name.startswith("s"):

continue;

#------STEP-2--------

#extract label number of subject from dir\_name

#format of dir name = slabel

#, so removing letter 's' from dir\_name will give us label

label = int(dir\_name.replace("s", ""))

#build path of directory containing images for current subject subject

#sample subject\_dir\_path = "training-data/s1"

subject\_dir\_path = data\_folder\_path + "/" + dir\_name

#get the images names that are inside the given subject directory

subject\_images\_names = os.listdir(subject\_dir\_path)

#------STEP-3--------

#go through each image name, read image,

#detect face and add face to list of faces

for image\_name in subject\_images\_names:

#ignore system files like .DS\_Store

if image\_name.startswith("."):

continue;

#build image path

#sample image path = training-data/s1/1.pgm

image\_path = subject\_dir\_path + "/" + image\_name

#read image

image = cv2.imread(image\_path)

#display an image window to show the image

cv2.imshow("Training on image...", image)

cv2.waitKey(100)

#detect face

face, rect = detect\_face(image)

#------STEP-4--------

#for the purpose of this tutorial

#we will ignore faces that are not detected

if face is not None:

#add face to list of faces

faces.append(face)

#add label for this face

labels.append(label)

cv2.destroyAllWindows()

cv2.waitKey(1)

cv2.destroyAllWindows()

return faces, labels

Data Preparation Code

|  |  |
| --- | --- |
|  | #let's first prepare our training data  #data will be in two lists of same size  #one list will contain all the faces  #and the other list will contain respective labels for each face  print("Preparing data...")  faces, labels = prepare\_training\_data("training-data")  print("Data prepared")  #print total faces and labels  print("Total faces: ", len(faces))  print("Total labels: ", len(labels)) |

### ****2.2 TRAIN FACE RECOGNIZER.****

As we mentioned earlier, OpenCV comes equipped with three face recognizers.

1. EigenFaces: cv2.face.createEigenFaceRecognizer()
2. FisherFaces: cv2.face.createFisherFaceRecognizer()
3. Local Binary Patterns Histogram (LBPH): cv2.face.LBPHFisherFaceRecognizer()

We are going to use now **LBPH recognizer**this time and see if my theory about the ghost of Elvis stealing my followers is right. It doesn't matter which of the OpenCV's face recognition programs you use because the code will remain the same. You just have to change one line, which is the face recognizer initialization line given below.

|  |  |
| --- | --- |
|  | #create our LBPH face recognizer  face\_recognizer = cv2.face.createLBPHFaceRecognizer()  #or use EigenFaceRecognizer by replacing above line with  #face\_recognizer = cv2.face.createEigenFaceRecognizer()  #or use FisherFaceRecognizer by replacing above line with  #face\_recognizer = cv2.face.createFisherFaceRecognizer()  #train our face recognizer of our training faces  face\_recognizer.train(faces, np.array(labels)) |

### ****2.3 PREDICTION****

This is where we get to see if our algorithm is recognizing our individual faces or not.

We’re going to take one test image of each person, use face detection and then pass those faces to our trained face recognizer. Then we find out if our face recognition is successful.

Below are some utility functions that we will use for drawing bounding box (rectangle) around the face and putting the person's name near the face bounding box.

Drawing Rectangle Around The Face Code

|  |  |
| --- | --- |
|  | #function to draw rectangle on image  #according to given (x, y) coordinates and  #given width and heigh  def draw\_rectangle(img, rect):  (x, y, w, h) = rect  cv2.rectangle(img, (x, y), (x+w, y+h), (0, 255, 0), 2)  #function to draw text on give image starting from  #passed (x, y) coordinates.  def draw\_text(img, text, x, y):  cv2.putText(img, text, (x, y), cv2.FONT\_HERSHEY\_PLAIN, 1.5, (0, 255, 0), 2) |

Prediction Method Code

#and draws a rectangle around detected face with name of the

#subject

def predict(test\_img):

#make a copy of the image as we don't want to change original image

img = test\_img.copy()

#detect face from the image

face, rect = detect\_face(img)

#predict the image using our face recognizer

label= face\_recognizer.predict(face)

#get name of respective label returned by face recognizer

label\_text = subjects[label]

#draw a rectangle around face detected

draw\_rectangle(img, rect)

#draw name of predicted person

draw\_text(img, label\_text, rect[0], rect[1]-5)

return img

|  |
| --- |
| print("Predicting images...")    #load test images  test\_img1 = cv2.imread("test-data/test1.jpg")  test\_img2 = cv2.imread("test-data/test2.jpg")    #perform a prediction  predicted\_img1 = predict(test\_img1)  predicted\_img2 = predict(test\_img2)  print("Prediction complete")    #display both images  cv2.imshow(subjects[1], predicted\_img1)  cv2.imshow(subjects[2], predicted\_img2)  cv2.waitKey(0)  cv2.destroyAllWindows() |

