Eigenface Approach for Recognition

CS 479 Assignment 3

Ethan Brown, Ethan Park

Date Due: April 14, 2020

Date Handed In: April 14, 2020

Division of Work:

Ethan Brown and Ethan Park worked together on the programming side of the assignment. For the report, Ethan Park worked on the Theory, Implementation, and Source Code, and Ethan Brown worked on the Results and Discussion.

Table of Contents

## Theory................................................................................................ 3

## Implementation.................................................................................. 4

## Results and discussion....................................................................... 5

### Part A......................................................................................... 5

### Part B......................................................................................... 9

### Part C......................................................................................... 10

### Part D......................................................................................... 15

### Part E......................................................................................... 16

## Source Code....................................................................................... 17

**Theory**

For this project we will be implementing the eigenface approach as discussed in Matthew Turk and Alex Pentland’s “Eigenfaces for Recognition” to recognize and classify faces given a dataset of training and test images of faces from the FERET face database using two datasets that differ in image resolution. To achieve this solution Principal Component Analysis (PCA) is used to reduce dimensionality of images by finding the most needed to retain a certain percentage (in our case eighty, ninety, and ninety-five percent) of the information given by the original data.

To apply PCA to facial recognition we must first obtain a dataset of face images all centered and of the same size (which is provided by the FERET face database). Then, for all images M, we must then represent each image as an matrix as an vector and form the matrix . We then compute the average face vector with the following equation

We can then use the mean face to center each face around zero with the following equation

Thebest eigenvectors (**u**) and thebest eigenvalues () can be found by computing using where . Eigenvectors should be normalized such that .

Only keep theeigenvectors that correspond to thelargest eigenvalues.

To apply eigenfaces for recognition we first find thebest eigenvectors using PCA that represent a certain percentage () of the data

Given an unknown image we first normalize the image

We then project the unknown face in eigenspace

We then compute the distance

Given some threshold, if , thenis a face. If it is a face, we then compute the distance from the projected to a project tion of each training face. We classify based on the smallest distance.

**Implementation**

As in the previous project assignment, the processing image files, Image, ReadImage, ReadImageHeader, and WriteImage, provided Dr. Bebis were used to read and write images for the project. The training and test faces are stored in trainingFaces and testFaces respectively as vector<pair<string, VectorXf>> in which the types of the pair represent the face ID and the face image respectively. The training and test faces are read in using the readFaces function which takes in a string for the name of the file directory that contains the images and data structures that will hold the eigenfaces (i.e. trainingFaces and testFaces).

The function computeEigenFaces is used to compute the eigenfaces and takes in the training faces to compute the average face, eigenfaces, and eigenvalues (all passed by reference). It also takes in the name of the directory (as a string) that was used to generate the training faces for output file naming purposes. To reduce runtime, we also created the function readSavedFaces so the eigenfaces would not need to be computed if the necessary files already existed. We normalize the eigenfaces using the normalizeEigenFaces function which takes in the eigenfaces to be normalized. writeFace is used to write the average face and store it as a pgm file. It takes the face to be written and the name of the generated image.

The classifier is run using the runClassifier function which takes in the name of the directory the results will be stored in, the average face, the eigenfaces, the eigenvalues, the training faces, the testfaces, and the PCA threshold. The classifier threshold is implemented in the function classifierThreshold which, similar to runClassifier, takes in the name of the directory the results will be stored in, the average face, the eigenfaces, the eigenvalues, the training faces, the testfaces, and the PCA threshold.

**Results and Discussion**

Part A

Part A.I

The average face can be seen in Figure 1. The eigenfaces corresponding to the 10 largest eigenvalues can be seen in Figure 2. These eigenfaces maintain most of the information, so the images resemble faces. The eigenfaces corresponding to the 10 smallest eigenvalues can be seen in Figure 3. These eigenfaces do not hold much information, and as a result, these images do not look like faces, but rather just pixelated images.



**Figure 1** shows the average face generated in Part A.I



**Figure 2** shows the eigenfaces corresponding to the 10 largest eigenvalues in Part A.I. Because these eigenfaces contain a large amount of information the images closely resemble real faces.



