...Finding out who's stealing my followers using Face Recognition and OpenCV...

DID YOU HEAR THAT?!

I think it's him again... Damn, I don't know what's what right now!

I'm not only losing my friends... Wait, scratch that, I don't have any friends... I'm losing my followers!

I'm not overreacting... My followers are everything to me!

I didn't think it would be necessary, but when necessity stops at your door, you have to answer it. I'm bringing out the big guns. That's right; I'm going all-out **Face Recognition** on this creep.

I know for a fact that there's something out there. My Snapchat never lies.

Well, except for all the filters: dog-ears, sparkle, rainbows, etc. but let's call that beauty enhancement.

If you followed my last post, Snapchat has been finding another person whenever I take a selfie. If you didn't follow my previous post, then you're dead to me!

JK! **Click here to see my last post on Face Detection** and catch up.

OK, BACK TO BUSINESS!

I've decided to attack this creep with Facial Recognition because I am not afraid of no ghost, but I need to know who this is!

So, according to my friend @superdazzlepimpnerd, Facial Recognition will do the job for me.

But first, I have to understand what it takes to recognize a face; how the technology works and how I can use it to find out who this ghost is!

In that sense, this second part of my story will unfold like this:

1. I'll explain the nerdy theory of OpenCV's three built-in face recognizers.
2. Then, I'll show you the coding process I followed.
3. Next, I'll compare all three face recognizers, with their pros and cons.
4. Finally, I'll bust this creep once and for all!

By the end of this tutorial, hopefully, I will be able to solve this mystery using my Face Recognizer, and you will learn how to build yours and use it in whatever the heck you want. Seriously, I don't care.

Just before we start:

Can I get a little peace up in here?

I've been listening to random 50's music up in the morning. Late at night. In the shower...

Is that "Love me tender" again? Am I listening to "Love me tender" for the 1,543rd time right now?!

That is just way too many times.

1. BACKGROUND ON FACE RECOGNITION

When you look at an apple, your mind immediately tells you: that is an apple. **This process is recognition** in the simplest of terms. So, what's facial recognition? The same, but for faces, obviously.

But, the real question is:

How can a computer recognize a face?

Take a real life example:

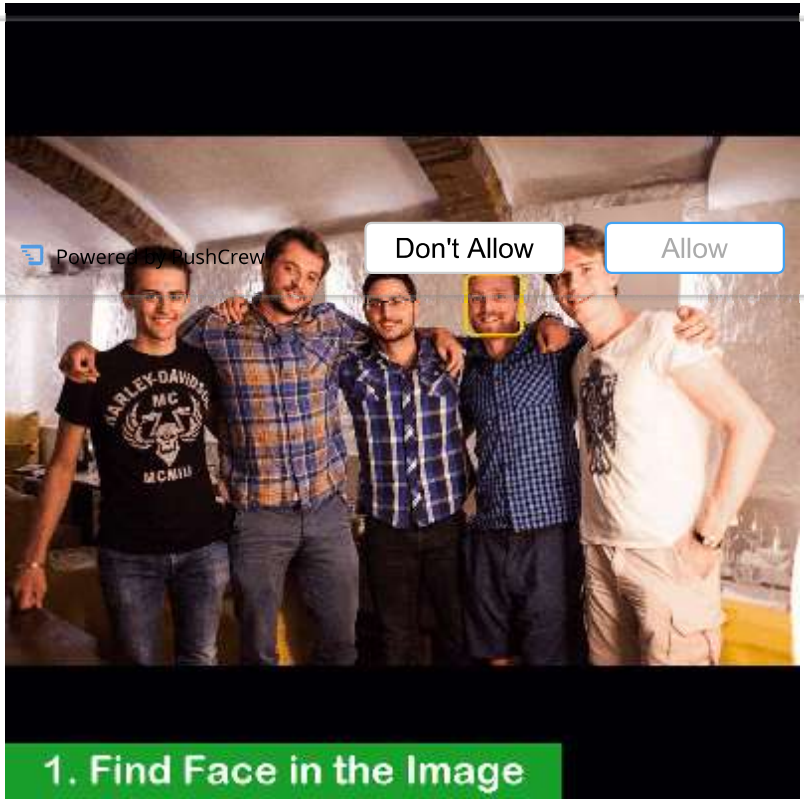
When you meet someone for the first time, you don't know who that person is at once, right? While he's talking to you or shaking your hand, you're looking at his face: eyes, nose, mouth, skin tone... This process is your mind gathering data and training for face recognition.

