CS479 Programming Assignment 3

Ashlee Ladouceur and Adam Lychuk

Due Date: 4/23/2019

Date Turned In: 4/23/2019

Division of work:

- Adam Lychuk designed and developed the code for Part A.
- Adam Lychuk designed and developed the code for Part A and Part B.
- Ashlee Ladouceur wrote the Technical Discussion and the Results and Discussion.
- Adam Lychuk wrote the Implementation Details and the Program Listings.

Technical Discussion

Technical Discussion

Principal Component Analysis (PCA) was utilized to reduce dimensionality of images by finding the k-most eigenvalues that retain a specified amount of information portrayed by N variables. For this project, we retained 80%, 90%, and 95% of the data given by the FERET face database. The FERET face database provided the training and testing set of images for recognition and classification in this project. Once PCA was used on the FERET database, M. Turk and A. Pentland's "Eigenfaces for Recognition" methodology to lead into classifying the recognized faces.

To use PCA for face recognition to reduce dimensionality, first obtain training faces $I_1, I_2, \cdots, I_M$ that are all centered and the same size. For all M images, use a $N \times N$ matrix to project into a $N^2 \times 1$ vector Γ_i to form the $N^2 xM$ matrix Γ. Equation 1 shows the formula to compute the average face vector ψ.

$$\psi = \frac{1}{M} \sum_{1}^{M} (\Gamma_i)$$

Equation 1: Computing average face vector.

Once the average face vector ψ is computed, one can center each face around 0 by subtracting the mean face using the formula shown in equation 2.

$$\phi_i = \Gamma_i - \psi$$

Equation 2: Center each face around 0.

Using matrix $A = [\phi_1, \phi_2, \cdots, \phi_M]$, one will find the M best eigenvectors u and the M best eigenvalues λ by using equation 3 to compute $A^T A$. Finally, one will keep only K eigenvectors that corresponding to the K largest eigenvalues.

$$A^T A v = \mu v$$

$$u = A v$$

$$\lambda = \mu$$

Equation 3: Computing $A^T A$.

After applying PCA to the images to reduce dimensionality, one should go on to use eigenfaces for recognition. Given an unknown image Γ, one will first normalize that image using $\phi = \Gamma - \psi$ before projecting onto the face space using the formula shown in equation 4.

$$\hat{\phi} = \sum_{i=1}^{K} \left(w_i u_i \right) + \psi$$

$$w_i = u_i^T \phi$$

$$\|u_i\| = 1$$

Equation 4: Project image onto face space.

Next, one will compute the distance in the face space using $e_d = \|\phi - \hat{\phi}\|$. If $e_d < T_d$, then the image Γ is a face. Once the image Γ is recognized, the distance from the projected Γ to a projection of each training face is computed to classify based on the smallest distance, shown in equation 5.

$$classification = min_i \left(\|\phi_{training-image-i} - \phi_{unknown-image}\| \right)$$

Equation 5: Classifying the face.

Implementation Details

Training:

In our approach, we used OpenCV for operations relating to images, normalization of our eigenface vectors, and computing the eigenvectors and eigenvalues of our system for the PCA approach. EigenFaces.cpp contains the implementation for the creation of

EigenFaces using the PCA approach which results in a trained model. The process begins using the function readInFaces() which takes in a directory file path pointing to the images and vector with data type Mat to store read faces. Each image is stored as a one-dimensional vector as it is read in. After all images are read in, the Average Face, Eigen Faces, and Eigen Values are created. This computation is contained in the function computeEigenFaces(). computeEigenFaces() computes the average face by adding each vector face image together and then dividing by the total number of face vectors. Matrix A, a matrix where each column is one of our facial image vectors, is created by concating each (face vector - average face) to a column in a Mat object. This defines our matrix A. A^T * A is then solved using cv::Eigen to find the eigenvectors and eigenvalues. Our eigenfaces are then calculated A*eigenvectors.

Each Eigen Face, which is still stored as a vector, is then normalized by dividing by the unit vector using the function normalizeEigenFaces() which uses cv::normalize() on each individual vector contained within the combined eigenface matrix. With our normalized Eigen Faces, Average Face, and Eigen Values we can then move to the testing phase of the project.

Part A:

The implementation of facial recognition using eigenfaces is implemented in Identify.cpp. Using the function identify() the actual facial recognition occurs. It relies on multiple helper functions. First, identify() establishes what eigenfaces to use based on the specified pca percentage. It then creates a new matrix that only includes those faces. Then the query faces are projected onto face space using the helper function projectEigenFace(). projectEigenFace() normalizes the new image by subtracting the average face from it, finding the dot product between the eigenface and normalized image, then computing the projected face by adding the eigenface to the normalized, and then finally adding back the average face. After the projection has been computed for both training and query images matches can be found. Each query is compared to every training face using the helper function distanceFaces() which returns the L2 norm or $\sqrt{\Sigma\,(f1 - f2)^2}$. This is the distance between two different faces in face space which is used for finding correct matches. These saved distances are sorted least to greatest. faceIdentified() decides whether the top results were actual matches by looking at image ids. Correct and incorrect matches are then recorded for the CNC curve.

Part B:

identifyThreshold() handles Part b and is almost identical to identify(). It handles projection, distance, and identification in the same way. However, instead of accepting top identified matches, it varies a threshold that decides whether the face distance between top matches is low enough to be a match and not an intruder. This threshold is varied and False Positives and True Positives are generated for an ROC curve.

Results and Discussion

Experiment A

A.I

In experiment A.I, fa_H was used for training and fb_H was used for testing. Figure 1 shows the average face. Figure 2 and figure 3 shows the eigenfaces corresponding to the 10 largest eigenvalues and the 10 smallest eigenvalues, respectively.

The results are expected, as the average face successfully shows a “normal” face. The top 10 largest eigenvalues are recognizable faces while the smallest 10 are not at all recognizable.

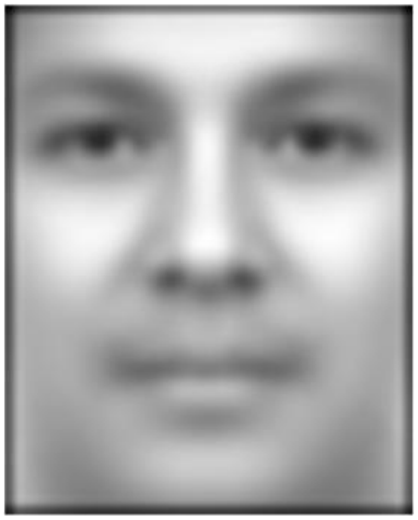
Figure 1: The average face.