**Figure 3** shows the eigenfaces corresponding to the 10 smallest eigenvalues in Part A.I. Since these eigenfaces do not contain much information, the corresponding images do not resemble faces.

Part A.II

Using the eigenfaces that preserve 80% of the information in the data and varying N from 1-50, we resulted with a CMC graph that can be seen in Figure 6 along with the results from Part.A.V.

Part A.III

Assuming N = 1, 3 query images which were correctly matched can be seen in Figure 4 alongside the corresponding training samples.







**Figure 4** shows 3 query images that were correctly matched in Part A.III alongside the corresponding training samples.

Part A.IV

Assuming N = 1, 3 query images which were incorrectly matched can be seen in Figure 5 alongside the corresponding training samples.



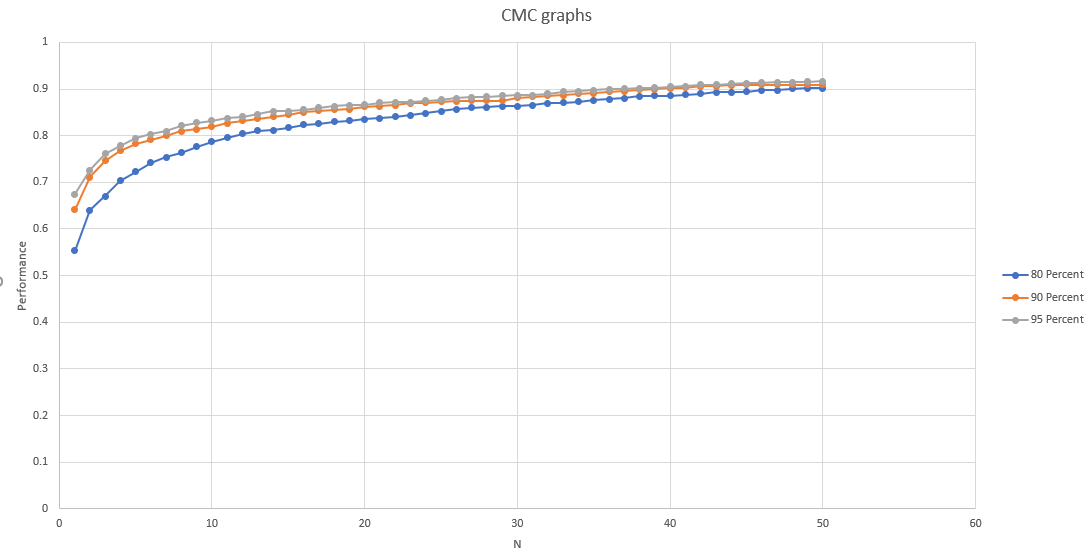




**Figure 5** shows 3 query images that were incorrectly matched in Part A.IV alongside the corresponding training samples.

Part A.V

We repeated Part A.II-A.IV using the top eigenvectors corresponding to 90% and 95% of the information in the data. The CMC curves can be seen in Figure 6 alongside the curve from Part A.II. Using the eigenvectors corresponding to 90% of the information and assuming N = 1, 3 correctly matched images and 3 incorrectly matched images alongside their corresponding training samples can be seen in Figure 7 and Figure 8 respectively. Using the eigenvectors corresponding to 95% of the information and assuming N = 1, 3 correctly matched images and 3 incorrectly matched images alongside their corresponding training samples can be seen in Figure 9 and Figure 10 respectively. There is a rather significant difference between A.II and A.V when the value of N is low, but as N increases, the differences get smaller and smaller. This can be seen in detail in the graph in Figure 6.



**Figure 6** shows the CMC curves from Part A. From the curves, you can see that as the amount of information preserved increases, the curve more quickly approaches 1 meaning better performance. It is also worth noting that the curves are furthest apart near N = 1 and as N increases, the curves tend to converge with the 95% curve being only marginally better than the other 2.







**Figure 7** shows 3 query images that were correctly matched while preserving 90% of the information in Part A.V alongside the corresponding training samples.







**Figure 8** shows 3 query images that were incorrectly matched while preserving 90% of the information in Part A.V alongside the corresponding training samples.