Next, that person tells you that his name is Kirill (*yes, our All-Star Data Science Mentor*). So, your brain has already gotten the face data, and now it has learned that this data belongs to Kirill.

The next time you see Kirill or see a picture of his face, your mind will follow this exact process:

1. **Face Detection:** Look at the picture and find a face in it.
2. **Data Gathering:** Extract unique characteristics of Kirill's face that it can use to differentiate him from another person, like eyes, mouth, nose, etc.
3. **Data Comparison:** Despite variations in light or expression, it will compare those unique features to all the features of all the people you know.

4. **Face Recognition:** It will determine “Hey, that’s my boy Kirill!”



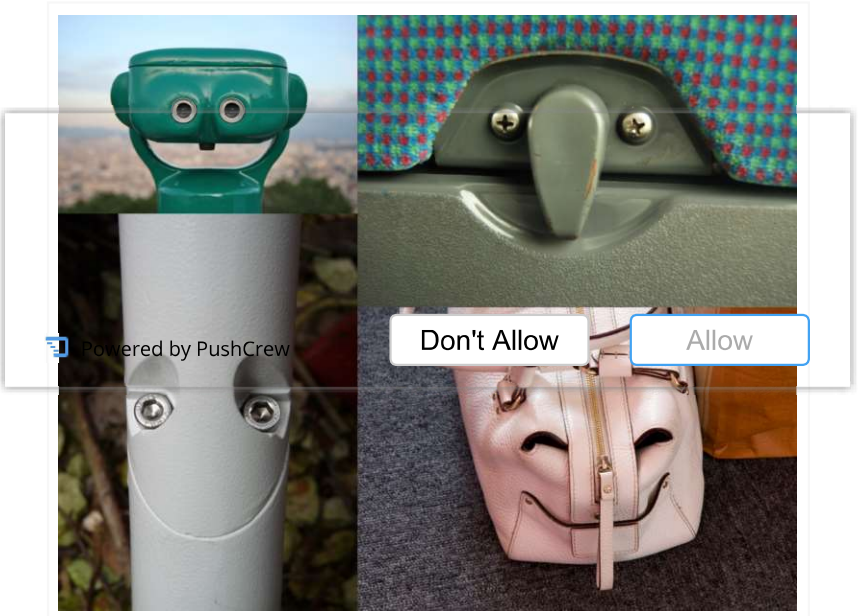
Our mind's Face Recognition Process

Then, the more you meet Kirill, the more data you will collect about him, and the quicker your mind will be able to recognize him.

Or, at least you should. Whether or not you are good with names is another story.

Here is when it gets better:

Our human brains are wired to do all these things automatically. In fact, we are very good at detecting faces almost everywhere:



Meet my (Loser) crew!

Computers aren't able, yet, to do this automatically, so we need to *teach them* how to do it step-by-step.

But, you already knew that which is why you're reading this article (duh).

However, you probably assumed that it's incredibly difficult to code your computer to recognize faces, right? Well, keep reading my friend cause I am here to end with this creep and your superstitions as well.