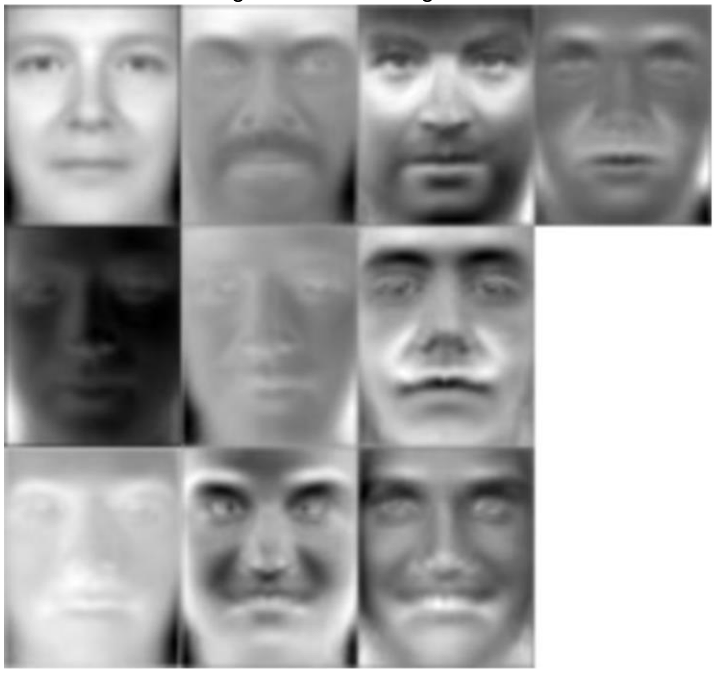
Figure 2: Eigenfaces of top 10 largest eigenvalues.

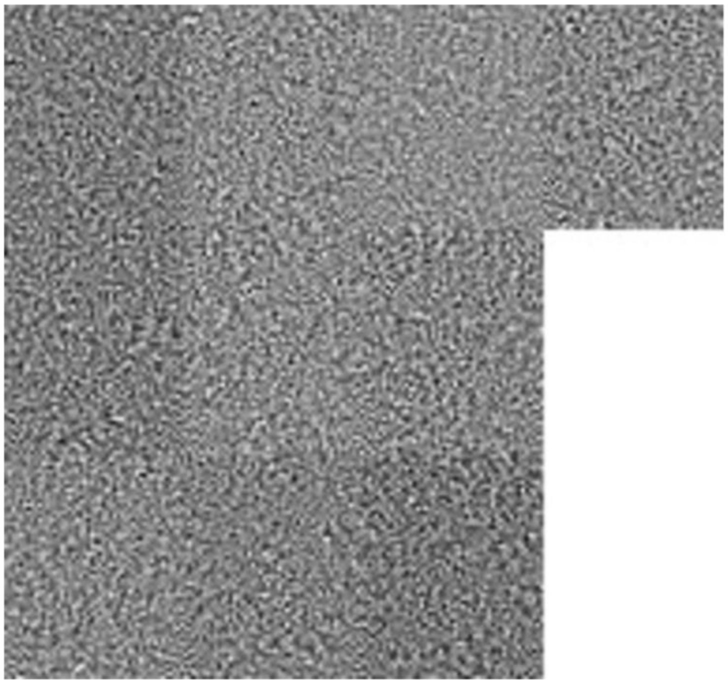
Figure 3: Eigenfaces of top 10 smallest eigenvalues.

A.II

For experiment A.II, we selected the top eigenfaces that preserved 80% of the information in the data. Then, we subtracted the average face to obtain the eigen-coefficients and projected both training and query images onto these eigenfaces as the basis. Next, the Mahalanobis distance between the eigen-coefficient vectors for each pair of training and query images was calculated. The top N face gallery images with the highest similarity score with the query face were considered a correct match. Figure 4 shows the Cumulative Match Characteristic) CMC curve for this experiment (80% preservation). The curve begins pretty low but with more eigenvectors, there is great performance.

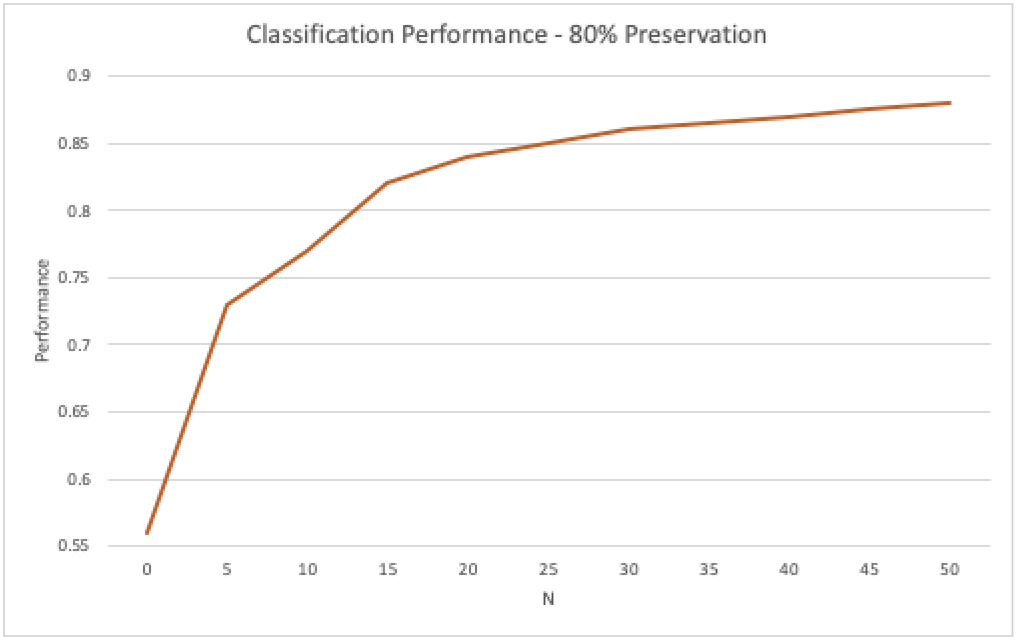

Figure 4: CMC curve for 80% preservation.

A.III

Figure 5 shows 3 query and the corresponding training images correctly matched assuming N=1. Despite the subtle differences, like smiling and not smiling, the classifier performs well on these faces.