**Figure 9** shows 3 query images that were correctly matched while preserving 95% of the information in Part A.V alongside the corresponding training samples.



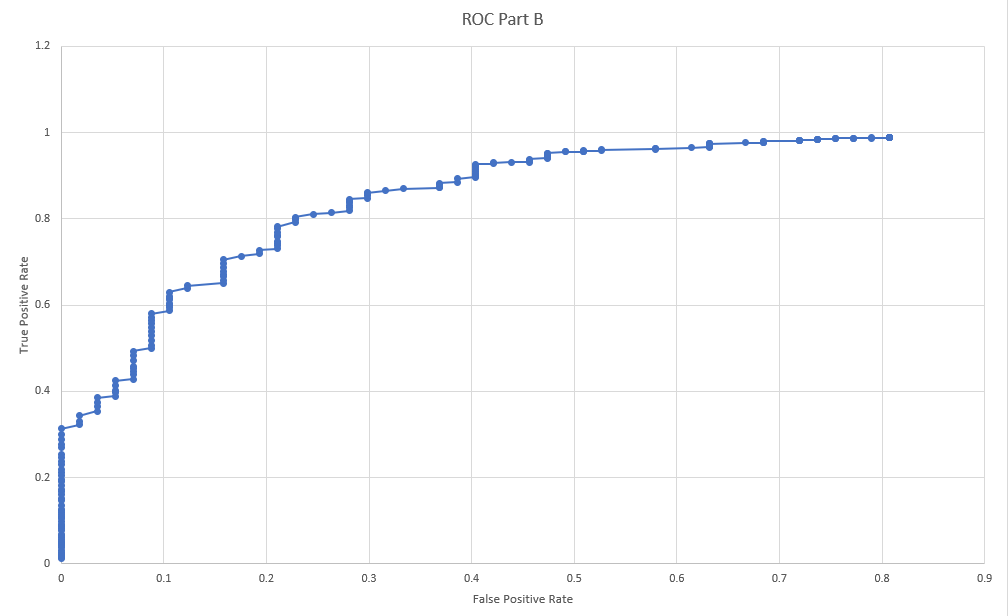




**Figure 10** shows 3 query images that were incorrectly matched while preserving 95% of the information in Part A.V alongside the corresponding training samples.

Part B

For this experiment, we removed the first 50 subjects from the training gallery and performed classification using the eigenvectors corresponding to 95% of the information to determine the number of true positives and false positives. We plotted the false positive rate against the true positive rate by varying the threshold in an ROC curve shown in Figure 11.



**Figure 11** shows the ROC curve for part B. Along the x axis is the false positive rate and along the y axis is the true positive rate at different thresholds. From the graph you can see that as the false positive rate increases, the true positive rate also increases. A higher threshold increases the number of true positives, but it also increases the number of false positives. Likewise, a lower threshold decreases the number of false positives, but also decreases the number of true positives.

Part C

Part C.I

Part C.I is the same as Part A.I, but we used low resolution images as opposed to high resolution images for training and testing. The new average face, the eigenfaces corresponding to the 10 largest eigenvalues, and the eigenfaces corresponding to the 10 smallest eigenvalues can be seen in Figure 12, Figure 13, and Figure 14 respectively.



**Figure 12** shows the average face generated in Part C.I. This average face is very similar the the average face from Part A.I it is just lower resolution.



**Figure 13** shows the eigenfaces corresponding to the 10 largest eigenvalues in Part C.I. Similarly to Part A.I, these images closely resemble real faces since the eigenfaces contain a lot of information. Just like the average face, these images are nearly identical to the ones in Part A.I shown in Figure 2 they are just lower resolution.



**Figure 14** shows the eigenfaces corresponding to the 10 smallest eigenvalues in Part C.I. Since these eigenfaces do not contain much information, the corresponding images do not resemble faces like in Part A.I.

Part C.II

Using the eigenfaces that preserve 80% of the information in the data and varying N from 1-50, we resulted with a CMC graph that can be seen in Figure 17 along with the results from Part C.V. Once again we used the low resolution images as opposed to the high resolution images in Part A.