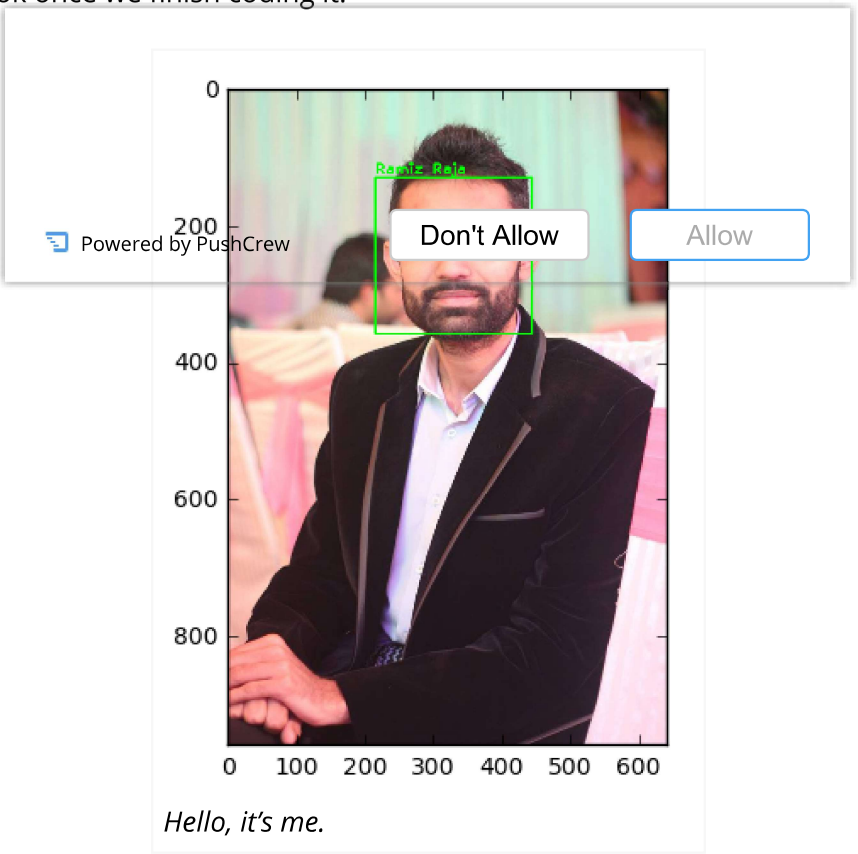
2. THEORY OF OPENCV FACE RECOGNIZERS

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.

It's that simple!

And this is how our Face Recognizer will look once we finish coding it:



Hello, it's me.

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()

You might be wondering:

"Which Face Recognizer should I use and when?"

Here is a summary of each one that will answer that question.

Let's rock!

2.1 EIGENFACES FACE RECOGNIZER

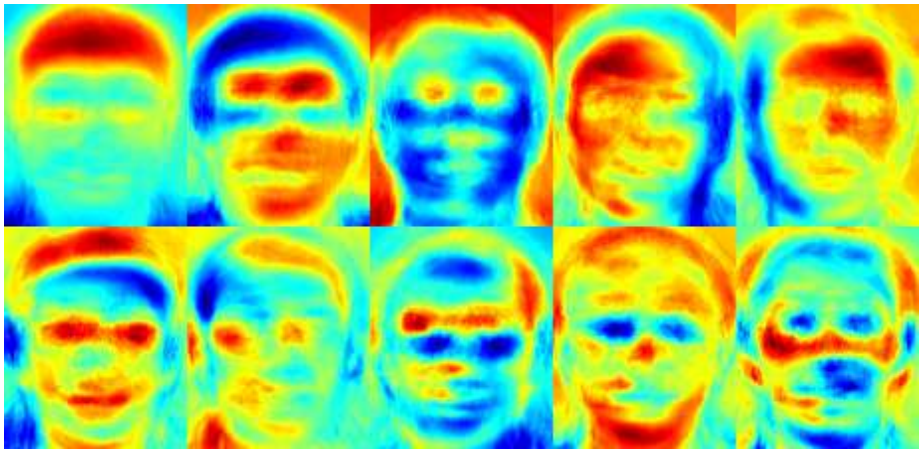
This algorithm considers the fact that **not all parts of a face are equally important or useful for face recognition**. Indeed, when you look at someone, you recognize that person by his distinct features, like the eyes, nose, cheeks or forehead; and how they vary respect to each other.

In that sense, you are focusing on the *areas of maximum change*. 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.

In this way, is how EigenFaces recognizer works. It looks at all the training images of all the people *as a whole* and tries to extract the components which are relevant and useful and discards the rest. These important features are called **principal components**.

Note: We will use the terms: *principal component*s, *variance*, *areas of high change* and *useful features* indistinctly as they all mean the same.

Below is an image showing the variance extracted from a list of faces.