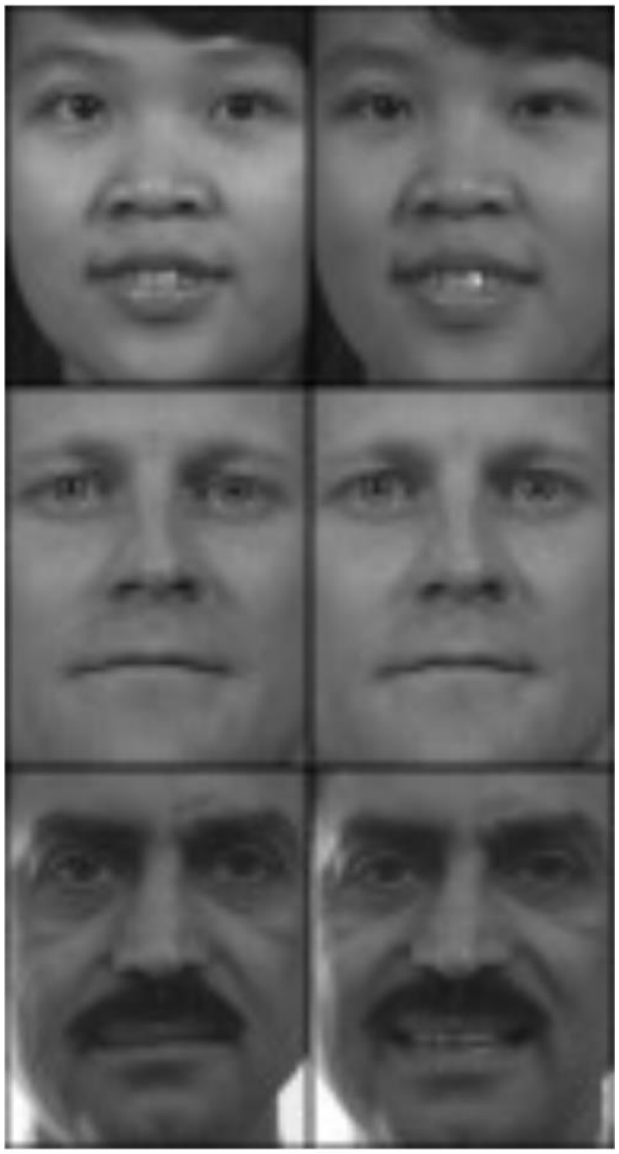

Figure 5: Correctly matched with top 80% of eigenvectors. Query image is on the right, training image is on the left.

A.IV

Figure 6 shows 3 query images and the corresponding training sample incorrectly matched assuming N=1. Those incorrectly matched have similar features but are absolutely not the same face. There is some obvious error in the 80% preservation of top eigenfaces, as men are classified as women and difference races are grouped together.

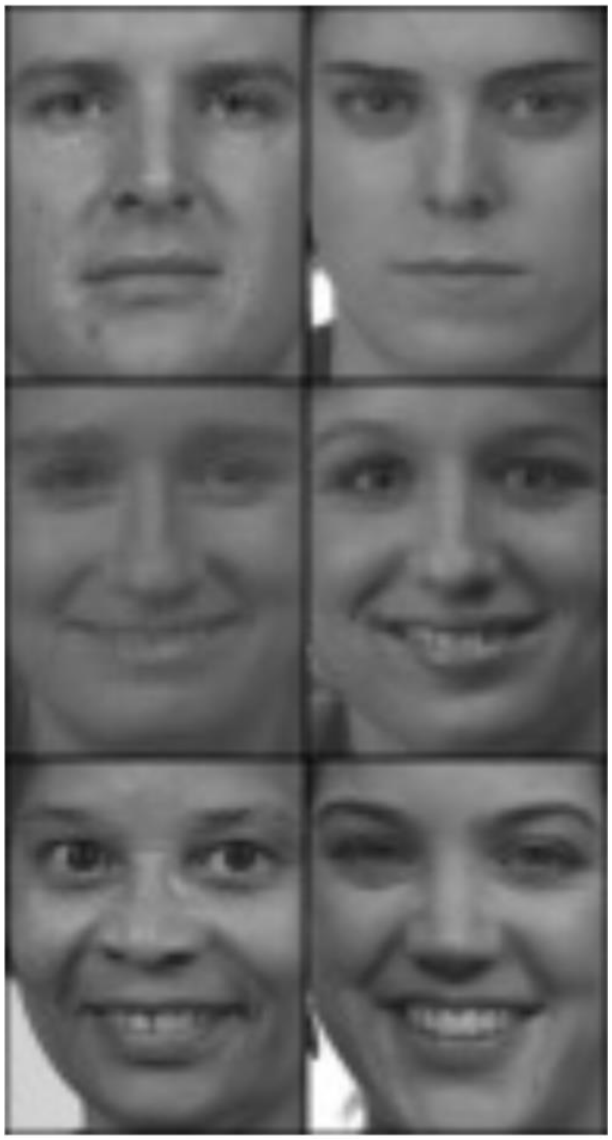
Figure 6: Incorrectly matched with top 80% of eigenvectors. Query image is on the right, training image is on the left.

A.V

For experiment A.V, we repeated A.II through A.IV when keeping the top eigenvectors corresponding to 90% and 95% of the information in the data.

90% Top Eigenvectors

Figure 7 shows the CMC curve for 90% performance. The curve shows better initial performance than 80% preservation, but then peters out near the same performance with higher N value. Figure 8 shows 3 query images and the corresponding training image correctly matched on the right and the 3 query images and the corresponding training image incorrectly matched on the left, both assuming N=1. The 3 correctly matched does not change from 80% preservation, but those incorrectly matched still classifies men as women and different races together as the same. The features of the incorrectly matched faces are admittedly similar.

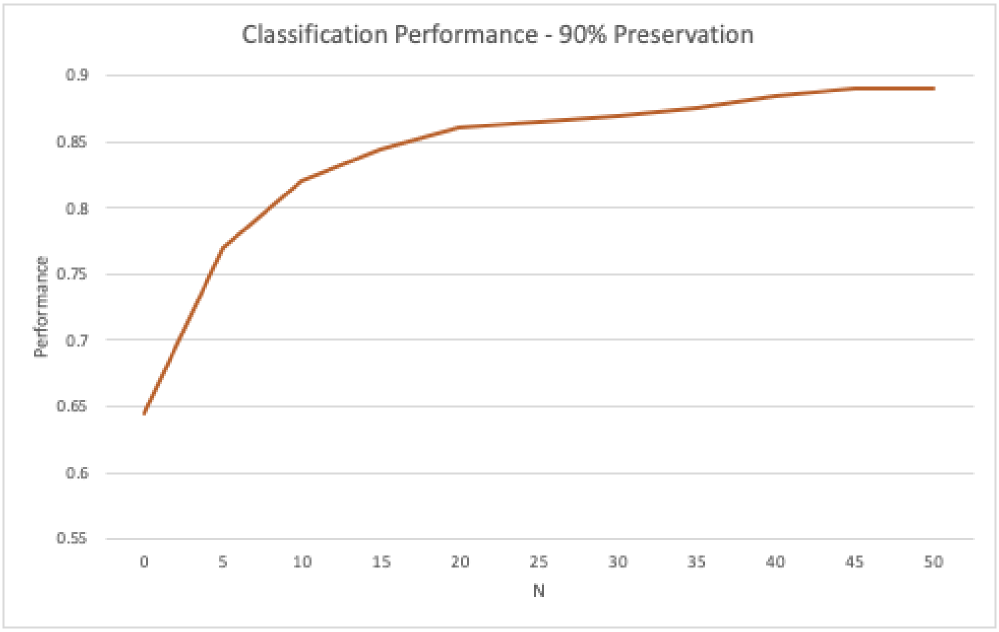

Figure 7: CMC curve for 90% preservation.