Part C.III

Assuming N = 1, 3 query images which were correctly matched can be seen in Figure 15 alongside the corresponding training samples.







**Figure 15** shows 3 query images that were correctly matched in Part C.III alongside the corresponding training samples. These are the same subjects identified in Part A.III.

Part C.IV

Assuming N = 1, 3 query images which were incorrectly matched can be seen in Figure 16 alongside the corresponding training samples.



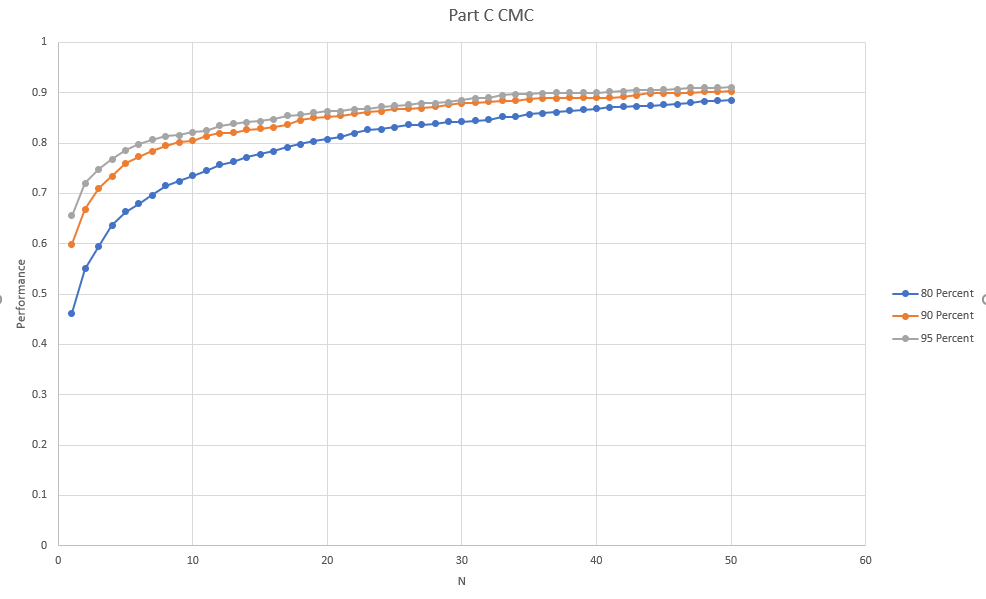




**Figure 16** shows 3 query images that were correctly matched in Part C.IV alongside the corresponding training samples. Once again, these are the same subjects as Part A.IV.

Part C.V

Part C.V is the same as Part A.V, but the low resolution images were used instead of the high resolution images. The CMC curves can be seen in Figure 17. Using the eigenvectors corresponding to 90% of the information and assuming N = 1, 3 correctly matched images and 3 incorrectly matched images alongside their corresponding training samples can be seen in Figure 18 and Figure 19 respectively. Using the eigenvectors corresponding to 95% of the information and assuming N = 1, 3 correctly matched images and 3 incorrectly matched images alongside their corresponding training samples can be seen in Figure 20 and Figure 21 respectively. Similarly to Part A, the difference between Part C.II and Part C.V is most significant for low values of N and as N increases, the differences get smaller and smaller.



**Figure 17** shows the CMC curves for Part C. These curves using the low resolution images are nearly identical to the curves from Part A that used the high resolution images. This shows that higher resolution in an image does not correspond to better classification accuracy. Just like The CMC curves from Part A, as the value of N increases, the differences between the curves corresponding to different amounts of information decreases.







**Figure 18** shows 3 query images that were correctly matched while preserving 90% of the information in Part C.V alongside the corresponding training samples.







**Figure 19** shows 3 query images that were incorrectly matched while preserving 90% of the information in Part C.V alongside the corresponding training samples.







**Figure 20** shows 3 query images that were correctly matched while preserving 95% of the information in Part C.V alongside the corresponding training samples. The subjects identified are the same subjects that were identified when preserving 90% of the information, so there was no change.