EigenFaces Face Recognizer Principal Components. Source: **docs.opencv.org**

You can see that the useful features represent faces which receive the name of *Eigen Faces*. I mean, *how else do you think the algorithm got its name?*

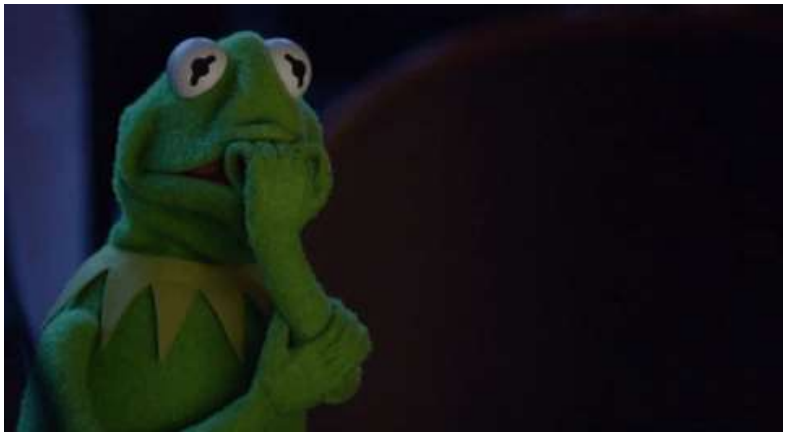
So, EigenFaces recognizer trains itself by extracting principal components, but it also **keeps a record** of which ones belong to which person. Thus, whenever you introduce a new image to the algorithm, it repeats the same process as follows:

1. Extract the principal components from the new picture.
2. Compare those features with the list of elements stored during training.
3. Find the ones with the best match.
4. Return the 'person' label associated with that best match compone

In simple words, it's a game of matching.

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.'

Face recognition picks up on human things, dominated by shapes and shadows: two eyes, a nose, a mouth.



So, maybe that shadow was an Eigen Face?!

2.2 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*.

But here's the kicker:

Consider the lighting changes in following images:

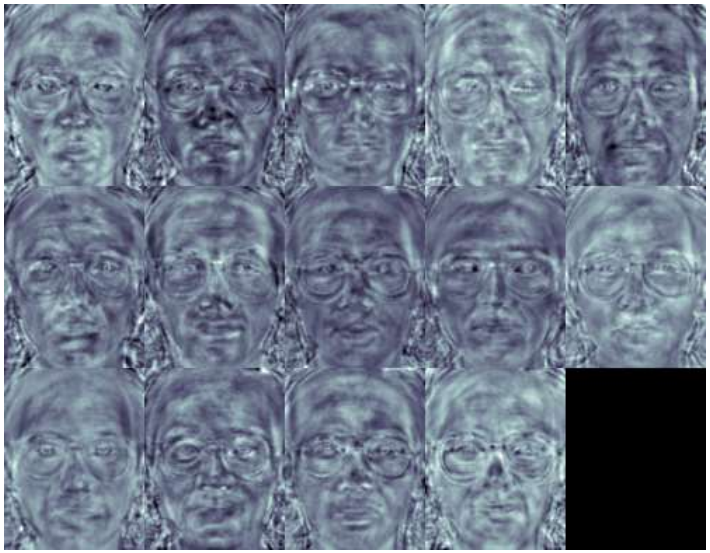
Since EigenFaces also finds illumination as a useful component, it will find this variation very relevant for face recognition and may discard the features of the other people's faces, considering them less useful. In the end, the variance that EigenFaces has extracted represents *just one individual's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

HOW TO FIX THIS ISSUE?

We can do it by tunning EigenFaces so that it extracts useful features from the faces of each person separately instead of extracting them from all the faces combined. In this way, even if one person has high illumination changes, it will not affect the other people's features extraction process.

Precisely, **FisherFaces** face recognizer algorithm extracts principal components that differentiate one person from the others. In that sense, an individual's components do not dominate (become more useful) over the others.

Below is an image of principal components using FisherFaces algorithm.



FisherFaces Face Recognizer Principal Components.
Source: **docs.opencv.org**

You can see that the features represent faces which receive the name of *Fisher Faces*. *Are you noticing a theme with the names of the algorithms?*

One thing to note here is that FisherFaces only prevents features of one person from becoming dominant, but it still considers illumination changes as a useful feature. We know that light variation is not a useful feature to extract as it is not part of the actual face.

Then, how to get rid of this problem?

Here is where our next face recognizer comes in.

2.3 LOCAL BINARY PATTERNS HISTOGRAMS (LBPH) FACE RECOGNIZER

I wrote a detailed explanation of Local Binary Patterns Histograms in my previous article on **face detection**, which I'm sure you've read by now.

So, here I will just give a brief overview of how it works.