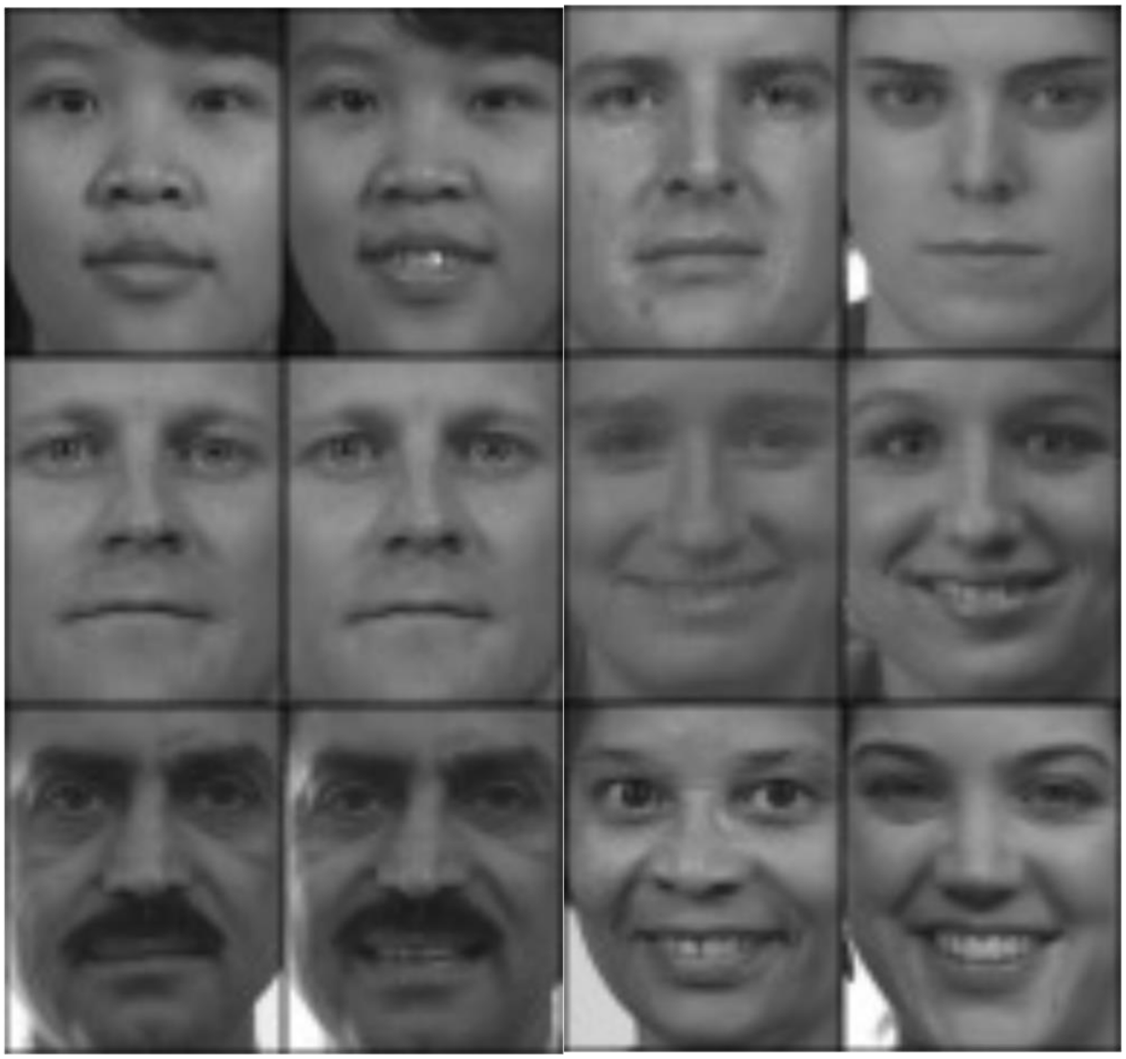

Figure 8: Correctly and incorrectly matched with top 90% of eigenvectors. Query image is on the right and training is on the left in each picture.

95% Top Eigenvectors

Figure 9 shows the CMC curve for 95% performance. The curve starts even better than 90% preservation and performs better faster than both, but again reaches near the same with higher N value. Figure 10 shows 3 query images and the corresponding training image correctly matched on the right and the 3 query images and the corresponding training image incorrectly matched on the left, both assuming N=1. Again, the 3 correctly matched are the same as 80% and 90% preservation. Those incorrectly matched are a bit different from 90% preservation, matching those with even more similar features than before. It can be confirmed that with more preservation, the better thee classifier performs in matching features.

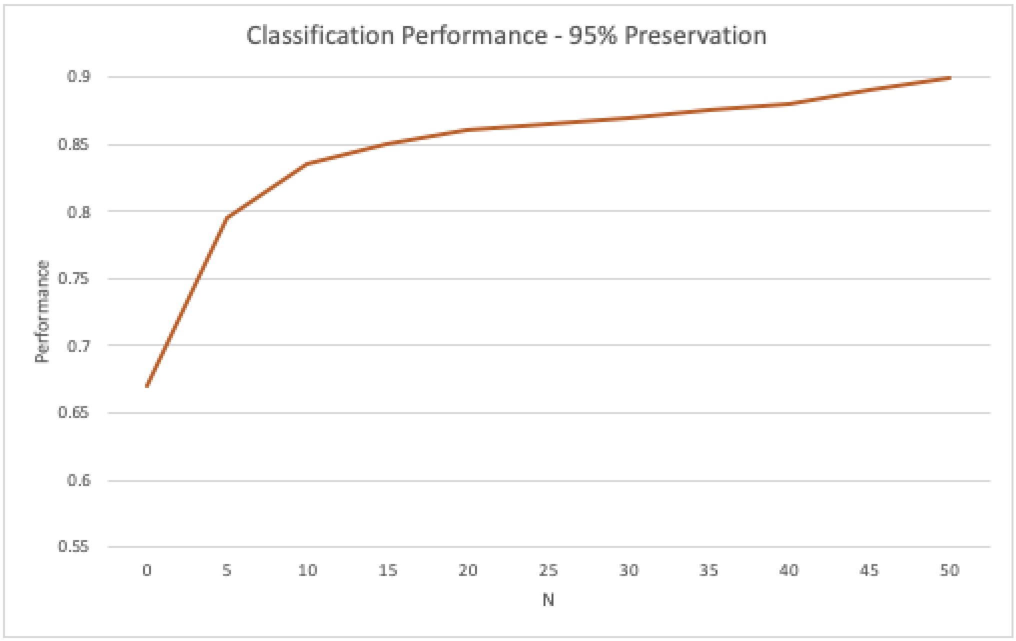

Figure 9: CMC curve for 95% preservation.

