

Approach:

First, I inspected the provided data set. It was huge. I knew that my first task would be to refine and limit the size of the dataset. To refine, I only selected images of faces that were forward facing (front-profile) and the subject was not wearing eyewear. I will discuss how to build a more robust classifier that accepts more diverse inputs in the conclusion to this report. Limiting the dataset allowed me to get a functional classifier working, and after that is accomplished, only then should people creating a biometrics classifier work to make it more functional and commercial grade.

Then, I implemented my facial recognition system. I used OpenCV's haar cascade system to implement both train and test the recognizer. In short, OpenCV comes with many files called *cascades* that represent high-precision trained information about certain features. There are cascades to extract eyes, smiles, side-profiles, front-profiles, and so on and so forth. Since I had limited the data to just front-profile images of faces, I used the front-profile haar cascade to train the recognizer. To train, I iterated through the database of each testing set, extracting a *region of interest* from each photo (where each ROI was the extracted facial features from the image). To test, I used OpenCV's *LBPHFaceRecognizer* to compare a given input image to the known dataset. I then used the confidence interval to determine whether it was a true accept, true reject, false accept, or false reject. I will expand on training and testing later in the report.

My recognizer is callable from the command line with the command, `$ FacesMain.py`, and assumes `Python 3.7`, `cv2`, `matplotlib`, `numpy`, and `os` are installed. All training images should be stored in directory "images1/train" with each correct label corresponding to the folder within "images1/train", and the faces images for each label are stored within that corresponding folder. The same setup is needed for "images1/test".

About the Dataset:

My training set consists of 10 unique individuals, where each person has 5 photos. So, my training set has 50 photos. My testing set has 15 unique individuals, where each person has 1 photo. So my testing set has 15 photos. Additionally, 10 individuals in my testing set are in my training set, and 5 individuals in my testing set are *not* in my training set.

Dataset found in "images1" directory:

| <u>Set/Description</u> | <u>Individuals</u> | <u>Photos per Individual</u> | <u>Total Photos</u> |
|-------------------------------|---------------------------|-------------------------------------|----------------------------|
| <u>Training</u> | 10 | 5 | 50 |
| <u>Testing</u> | 15 (10 from training) | 1 | 15 |




Training:

When my classifier is invoked, the first task is to train the recognizer from the given dataset found in "images1/train".

In FaceMain.py, on line 124, the recognizer is trained by calling, FacesTrain.main("images1\\train"). In, FacesTrain.py, the following logic is executed:

1. Construct a face_cascade object using the haar cascade, "haarcascade_frontalface_alt2.xml" file found in the "cascades/data" directory.
2. Initialize an *LPBFaceRecognizer*, named recognizer.
3. Iterate through the database of training photos, and for each photo:
 - a. Convert the photo from RGB to Grayscale.
 - b. Extract the *Region of Interest*, represented as a 2-D numpy array by using the haar cascade.
 - c. Append the current ROI to the x_train list, and append the current known training label to the y_labels list.
4. With x_train and y_labels construct, train the recognizer by invoking, recognizer.train(x_train, y_labels) and save this recognizer information as "trainer.yml" in the working directory.

For example, when an input image, "1.jpg" in training folder "9003", is being iterated over, it will have these properties:

| <u>Image</u> | <u>Region of Interest</u> | <u>Extraction</u> |
|---|---|---|
|  |  |  |

I do this for all of the photos, adding each pixel array of the extraction to x_labels the the corresponding label to y_labels. Then, I construct the recognizer with all of this information.

The recognizer constructs a template for identifying whether a given input belong to one of the trained labels, from the trained data. After training a recognizer, testing can then occur.

Testing:

After training, `FacesMain.py`, will then test on the images stored in the “*images1/test*” directory. For each test photo, the following logic is executed:

1. Read the image and convert it from RGB to Grayscale.
2. Use the haar cascade to extract the *Region of Interest*.
3. Gather a classification label and confidence interval from the trained recognizer, by invoking, `out_id, conf = recognizer.predict(roi)` on line 54 in `FacesMain.py`.
4. If the confidence interval is greater than a threshold (hardcoded to 5), the classification is reject. Else, if the `out_id` is the same as the testing label id, then it is a correct classification. Otherwise, a false classification.
5. Likewise, if the testing input is rejected when it should have been accepted, then it is a false classification, and vice-versa.
6. For each testing, update these 4 counts:
 - a. `true_accept`: a photo that should have been accepted was accepted
 - b. `true_reject`: a photo that should have been rejected was rejected
 - c. `false_accept`: a photo that should have been rejected was accepted
 - d. `false_reject`: a photo that should have been accepted was rejected.

After testing, I can use these counts to analyze the effectiveness of my classifier.