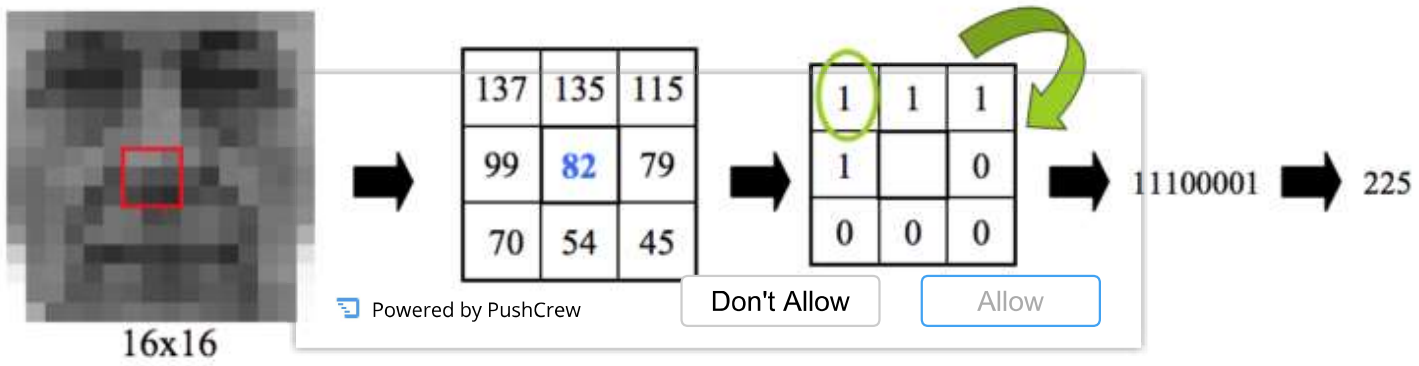
We know that *Eigenfaces* and *Fisherfaces* are both affected by light and, in real life, we can't guarantee perfect light conditions. *LBPH face recognizer is an improvement to overcome this drawback*.

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

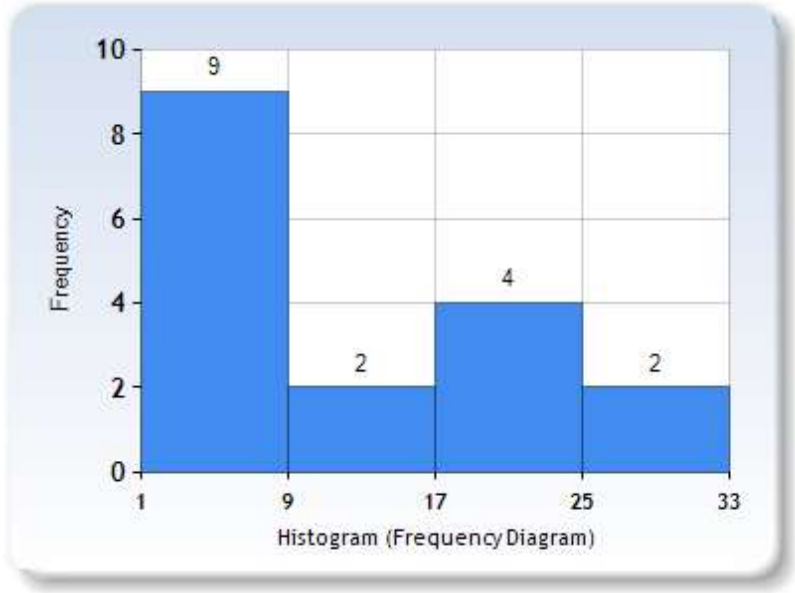
Take a 3×3 window and move it across one image. At each move (each local part of the picture), compare the pixel at the center, with its surrounding pixels. Denote the neighbors with intensity value *less than or equal to* the center pixel by *1* and the rest by 0.