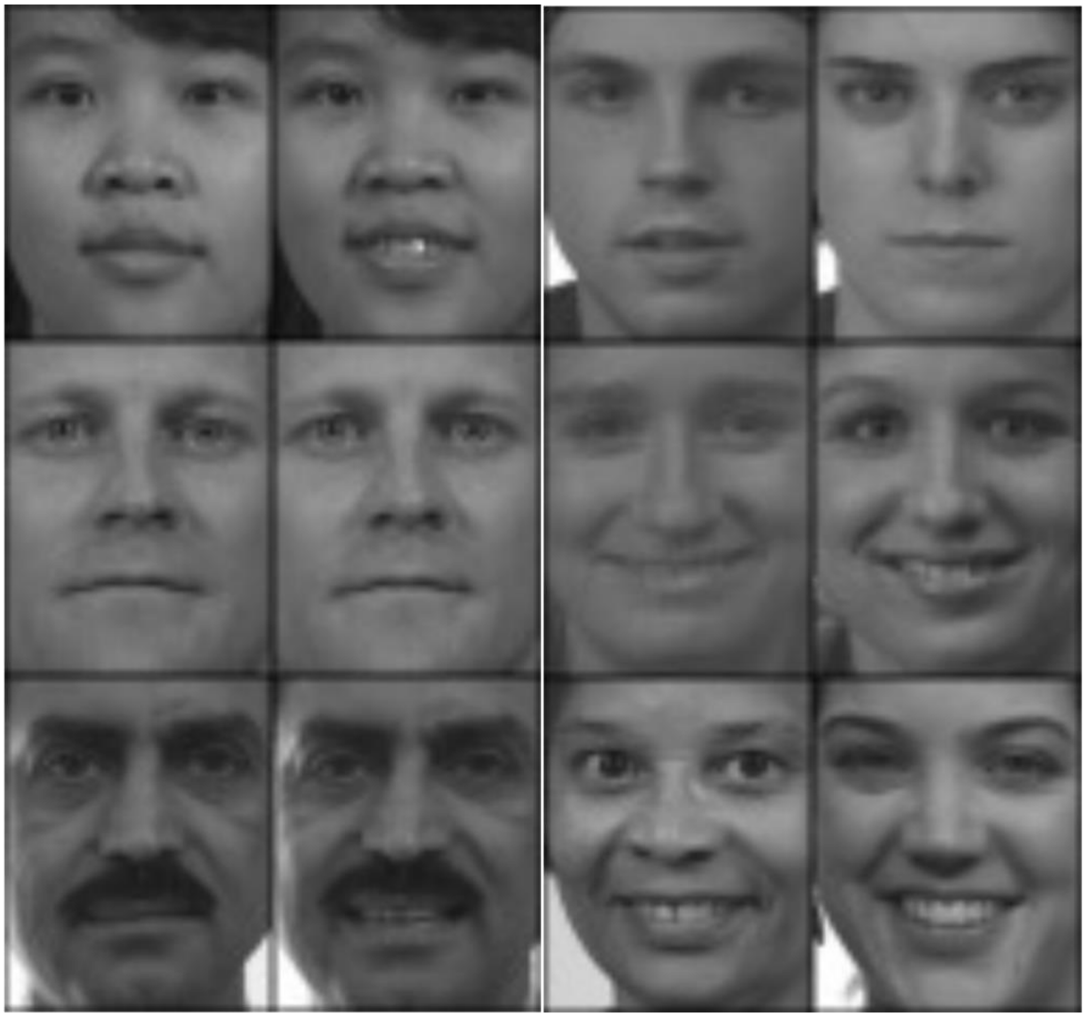
Figure 10: Correctly and incorrectly matched with top 95% of eigenvectors. Query image is on the right and training is on the left in each picture.

Experiment B

In Experiment B, we first removed all the images of the first 50 subjects from fa_H, referring to the reduced set as fa2_H. We then performed recognition using fa2_H for training and fb_H for testing. The eigenvectors corresponding to 95% of the information were used to classify as either intruders or non-intruders. Figure 11 shows the ROC graph for this experiment. Looking at the ROC curve, you can see that as the threshold is increased, the true positive rate increases substantially between 0.2 and 0.7, until it begins to even out at 0.8. The false positive rate evens out at 0.3.

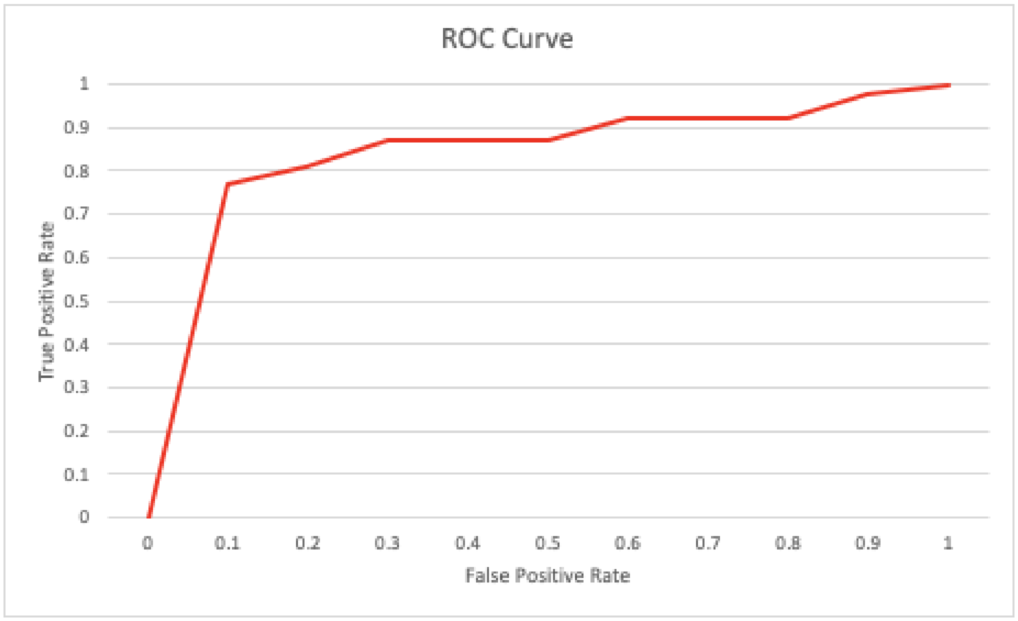

Figure 11: ROC curve generated by varying classification threshold.