Results:

After training the classifier on the training set and testing the classifier on the testing set, I obtained these results:

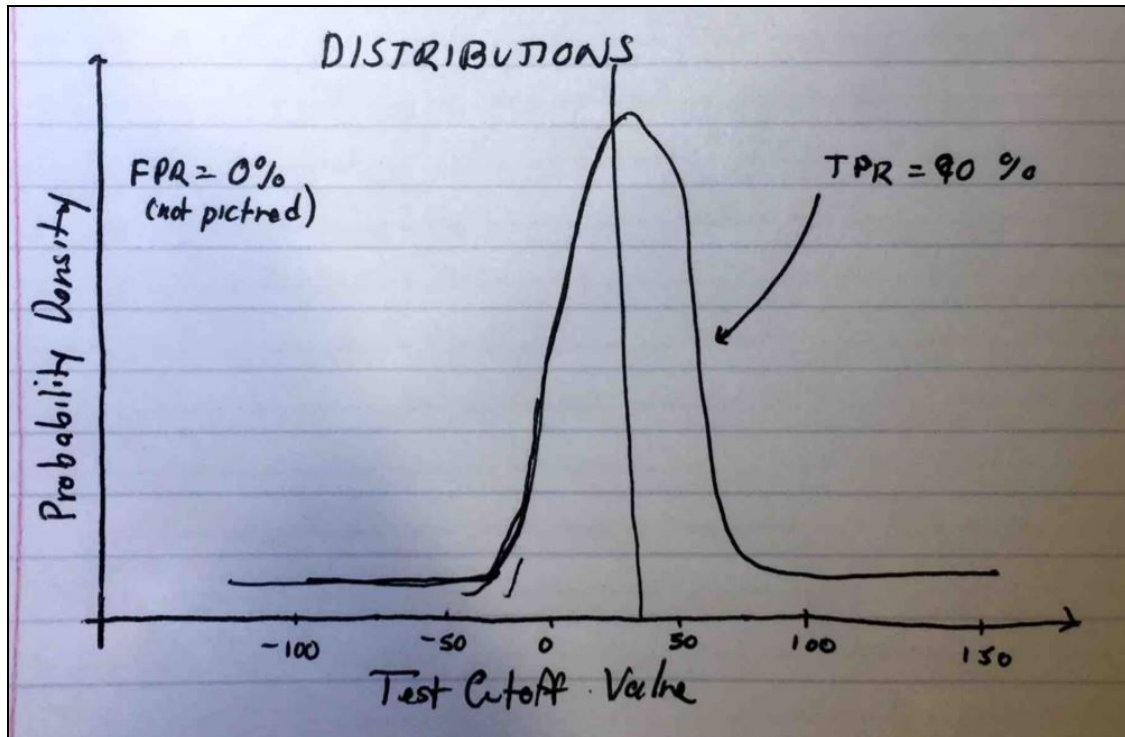
| <u>Category</u> | <u>Absolute Number</u> | <u>Rate</u> |
|------------------------|-------------------------------|--------------------|
| False reject | 1 | 1/10 = 10% |
| False accept | 0 | 0/6 = 0% |
| True accept | 9 | 9/10 = 90% |
| True reject | 6 | 6/6 = 100% |

This data is pretty good. In all, my classifier only made 1 false reject and no false accepts. Additionally, my classifier made 0 false accepts and all 6 true rejects. Those numbers are nearly perfect.

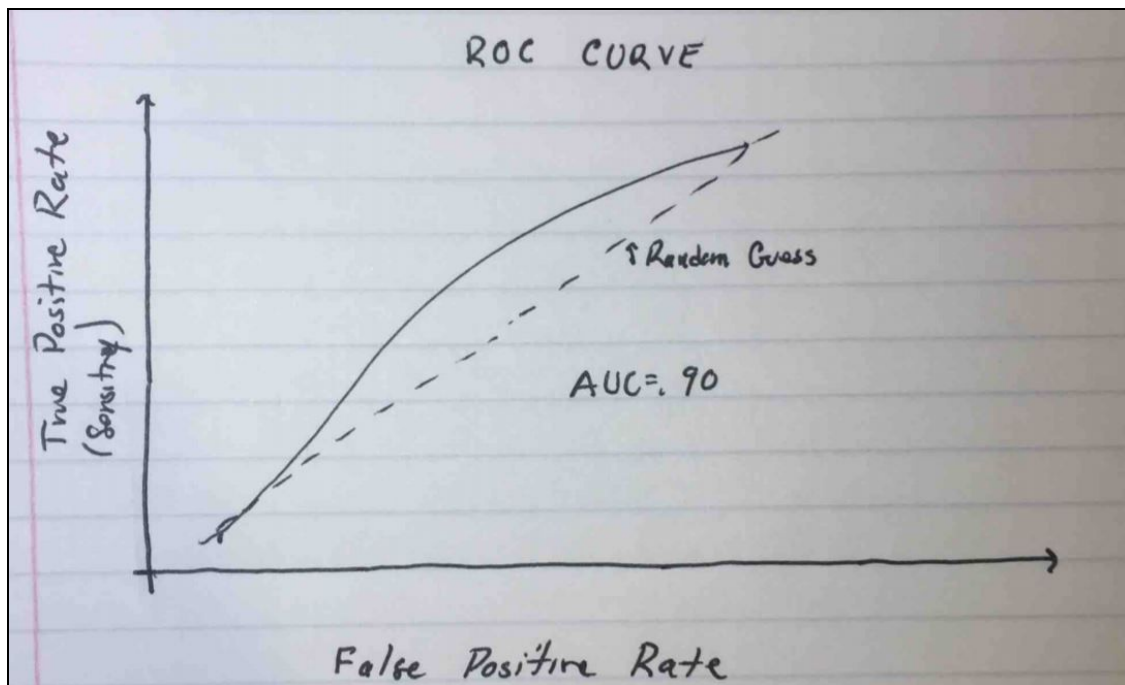
From this collection, I can say that the TPR (true positive rate) is 90% and the FPR (false positive rate) is 0%.