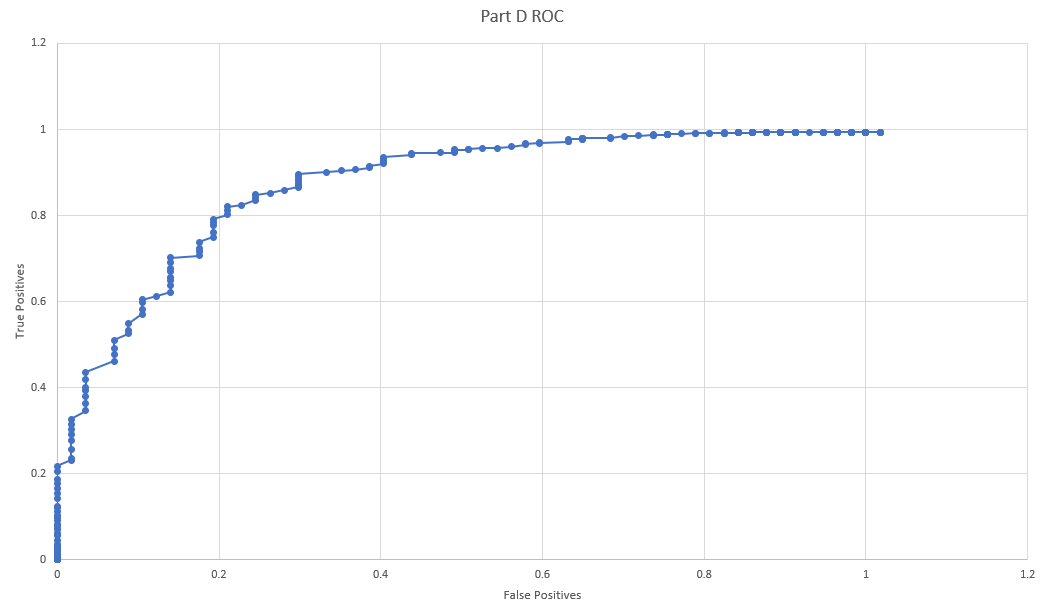




**Figure 21** shows 3 query images that were incorrectly matched while preserving 95% of the information in Part C.V alongside the corresponding training samples. The subjects identified are the same subjects that were identified when preserving 90% of the information, so there was no change.

Part D

Part D was the same as Part B, but we used the low resolution images instead of the high resolution images. Similarly to part B, we removed the first 50 subjects from the training gallery and performed classification using the eigenvectors corresponding to 95% of the information to determine the number of true positives and false positives. We plotted the false positive rate against the true positive rate by varying the threshold in an ROC curve shown in Figure 22.



**Figure 22** shows the ROC curve generated in Part D. This graph is nearly identical to that of Part B once again showing that the image resolution does not have a significant impact on classification performance. Just like in Part B, as the number of false positives increases, so does the number of true positives.

Part E

There are no significant differences in identification performance when using high resolution images as opposed to low resolution images. This can be seen in Figure 6 and Figure 17 which correspond to the CMC curves using high and low resolution images and in Figure 11 and Figure 22 which correspond to the ROC curves using high and low resolution images. The curves are very similar to one another with no significant differences. While there was no significant impact on identification performance, The low resolution images ran significantly faster than the high resolution images. This is likely because the high resolution images include more information that needs to be processed than the low resolution images.

**Source Code**

main.cpp

#include <iostream>

#include "Functions.cpp"

#include <vector>

#include <Eigen/Dense>

#include <dirent.h> // for reading in image directories

using namespace Eigen;

using namespace std;

int main()

{

// variables for training/test faces, eigenfaces, eigenvalues, and the average face

vector<pair<string, VectorXf>> trainingFaces, testFaces;

MatrixXf eigenfaces;

VectorXf eigenvalues;

VectorXf avgFace;

/\*\*\*Part A\*\*\*/

// read in training/test faces

readFaces((char\*)"./fa\_H", trainingFaces);

readFaces((char\*)"./fb\_H", testFaces);

// get eigenfaces, if they don't exist compute them

if(readSavedFaces(avgFace, eigenfaces, eigenvalues, "fa\_H") == false)

computeEigenFaces(trainingFaces, avgFace, eigenfaces, eigenvalues, "fa\_H");