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** (as shown in above image) and then you make a **histogram** of all of those decimal values. A sample histogram looks like this:



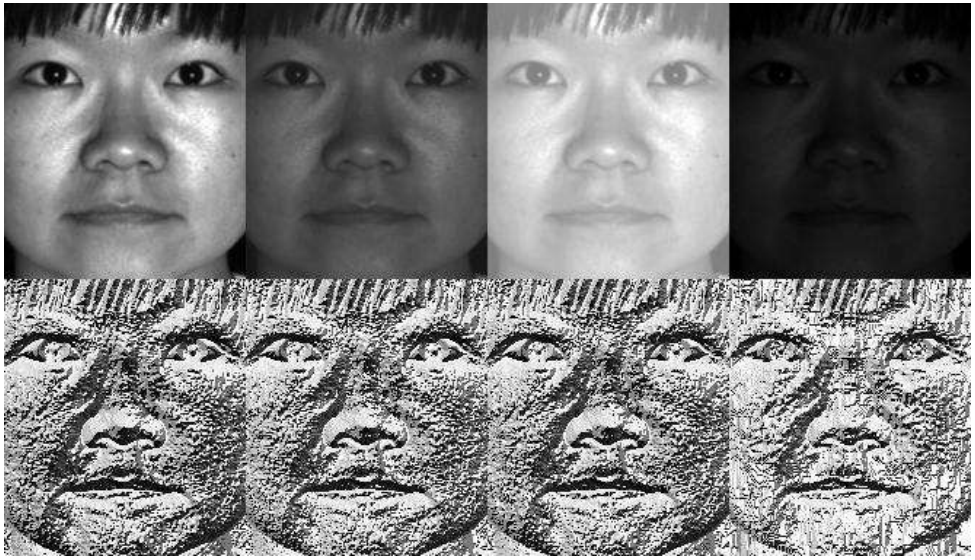
Histogram Sample.

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.

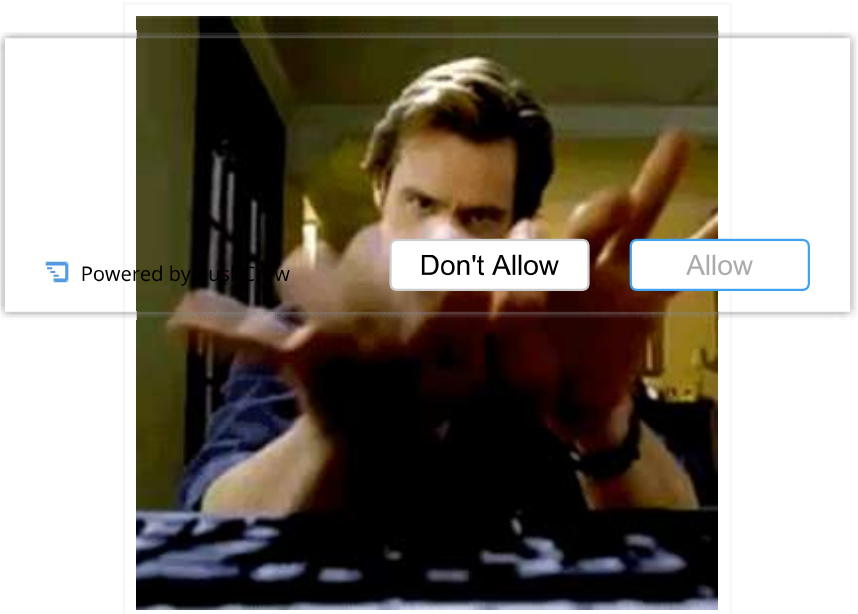
Below is a group of faces and their respective local binary patterns images. You can see that the **LBP faces are not affected by changes in light conditions**:



LBPH Face Recognizer Principal Components. Source: **docs.opencv.org**

The theory part is over, and now it is time to unmask this moron.

Brace yourself. Here comes the coding!



This is how I code. In the zone. Much speed.

3. CODING FACE RECOGNITION USING PYTHON AND OPENCV

We are going to divide the Face Recognition process in this tutorial into three steps:

1. **Prepare Training Data:** Read training images for each person/subject along with their labels, detect faces from each image and assign each detected face an **integer label** of the person it belongs.
2. **Train Face Recognizer:** Train OpenCV's LBPH recognizer by feeding it the data we prepared in step 1.
3. **Prediction:** Introduce some test images to face recognizer and see if it predicts them correctly.

To detect faces, I will use the code from my previous article on **face detection**.

Before we start the actual coding, we need to install the **Code Dependencies** and import the **Required Modules**:

CODE DEPENDENCIES

Install the following dependencies:

1. **OpenCV 3.2.0**
2. **Python v3.5**
3. **NumPy** that makes computing in Python easy. It contains a powerful implementation of N-dimensional arrays which we will use for feeding data as input to OpenCV functions.