Program Listings

Driver.cpp

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <fstream>
#include <sstream>
#include <stdio.h>
#include "EigenFaces.h"
#include "Identify.h"

using namespace std;
using namespace cv;

float PCA_PERCENTAGE    = .80;

int main(int argc, char* argv[])
{
    vector<pair<string, Mat1f> > trainingFaces, queryFaces;
    Mat1f eigenFaces, eigenValues, averageFace;

    if(argc < 2)
    {
        cout << "Need PCA percentage! aborting" << endl;
        return 1;
    }

    PCA_PERCENTAGE = atof(argv[1]);

    // Part A

    // Enter path as "../DirectoryPath/*.pgm
    readFaces("../Faces_FA_FB/fa_H/*.pgm", trainingFaces);
    readFaces("../Faces_FA_FB/fb_H/*.pgm", queryFaces);


    cout << "Reading" << endl;
    if(!savedFacesExist(averageFace, eigenFaces, eigenValues, "../Faces_FA_FB/fa_H")) // faces haven't
been computed yet
    {
        cout << "No saved faces. Computing..." << endl;

        computeEigenFaces(trainingFaces, averageFace, eigenFaces, eigenValues, "../Faces_FA_FB/fa_H");
    }
    normalizeEigenFaces(eigenFaces, trainingFaces.size());
    cout << "Done." << endl;

    // Write average face to file
    writeEigenFace(averageFace, "averageFace.pgm");
    identify("../N-Results/NData", averageFace, eigenFaces, eigenValues, trainingFaces, queryFaces);


    // Print top 10 eigenvalues
    char faceFileName[150];
    for(int i = 0; i < 10; i++)
    {
        sprintf(faceFileName, "Part_A_largestFace%i.pgm", i + 1);
        writeEigenFace(eigenFaces.col(i), faceFileName);
    }

    // Print top 10 eigenvalues
    for(int i = eigenFaces.cols - 1; i > eigenFaces.cols - 1 - 10; i--)
    {
        sprintf(faceFileName, "Part_A_smallestFace%i.pgm", i - eigenFaces.cols + 2);
        writeEigenFace(eigenFaces.col(i), faceFileName);
    }

    // Part B

    trainingFaces.clear();
    queryFaces.clear();

    readFaces("../Faces_FA_FB/fa2_H/*.pgm", trainingFaces);
    readFaces("../Faces_FA_FB/fb_H/*.pgm", queryFaces);

    cout << "Reading Part B" << endl;
    if(!savedFacesExist(averageFace, eigenFaces, eigenValues, "fa2_H")) // faces haven't been computed
yet
    {
        cout << "No saved faces. Computing..." << endl;

        computeEigenFaces(trainingFaces, averageFace, eigenFaces, eigenValues, "fa2_H");
    }

    normalizeEigenFaces(eigenFaces, trainingFaces.size());

    writeEigenFace(averageFace, "averageFace_PartB.pgm");


    PCA_PERCENTAGE = .95;

    identifyThreshold("../B_Results/BData", averageFace, eigenFaces, eigenValues, trainingFaces,
queryFaces);

    return 0;
}
```

EigenFaces.h

```
#ifndef EIGENRECOGNITION_EIGENFACES_H
#define EIGENRECOGNITION_EIGENFACES_H

#include <iostream>
#include <Eigen/Dense>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <fstream>
#include <sstream>
#include <stdio.h>
#include "opencv2/core.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"


bool savedFacesExist(cv::Mat1f &averageFace, cv::Mat1f &eigenFaces, cv::Mat1f &eigenValues,
                  const char *path);

void readFaces(const char *path, std::vector<std::pair<std::string, cv::Mat1f> > &faces);

void writeEigenFace(cv::Mat1f theImage, char *fileName);

void normalizeEigenFaces(cv::Mat1f &eigenfaces, int numberOfEigenFaces);

void computeEigenFaces(std::vector<std::pair<std::string, cv::Mat1f> > trainingFaces, cv::Mat1f
&averageFace,
                   cv::Mat1f &eigenFaces, cv::Mat1f &eigenValues, const char *path);

#endif //EIGENRECOGNITION_EIGENFACES_H
```

EigenFaces.cpp

```
#include "EigenFaces.h"

// Checks if files containing eigen face data, eigen value data, and average face
// are available/created for reading. If they are available returns true and reads them
// else returns false.
bool savedFacesExist(cv::Mat1f &averageFace, cv::Mat1f &eigenFaces, cv::Mat1f &eigenValues,
        const char *path)
{
    char fileName[50];
    sprintf(fileName, "%s/eigen_faces.yml", path);
    cv::FileStorage fsEigenFaces(fileName, cv::FileStorage::READ);

    if(fsEigenFaces.isOpened()) {
        fsEigenFaces["Eigen Faces"] >> eigenFaces;
    }
    else {
        return false;
    }
    sprintf(fileName, "%s/eigen_values.yml", path);
    cv::FileStorage fsEigenValues(fileName, cv::FileStorage::READ);

    if(fsEigenFaces.isOpened()) {
        fsEigenFaces["Eigen Values"] >> eigenFaces;
    }
    else {
        return false;
    }

    sprintf(fileName, "%s/average_face.yml", path);
    cv::FileStorage fsAverageFace(fileName, cv::FileStorage::READ);

    if(fsEigenFaces.isOpened()) {
        fsEigenFaces["Average Face"] >> eigenFaces;
    }
    else {
        return false;
    }
    return true;
}

// Reads in images from directory specified by path variable into vector that
// includes the filename in a pair with the image
void readFaces(const char *path, std::vector<std::pair<std::string, cv::Mat1f> > &faces)
{
    std::vector<cv::String> fileNames;
    cv::Mat1f temp;
    glob(path, fileNames, false);
    size_t count = fileNames.size(); //number of png files in images folder
    for (size_t i = 0; i < count; i++) {
        temp = imread(fileNames[i], cv::IMREAD_GRAYSCALE);
        temp.reshape(1,1);

        faces.push_back(std::pair<std::string, cv::Mat1f>(fileNames[i], temp));
    }
}

// Converts the face vector to an image normalizes and then writes it out to file
void writeEigenFace(cv::Mat1f theImage, char *fileName)
{
    cv::Mat1f temp = theImage;
    temp.reshape(1,48);
    cv::normalize(theImage, temp, 255, 0, cv::NORM_MINMAX, -1);
    cv::imwrite(fileName, temp);
}

// Normalizes every eigenface using cv::normalize
void normalizeEigenFaces(cv::Mat1f &eigenfaces, int numberOfEigenFaces)
{
    cv::Mat1f subImage;
    for(size_t i = 0; i < numberOfEigenFaces; i++)
    {
        subImage = eigenfaces.col(i);
        // normalizes the values by dividing by the unit vector
        normalize(subImage, subImage, 1, 0, cv::NORM_L1, -1 );
    }
}