These findings can be further illustrated in the corresponding, TPR (true positive rate) and FPR (false positive rate) distributions, ROC curve, and CMC curves:

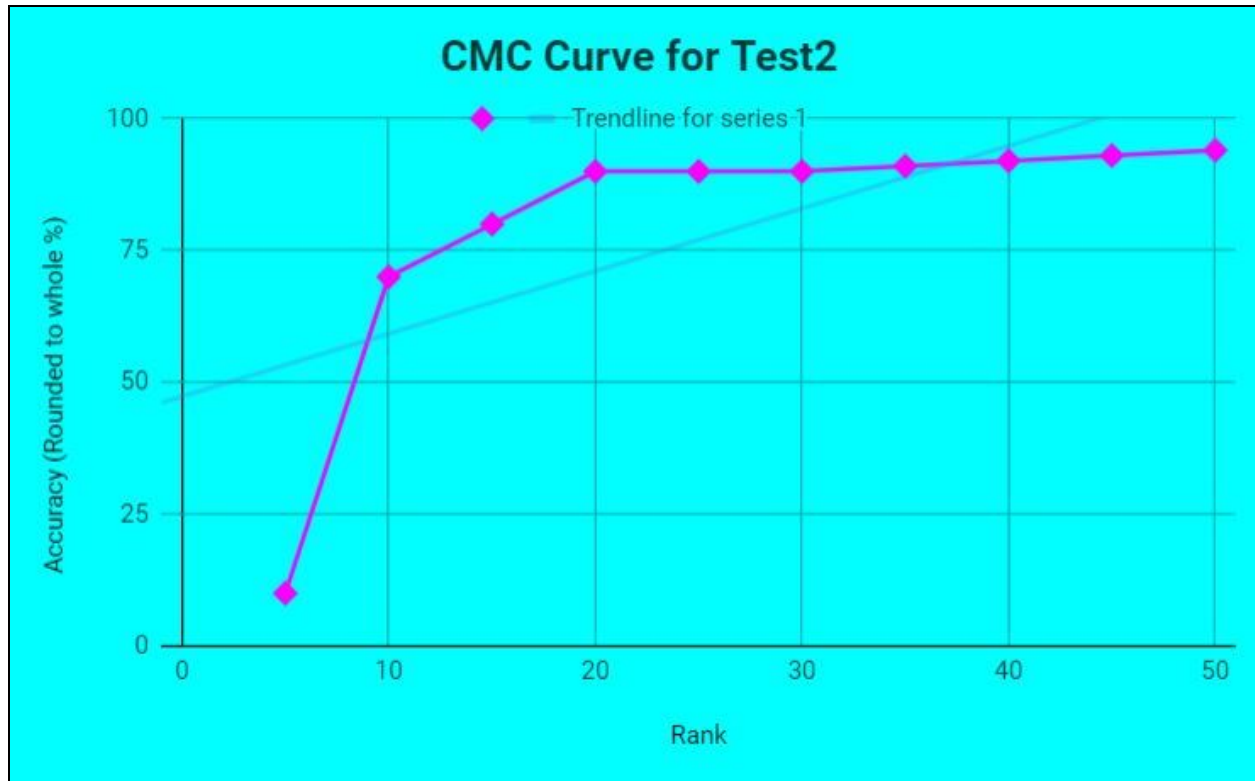
TPR and FNR Distributions for "images1/test"



ROC Curve for "images/test1"



CMC Curves for "test1/images"



Command Line Data from "images1/test"

Training recognizer on training data!
 Recognizer is trained!
 Testing recognizer on testing data!