// normalize the eigenfaces

normalizeEigenFaces(eigenfaces);

// create the image of the average face

writeFace(avgFace, (char\*)"averageFace.pgm");

// run the classfiler where PCA threshold = 80%, 90%, and 95%

runClassifier("N-Results/NData", avgFace, eigenfaces, eigenvalues, trainingFaces, testFaces, 0.80);

runClassifier("N-Results/NData", avgFace, eigenfaces, eigenvalues, trainingFaces, testFaces, 0.90);

runClassifier("N-Results/NData", avgFace, eigenfaces, eigenvalues, trainingFaces, testFaces, 0.95);

// output the eigenfaces corresponding to the 10 largest and smallest eigenvalues

char faceFileName[100];

for (int i = 0; i < 10; i++)

{

sprintf(faceFileName, "Part-AlargestFace%i.pgm", i+1);

writeFace(eigenfaces.col(i), faceFileName);

}

for(int i = eigenfaces.cols() -1; i > eigenfaces.cols() -11; i--)

{

sprintf(faceFileName, "Part-AsmallestFace%i.pgm", (int)(i - eigenfaces.cols() + 3));

writeFace(eigenfaces.col(i), faceFileName);

}

/\*\*\*Part B\*\*\*/

// clear training/test faces

trainingFaces.clear();

testFaces.clear();

// get eigenfaces, if they don't exist compute them

readFaces((char\*)"./fa2\_H", trainingFaces);

readFaces((char\*)"./fb\_H", testFaces);

// get eigenfaces, if they don't exist compute them

if(readSavedFaces(avgFace, eigenfaces, eigenvalues, "fa2\_H") == false)

computeEigenFaces(trainingFaces, avgFace, eigenfaces, eigenvalues, "fa2\_H");

// normalize the eigenfaces

normalizeEigenFaces(eigenfaces);

// create the image of the average face

writeFace(avgFace, (char\*)"averageFace-PartB.pgm");

// run the classifier threshold function

classifierThreshold("C-Results/CData", avgFace, eigenfaces, eigenvalues, trainingFaces, testFaces, 0.95);

/\*\*\*Part C\*\*\*/

// clear training/test faces

trainingFaces.clear();

testFaces.clear();

// read in training/test faces

readFaces((char\*)"./fa\_L", trainingFaces);

readFaces((char\*)"./fb\_L", testFaces);

// get eigenfaces, if they don't exist compute them

if(readSavedFaces(avgFace, eigenfaces, eigenvalues, "fa\_L") == false)

computeEigenFaces(trainingFaces, avgFace, eigenfaces, eigenvalues, "fa\_L");

// normalize the eigenfaces

normalizeEigenFaces(eigenfaces);

// create the image of the average face

writeFace(avgFace, (char\*)"averageFace-PartC.pgm");

// run the classfiler where PCA threshold = 80%, 90%, and 95%

runClassifier("PartC-Results/CData", avgFace, eigenfaces, eigenvalues, trainingFaces, testFaces, 0.80);

runClassifier("PartC-Results/CData", avgFace, eigenfaces, eigenvalues, trainingFaces, testFaces, 0.90);

runClassifier("PartC-Results/CData", avgFace, eigenfaces, eigenvalues, trainingFaces, testFaces, 0.95);

// output the eigenfaces corresponding to the 10 largest and smallest eigenvalues

char faceFileName\_c[100];

for (int i = 0; i < 10; i++)

{

sprintf(faceFileName\_c, "PartC-largestFace%i.pgm", i+1);

writeFace(eigenfaces.col(i), faceFileName\_c);

}

for(int i = eigenfaces.cols() -1; i > eigenfaces.cols() -11; i--)

{

sprintf(faceFileName\_c, "PartC-smallestFace%i.pgm", (int)(i - eigenfaces.cols() + 3));

writeFace(eigenfaces.col(i), faceFileName\_c);

}

/\*\*\*Part D\*\*\*/

// clear training/test faces

trainingFaces.clear();

testFaces.clear();

// read in training/test faces

readFaces((char\*)"./fa2\_L", trainingFaces);

readFaces((char\*)"./fb\_L", testFaces);