// Computes eigenfaces, average face and eigen values for the
void computeEigenFaces(std::vector<std::pair<std::string, cv::Mat1f> > trainingFaces, cv::Mat1f
&averageFace,
        cv::Mat1f &eigenFaces, cv::Mat1f &eigenValues, const char *path)
{
    char fileName[150];
    // Initializes the Average Face Mat to size of training data with zeros i to j
    averageFace = cv::Mat::zeros(trainingFaces[0].second.size(), trainingFaces[0].second.type());
    // Creates an average of all faces by adding respective matrices together and then
    // dividing them by the amount of faces
    for(size_t i = 0; i < trainingFaces.size(); i++)
    {
        add(averageFace, trainingFaces[0].second, averageFace);
    }
    averageFace /= trainingFaces.size();

    // Writes out Average Face data to yml file
    sprintf(fileName, "%s/average_face.yml", path);
    cv::FileStorage fsAvg(fileName, cv::FileStorage::WRITE);
    fsAvg << "Average Face" << averageFace;

    // Subtracts average face from the training face and then
    // concats all face images column wise to create Matrix A for SVD
    cv::Mat1f A(trainingFaces[0].second.rows * trainingFaces[0].second.cols, trainingFaces.size(),
            trainingFaces[0].second.type());
    for(size_t i = 0; i < trainingFaces.size(); i++)
    {
        cv::hconcat(A, trainingFaces[i].second.t() - averageFace, A);
    }

    // Performs matrix multiplication A transpose by A to get decomposed matrix and then
    // computes eigenvalues and eigenVectors
    cv::Mat1f eigenVectors;
    cv::eigen(A.t()*A, eigenValues, eigenVectors);

    eigenFaces = A * eigenVectors;


    // Saves Eigen values to yml file
    sprintf(fileName, "%s/eigen_values.yml", path);
    cv::FileStorage fsEigenValues(fileName, cv::FileStorage::WRITE);
    fsEigenValues << "Eigen Values" << eigenValues;
    // Save Eigen Faces to yml file
    sprintf(fileName, "%s/eigen_faces.yml", path);
    cv::FileStorage fsEigenFaces(fileName, cv::FileStorage::WRITE);
    fsEigenFaces << "Eigen Faces" << eigenFaces;
}
```

Identify.h

```
#ifndef EIGENRECOGNITION_IDENTIFY_H
#define EIGENRECOGNITION_IDENTIFY_H
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <fstream>
#include <sstream>
#include <stdio.h>
#include "opencv2/core.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"

using namespace std;

bool Compare(pair<string, float> x, pair<string, float> y);

float distanceFaces(cv::Mat1f &originalFace, cv::Mat1f &newFace);

bool faceIdentified(vector<pair<string, float> > similarFaces, int N, string searchID);

cv::Mat1f projectEigenFace(cv::Mat1f &newFace, cv::Mat1f &averageFace, cv::Mat1f &eigenFaces);

void identify(const char *resultsPath, cv::Mat1f averageFace, cv::Mat1f eigenFaces, cv::Mat1f eigenValues,
            vector<pair<string, cv::Mat1f> > trainingFaces, vector<pair<string, cv::Mat1f> > queryFaces);

void identifyThreshold(const char *resultsPath, cv::Mat1f averageFace, cv::Mat1f eigenFaces, cv::Mat1f
eigenValues,
                    vector<pair<string, cv::Mat1f> > trainingFaces, vector<pair<string, cv::Mat1f> > queryFaces);

#endif //EIGENRECOGNITION_IDENTIFY_H
```

Identify.cpp

```
#include "Identify.h"

// Global vars
extern float PCA_PERCENTAGE;

// Returns bool if x is bigger than y
bool Compare(pair<string, float> x, pair<string, float> y)
{
    return x.second < y.second;
```

```
}

// Returns float of norm of original space subtracted by new face
float distanceFaces(cv::Mat1f &originalFace, cv::Mat1f &newFace)
{
    return cv::norm(originalFace, newFace, cv::NORM_L2);
}

// Returns bool if a face as same ID as the search ID
bool faceIdentified(vector<pair<string, float> > similarFaces, int N, string searchID)
{
    for(int i = 0; i < N; i++)
    {
        if(similarFaces[i].first == searchID)
        {
            return true;
        }
    }
    return false;
}

// Calculates face projection
cv::Mat1f projectEigenFace(cv::Mat1f &newFace, cv::Mat1f &averageFace, cv::Mat1f &eigenFaces)
{
    cv::Mat1f normalizedFace = newFace - averageFace;
    // Initializes the Projected Face Mat to size of training data with zeros i to j
    cv::Mat1f projectedFace = cv::Mat::zeros(averageFace.size(), averageFace.type());
    cv::Mat1f subImage;
    for(size_t i = 0; i < eigenFaces.cols; i++)
    {
        subImage = eigenFaces.col(i);
        float faceCoefficients = normalizedFace.dot(subImage.t());
        projectedFace += (faceCoefficients * subImage);
    }
    return projectedFace + averageFace;
}

// Runs classifier cycling from 1 <= N <= 50
// Prints correctly/incorrectly matched faces and CMC curve data
void identify(const char *resultsPath, cv::Mat1f averageFace, cv::Mat1f eigenFaces, cv::Mat1f eigenValues,
              vector<pair<string, cv::Mat1f> > trainingFaces, vector<pair<string, cv::Mat1f> > queryFaces)
{
    // Perform PCA dimensionality reduction
    float eigenValuesSum = cv::sum(eigenValues)[0];
    float currentEigenTotal = 0;
    int count;
    char fileName[100];

    // Find the number of vectors to preserve PCA_PERCENTAGE of the information
    for(count = 0; currentEigenTotal / eigenValuesSum < PCA_PERCENTAGE && count < eigenValues.rows;
count++)
```

```
{
    // y and x are swapped in cv.
    currentEigenTotal += eigenValues.at<uchar>(0, count);
}

cout << "Reducing Dimensionality from " << eigenFaces.cols << " to " << count << "!" << endl;

cv::Mat1f reducedEigenFaces(eigenFaces, cv::Range::all(), cv::Range(0,count));