 Testing: 90003
 Test in Train? Yes
 Recognizer thinks test in train? Yes
 Class: Correct
 Actual: 90003
 Predict: 90003
 Confidence: 1.05

 Testing: 90010
 Test in Train? Yes
 Recognizer thinks test in train? Yes
 Class: Correct
 Actual: 90010
 Predict: 90010
 Confidence: 0.92

 Testing: 90034
 Test in Train? Yes
 Recognizer thinks test in train? Yes
 Class: Correct
 Actual: 90034
 Predict: 90034
 Confidence: 0.87

 Testing: 90066
 Test in Train? Yes
 Recognizer thinks test in train? Yes
 Class: Correct
 Actual: 90066
 Predict: 90066
 Confidence: 2.64

 Testing: 90068
 Test in Train? No

 Recognizer thinks test in train? No
 Input Correctly Rejected!
 Confidence: 44.8075

 Testing: 90070
 Test in Train? Yes
 Recognizer thinks test in train? Yes
 Class: Correct
 Actual: 90070
 Predict: 90070
 Confidence: 0.98

 Testing: 90072
 Test in Train? Yes
 Recognizer thinks test in train? Yes
 Class: Correct
 Actual: 90072
 Predict: 90072
 Confidence: 0.66

| | | |
|--------------------------------------|-------------------------------------|--------------------------------------|
| ----- | Testing: 90138 | Confidence: 46.8277 |
| ---- | Test in Train? No | ---- |
| Testing: 90114 | Recognizer thinks test in train? No | Testing: 90318 |
| Test in Train? Yes | Input Correctly Rejected! | Test in Train? Yes |
| Recognizer thinks test in train? Yes | Confidence: 41.7812 | Recognizer thinks test in train? Yes |
| Class: Correct | ----- | Class: Correct |
| Actual: 90114 | ---- | Actual: 90318 |
| Predict: 90114 | Testing: 90144 | Predict: 90318 |
| Confidence: 1.03 | Test in Train? No | Confidence: 0.99 |
| ----- | Recognizer thinks test in train? No | ----- |
| ---- | Input Correctly Rejected! | ---- |
| Testing: 90114 | Confidence: 43.4982 | Testing: 90322 |
| Test in Train? Yes | ----- | Test in Train? No |
| Recognizer thinks test in train? No | ---- | Recognizer thinks test in train? No |
| ----- | Testing: 90296 | Input Correctly Rejected! |
| ---- | Test in Train? No | Confidence: 52.0681 |
| Testing: 90124 | Recognizer thinks test in train? No | --- |
| Test in Train? Yes | Input Correctly Rejected! | ---Rates!--- |
| Recognizer thinks test in train? Yes | Confidence: 45.6858 | True Accept: 9 |
| Class: Correct | ----- | True Reject: 6 |
| Actual: 90124 | ---- | False Accept: 0 |
| Predict: 90124 | Testing: 90296 | False Reject 1 |
| Confidence: 0.78 | Test in Train? No | |
| ----- | Recognizer thinks test in train? No | |
| ---- | Input Correctly Rejected! | |

Results Analysis:

In all, I am very pleased with the performance of my classifier. It makes almost no errors: when it receives an input photo from an individual of which it has no training data on, it attempts to find the nearest individual in the training set and ascribe that label to the requested input. However, when it does this, it report a very low confidence interval.

For example, on test “90068”, the classifier has a 41.812 confidence interval in its classification. This is much higher than the threshold (which is set to 5), so the classifier correctly rejects this input.

On the other hand, when the classifier receives input from an individual it trained on, it can easily classify in the input correctly. For example, on test “90124”, it correctly classifies the input as belonging to individual “90124” and has a confidence interval of .78. This is well within the established threshold, so it accepts the classification.

This indicates that this classifier comes from good training data. The data is well processed because I tried to choose front-profiles with good lighting and no eyewear. This allows the extracted *regions of interest* in each training photo to have clear features of which the recognizer can train on.

Clearly, this is not a commercial grade product. Despite its accuracy and good functionality, a good facial recognition system will need to accept input that includes partial profiles, side profiles, poor lighting, variations in facial hair, eyewear, hats, and so on and so forth. To make my recognition system more robust, I will need to implement a way to handle these inputs in the future. But for now, I think this current system is a highly functional way to begin.

Submitted Materials:

- */cascades/*
 - This directory contains the many *haar* cascades that can be used to detect regions like front profiles, side profiles, eyes, smiles etc...
- */images1/*
 - This directory contains the training and testing folder.
- */results/*
 - The directory contains the regions of interest for any extracted image.
- *FacesMain.py*
 - This python file contains the main driver for the biometric classification system.
- *FacesTrain.py*
 - This python file contains the code to train the recognizer.
- *HW3 Report*
 - This report.