// get eigenfaces, if they don't exist compute them

if(readSavedFaces(avgFace, eigenfaces, eigenvalues, "fa2\_L") == false)

computeEigenFaces(trainingFaces, avgFace, eigenfaces, eigenvalues, "fa2\_L");

// normalize the eigenfaces

normalizeEigenFaces(eigenfaces);

// create the image of the average face

writeFace(avgFace, (char\*)"averageFace-PartD.pgm");

// run the classifier threshold function

classifierThreshold("PartD-CResults/CData", avgFace, eigenfaces, eigenvalues, trainingFaces, testFaces, 0.95);

}

Functions.cpp

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <fstream>

#include <sstream>

#include <stdio.h>

//#include "jacobi.cpp"

#include "ReadImage.cpp"

#include "WriteImage.cpp"

#include "ReadImageHeader.cpp"

#include "image.h"

#include "image.cpp"

#include <vector>

#include <Eigen/Dense>

#include <dirent.h> // for reading in image directories

using namespace Eigen;

using namespace std;

// functions to read and write binary files using Eigen

namespace Eigen

{

template<class Matrix>

void write\_binary(const char\* filename, const Matrix& matrix)

{

std::ofstream out(filename, ios::out | ios::binary | ios::trunc);

typename Matrix::Index rows=matrix.rows(), cols=matrix.cols();

out.write((char\*) (&rows), sizeof(typename Matrix::Index));

out.write((char\*) (&cols), sizeof(typename Matrix::Index));

out.write((char\*) matrix.data(), rows\*cols\*sizeof(typename Matrix::Scalar));

out.close();

}

template<class Matrix>

void read\_binary(const char\* filename, Matrix& matrix)

{

std::ifstream in(filename, ios::in | std::ios::binary);

typename Matrix::Index rows = 0, cols = 0;

in.read((char\*) (&rows), sizeof(typename Matrix::Index));

in.read((char\*) (&cols), sizeof(typename Matrix::Index));

matrix.resize(rows, cols);

in.read((char\*) matrix.data(), rows\*cols\*sizeof(typename Matrix::Scalar));

in.close();

}

}

void readFaces(char \*filepath, vector<pair<string, VectorXf>> &faces)

{

DIR \*directory;

struct dirent \*ent;

// read in all images in the directory

directory = opendir(filepath);

while ((ent = readdir(directory)) != NULL)

{

if(ent->d\_name[0] != '.')

{

int rows, cols, levels;

bool type;

char name[100] = "";

//read in each individual image

strcat(name, filepath);

strcat(name, "/");

strcat(name, ent->d\_name);

readImageHeader(name, rows, cols, levels, type);

ImageType currentImage(rows, cols, levels);

readImage(name, currentImage);

VectorXf currentFace = VectorXf(rows\*cols);

for(int i = 0; i < rows; i++)

{

for(int j = 0; j < cols; j++)

{

int temp = 0;

currentImage.getPixelVal(i, j, temp);

currentFace[i\*cols + j] = temp;

}

}

// store in the faces vector

faces.push\_back(pair<string, VectorXf>(string(ent->d\_name, 5), currentFace));

}

}

closedir(directory);

}

void writeFace(VectorXf face, char \*fileName)

{

// determine the number of rows/cols/levels of the image

int rows, cols;

if(face.rows() == 320)

{

rows = 20;

cols = 16;

}

if(face.rows() == 2880)

{

rows = 60;

cols = 48;

}

int levels = 255;

ImageType img(rows, cols, levels);

// find the min/max

float min = face.minCoeff();

float max = face.maxCoeff();

// determin the values of each pixel

for(int i = 0; i < rows; i++)

{

for(int j = 0; j < cols; j++)

{

float val = (face[i\*cols + j] - min) / (max - min);

img.setPixelVal(i, j, val\*255);

}

}

// write the image

writeImage(fileName, img);

}

void computeEigenFaces(vector<pair<string, VectorXf> > trainingFaces, VectorXf &averageFace, MatrixXf &eigenFaces, VectorXf &eigenValues, const char \*path)

{

char fileName[100];

EigenSolver<MatrixXf> solver;