// Project the faces onto the reduced eigenFaces
vector<pair<string, cv::Mat1f> > projectedTrainingFaces, projectedQueryFaces;
for(unsigned int i = 0; i < trainingFaces.size(); i++)
{
    pair<string, cv::Mat1f> temp(trainingFaces[i].first,
                          projectEigenFace(trainingFaces[i].second, averageFace, reducedEigenFaces));
    projectedTrainingFaces.push_back(temp);
}
for(unsigned int i = 0; i < queryFaces.size(); i++)
{
    pair<string, cv::Mat1f> temp(queryFaces[i].first,
                          projectEigenFace(queryFaces[i].second, averageFace, reducedEigenFaces));
    projectedQueryFaces.push_back(temp);
}
cout << "Faces projected" << endl;

cv::Mat1f projQueryFace;
int correct, incorrect;
correct = incorrect = 0;
bool querySaved = false;

ofstream output;
vector<float> N_Performances(50, 0);
sprintf(fileName, "%s_%i_NImageNames.txt", resultsPath, (int)(PCA_PERCENTAGE*100));

// Open file
output.open(fileName);

// Iterate through each query face and see if it can be classified correctly
for(unsigned int i = 0; i < queryFaces.size(); i++)
{
    projQueryFace = projectedQueryFaces[i].second;
    vector< pair<string, float> > queryPairs;
    querySaved = false;

    // Find the distances from this query face to every training face
    for(unsigned int t = 0; t < trainingFaces.size(); t++)
    {
        pair<string, float> newPair(trainingFaces[t].first,
                              distanceFaces(projQueryFace, projectedTrainingFaces[t].second));
        queryPairs.push_back(newPair);
    }
```

```
        // Sort the distances from least to greatest
        sort(queryPairs.begin(), queryPairs.end(), Compare);

        // Iterate from n = 1 to 50
        for(int n = 0; n < 50; n++)
        {
            if(faceIdentified(queryPairs, n + 1, projectedQueryFaces[i].first))
            {
                N_Performances[n] += 1;
                // Only save a correct match if N = 1 (0 in this case since we start at 0)
                if(correct < 3 && !querySaved && n == 0)
                {
                    output << "Correct Query Image " << correct << " ID: " << queryFaces[i].first;
                    output << " Correct Training Image " << correct << " ID: " << queryPairs[0].first;
                    output << endl << endl;
                    correct++;
                    querySaved = true;
                }
            }
            else
            {
                // Only save an incorrect match if N = 1 (0 in this case since we start at 0)
                if(incorrect < 3 && !querySaved && n == 0)
                {
                    output << "Incorrect Query Img " << incorrect << " ID: " << queryFaces[i].first;
                    output << " Incorrect Training Img " << incorrect << " ID: " << queryPairs[0].first;
                    output << endl << endl;
                    incorrect++;
                    querySaved = true;
                }
            }
        }
    }

    // Close file
    output.close();

    // Open next file
    sprintf(fileName, "%s_%i.txt", resultsPath, (int)(PCA_PERCENTAGE*100));
    output.open(fileName);

    // Print out the data for the CMC curve
    for(int n = 0; n < 50; n++)
    {
        output << n+1 << "\t" << (N_Performances[n] / (float)queryFaces.size()) << endl;
    }

    // Close file
    output.close();
}
```

```
// Runs threshold classifier!
// Varies the threshold to determine if a face can fit
// Prints the results to a series of files in directory and prints data for ROC curve for report.
void identifyThreshold(const char *resultsPath, cv::Mat1f averageFace, cv::Mat1f eigenFaces, cv::Mat1f
eigenValues,
                    vector<pair<string, cv::Mat1f> > trainingFaces, vector<pair<string, cv::Mat1f> > queryFaces)
{
    // Perform PCA dimensionality reduction
    float eigenValuesSum = cv::sum(eigenValues)[0];
    float currentEigenTotal = 0;
    int count;
    char fileName[100];

    // Find the number of vectors to preserve PCA_PERCENTAGE of the information
    for(count = 0; currentEigenTotal / eigenValuesSum < PCA_PERCENTAGE && count < eigenValues.rows;
count++)
    {
        // y and x are swapped in cv.
        currentEigenTotal += eigenValues.at<uchar>(0, count);
    }

    cout << "Reducing Dimensionality from " << eigenFaces.cols << " to " << count << "!" << endl;

    cv::Mat1f reducedEigenFaces(eigenFaces, cv::Range::all(), cv::Range(0,count));

    // Project the faces onto the reduced eigenFaces
    vector<pair<string, cv::Mat1f> > projectedTrainingFaces, projectedQueryFaces;
    for(unsigned int i = 0; i < trainingFaces.size(); i++)
    {
        pair<string, cv::Mat1f> temp(trainingFaces[i].first,
                        projectEigenFace(trainingFaces[i].second, averageFace, reducedEigenFaces));
        projectedTrainingFaces.push_back(temp);
    }
    for(unsigned int i = 0; i < queryFaces.size(); i++)
    {
        pair<string, cv::Mat1f> temp(queryFaces[i].first,
                        projectEigenFace(queryFaces[i].second, averageFace, reducedEigenFaces));
        projectedQueryFaces.push_back(temp);
    }
    cout << "Faces projected" << endl;

    cv::Mat1f projQueryFace;
    int TruePositiveCount, FalsePositiveCount;
    TruePositiveCount = FalsePositiveCount = 0;
    pair<int, int> temp(0,0);
    vector< pair<int, int> > counts(1800, temp);

    // Iterate through each query face and see if it can be classified correctly
    for(unsigned int i = 0; i < projectedQueryFaces.size(); i++)
    {
        cout << "\rQuery Face: " << i;
        projQueryFace = projectedQueryFaces[i].second;
        vector< pair<string, float> > queryPairs;

        for(unsigned int t = 0; t < trainingFaces.size(); t++)
        {
            pair<string, float> newPair(trainingFaces[t].first, distanceFaces(projQueryFace,
trainingFaces[t].second));
            queryPairs.push_back(newPair);
        }

        sort(queryPairs.begin(), queryPairs.end(), Compare);
        cout << "\t" << queryPairs[0].second << endl;

        for(int threshold = 380; threshold < 1500; threshold+=2)
        {
            // Best match is less than the threshold
            if(queryPairs[0].second <= threshold)
            {
                // Check if true or false positive
                if(atoi(projectedQueryFaces[i].first.c_str()) <= 93)
                {
                    counts[threshold].first++;
                }
                else
                {
                    counts[threshold].second++;
                }
            }
        }
    }

    // Output info
    sprintf(fileName, "%s_%i.txt", resultsPath, (int)(PCA_PERCENTAGE*100));
    ofstream output;

    // Open file
    output.open(fileName);

    // Print ROC Curve info
    for(int threshold = 50; threshold < 600; threshold+=2)
    {
        float TruePositiveRate = counts[threshold].first / (float)trainingFaces.size();
        float FalsePositiveRate = counts[threshold].second / (float)(queryFaces.size() - trainingFaces.size());

        output << threshold << "\t" << TruePositiveRate<< "\t" << FalsePositiveRate << endl;
    }

    // Close file
    output.close();
}
```