REQUIRED MODULES

Import the following modules:

- **cv2:** This is the *OpenCV* module for Python used for face detection and face recognition.
- **os:** We will use this Python module to read our training directories and file names.
- **numpy:** This module converts Python lists to *numpy* arrays as OpenCV face recognizer needs them for the face recognition process.

In [1]

```
#OpenCV module
import cv2
#os module for reading training data directories and paths
import os
#numpy to convert python lists to numpy arrays as it is needed by OpenCV face recognizers
import numpy as np
```

3.1 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:

In [2]

```
1 #there is no label 0 in our training data so subject name for index/label 0 is empty
2 subjects = ["", "Ramiz Raja", "Elvis Presley"]
```

DATA PREPARATION FOR FACE RECOGNITION.

Perhaps you are thinking:

Why are are we talking about preparing data?

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:

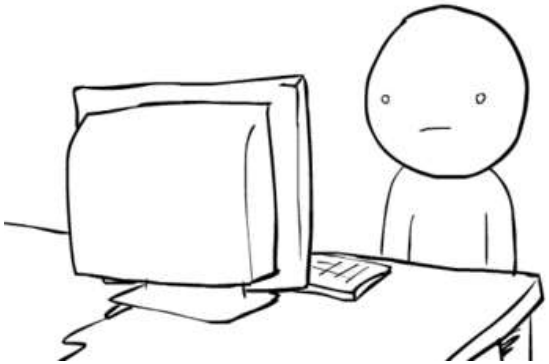
1. Read all the sub folders names provided in the folder *`training-data`*. In this tutorial; we have folder names:`s1, s2.`

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)

LET'S CODE THIS PART!

As I said before, we are going to use the code of my last article on face detection.

"Wait, what article? What is this dude talking about?!"



Unbelievable. Here is the **link** again.

In [3]

```
#function to detect face using OpenCV
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]
```


As you can see, we are using OpenCV's **LBP face detector**. To break it down:

- In *line 4*, we converted the image to grayscale because OpenCV mostly operates in gray scale.
- Then, in *line 8*, we loaded LBP face detector using class`cv2.CascadeClassifier`.
- After that, in *line 12,* we used class`cv2.CascadeClassifier`' `detectMultiScale` method to detect all the faces in the image.
- Next, in *line 20*, from exposed faces, we only picked the first one, because in a person's portrait it is supposed to be only one face *(under the assumption that there will be only one prominent face)*.
- As faces returned by the method `detectMultiScale`are rectangles (x, y, width, height) and not actual faces images, we have to extract face area from the main image. So in *line 23*, we extracted face area from the gray picture and return both the face image area and face rectangle.

Now, you have a face detector. Also, you know the four steps to data preparation. I think we are ready to code it.

Let's go!

In [4]

```
#this function will read all persons' training images, detect face from each image
#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
```

```
64  faces.append(face)
65  #add label for this face
66  labels.append(label)
67
68  cv2.destroyAllWindows()
69  cv2.waitKey(1)
70  cv2.destroyAllWindows()
71
72  return faces, labels
```


We have defined a function that takes the path where it stores training subjects' folders as a parameter. This function follows the same data preparation sub steps that we reviewed previously.

(step-1) In *line 8,* we used the method`os.listdir` to read names of all folders stored on the path, so they start functioning as a parameter.

In *line 10-13* we defined labels and faces vectors.

(step-2) After that, we went through all subjects' folder names and from each one we extracted, in *line 27,* the label information. As folder names follow the `sLabel`naming convention, just by removing the letters from folder name we will get the tag assigned to that subject.

(step-3) In *line 34*, we read all the current person's images, and in *lines 39-66* we went through traverse those images one by one.

In *lines 53-54,* we used OpenCV's `imshow(window_title, image)` along with OpenCV's `waitKey(interval)` methods to display the current picture.

The method`waitKey(interval)` pauses the code flow for the given interval (milliseconds). So, we are using a 100ms interval so that we can view the image window for that time.

In *line 57*, we implemented face detection on the current picture.

(step-4) In *lines 62-66*, we added the detected face and label to their respective vectors.

But a function can't do anything unless we call it on some data that it has to prepare, right?

Don't worry; I have got data for two faces. I am sure you will recognize at least one of them!

I'm pretty sure that The King is the ghost, so I've put his image into the face recognition step. I recognize his face, and I am sure you will recognize him, too!

Look at these handsome faces

Let's use these images of two handsome devils to prepare data for training of our Face Recognizer. Here is a simple code to do that:

In [5]

```
#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))
```

It's time to train our face recognizer so that, once trained, it can recognize new faces of the people it's been trained on. Ready? Ok then let's get training.

3.2 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.

In [6]

```
#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()
```

Now that we have initialized our face recognizer and we also have prepared our training data, it's time to train. We will do that by calling the method`train(faces-vector, labels-vector)` of face recognizer.

In [7]

```
#train our face recognizer of our training faces
face_recognizer.train(faces, np.array(labels))
```

Did you notice that instead of passing vector`labels` directly to face recognizer, we are first converting it to **numpy** array? The reason is that OpenCV expects labels vector to be a `numpy`array.

This is my favourite part now, when we find out if the thief is the King of Rock and Roll or not.
The anticipation is killing me...
Here we go!
It's time for the...

3.3 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.

In [8]

```
#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)
```


The first function`draw_rectangle` draws a rectangle on the image based on given coordinates. It uses OpenCV's built in function`cv2.rectangle(img, topLeftPoint, bottomRightPoint, rgbColor, lineWidth)` to do so.

The second function`draw_text` uses OpenCV's built in function`cv2.putText(img, text, startPoint, font, fontSize, rgbColor, lineWidth)` to draw text on the image.

Now that we have the drawing functions, we just need to call the face recognizer's `predict(face)` method to test our face recognizer on test images. The following function does the prediction for us:

In [9]

```
#this function recognizes the person in image passed
#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
```

```
13 label_text = subjects[label]
14
15 #draw a rectangle around face detected
16 draw_rectangle(img, rect)
17 #draw name of predicted person
18 draw_text(img, label_text, rect[0], rect[1]-5)
19
20 return img
```


If we break this last code down:

- **line-6** read the test image.
- **line-7** detect the face from test image
- **line-11** recognize the face by calling face recognizer's `predict(face)` method. This method will return a label.
- **line-12** get the name associated with the tag.
- **line-16** draw rectangle around the detected face.
- **line-18** draw name of the predicted individual above face rectangle.

Now that we have the prediction function well defined, the next step is to call this function on our test images, display them to see if our face recognizer correctly performs face recognition.

So, let's do it. This step is what we have been waiting for...

It's Now or Never

In [10]

```
1  print("Predicting images...")
2
3  #load test images
4  test_img1 = cv2.imread("test-data/test1.jpg")
5  test_img2 = cv2.imread("test-data/test2.jpg")
6
7  #perform a prediction
8  predicted_img1 = predict(test_img1)
9  predicted_img2 = predict(test_img2)
10 print("Prediction complete")
11
12 #display both images
13 cv2.imshow(subjects[1], predicted_img1)
14 cv2.imshow(subjects[2], predicted_img2)
15 cv2.waitKey(0)
16 cv2.destroyAllWindows()
```

The face recognition worked! It can tell who I am and who Elvis Presley is, so it's been trained correctly.
I had a suspicious mind...
I kept listening to random 50s music and dancing with my hips; not trying to be sexy. It just happened. I mean I didn't remember buying those blue suede shoes but I thought they might have been left here by someone else.
I'm all shook up!

4. ENDNOTES

You can download the complete code and relevant files from this Github

Face Recognition is fascinating on and OpenCV has made it incredibly straightforward and easy for us to code it. It just takes a few lines of code to have a fully working face recognition application.

We can switch between all three OpenCV face recognizers by changing only a single line of code. It's that simple.

Although EigenFaces, FisherFaces, and LBPH face recognizers are fine, there are even better ways to perform face recognition like using **Histogram of Oriented Gradients (HOGs)** and **Neural Networks.**

More advanced face recognition algorithms are implemented using a combination of OpenCV and Machine Learning.

I have plans to write some articles for those more advanced methods as well.

But for now, I want to figure out if the King has left the building...

Please let me know in your comments what do you think about this tutorial and my future plans (now that I have more peace in my life)