MatrixXf A;

ofstream output;

MatrixXf eigenVectors;

// compute the average face

averageFace = VectorXf(trainingFaces[0].second.rows());

averageFace.fill(0);

for(auto it = trainingFaces.begin(); it != trainingFaces.end(); it++)

{

averageFace += (\*it).second;

}

averageFace /= trainingFaces.size();

// store to output file

sprintf(fileName, "%s-avg-binary.dat", path);

Eigen::write\_binary(fileName, averageFace);

A = MatrixXf(averageFace.rows(), trainingFaces.size());

for(vector<VectorXf>::size\_type i = 0; i < trainingFaces.size(); i++)

{

A.col(i) = trainingFaces[i].second - averageFace;

}

eigenVectors = MatrixXf(trainingFaces.size(), trainingFaces.size());

eigenVectors = A.transpose()\*A;

solver.compute(eigenVectors, true);

// compute the eigenfaces

eigenFaces = MatrixXf(averageFace.rows(), trainingFaces.size());

eigenFaces = A \* solver.eigenvectors().real();

// compute the eigenvalues

eigenValues = VectorXf(eigenFaces.cols());

eigenValues = solver.eigenvalues().real();

// store to output files

sprintf(fileName, "%s-binary.dat", path);

Eigen::write\_binary(fileName, eigenFaces);

Eigen::read\_binary(fileName, eigenFaces);

sprintf(fileName, "%s-values-binary.dat", path);

Eigen::write\_binary(fileName, eigenValues);

sprintf(fileName, "%s-EigenVectors.txt", path);

output.open(fileName);

output << eigenFaces;

output.close();

}

// function to normalize eigenfaces

void normalizeEigenFaces(MatrixXf &eigenfaces)

{

for(int i = 0; i < eigenfaces.cols(); i++)

{

eigenfaces.col(i).normalize();

}

}

bool readSavedFaces(VectorXf &averageFace, MatrixXf &eigenfaces, VectorXf &eigenvalues, const char \*filepath)

{

char filename[100];

ifstream inputStream;

// check if eigenfaces binary file exists

sprintf(filename, "%s-binary.dat", filepath);

inputStream.open(filename);

if(inputStream.fail())

{

return false;

}

else

{

Eigen::read\_binary(filename, eigenfaces);

}

inputStream.close();

// check if eigenvalues binary file exists

sprintf(filename, "%s-values-binary.dat", filepath);

inputStream.open(filename);

if(inputStream.fail())

{

return false;

}

else

{

Eigen::read\_binary(filename, eigenvalues);

}

inputStream.close();

// check if eigenvalues average face file exists

sprintf(filename, "%s-avg-binary.dat", filepath);

inputStream.open(filename);

if(inputStream.fail())

{

return false;

}

else

{

Eigen::read\_binary(filename, averageFace);

}

inputStream.close();

return true;

}

// compares the second element of the pair

bool compare(pair<string, float> a, pair<string, float> b)

{

return a.second < b.second;

}

VectorXf projectOntoEigenspace(VectorXf newFace, VectorXf averageFace, MatrixXf eigenfaces)

{

//calculte the projection of the new face on the new face

vector<float> faceCoefficients;

VectorXf normalizedFace = newFace - averageFace;

VectorXf projectedFace(averageFace.rows());

projectedFace.fill(0);

for (int i = 0; i < eigenfaces.cols(); i++)

{

float a = (eigenfaces.col(i).transpose() \* normalizedFace)(0, 0);

faceCoefficients.push\_back(a);

projectedFace += (faceCoefficients[i] \* eigenfaces.col(i));

}

return projectedFace + averageFace;

}

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

{

// check if search ID maches ID of of any top n similar faces

for(int i=0; i<N; i++)

{

if(similarFaces[i].first == searchID)

{

return true;

}

}

return false;

}

void runClassifier(const char\* resultsFilepath, VectorXf averageFace, MatrixXf eigenfaces, VectorXf eigenvalues, vector<pair<string, VectorXf>> trainingFaces, vector<pair<string, VectorXf>> testFaces, float PCA\_percentage)

{

// PCA dimensionality reduction

float eigenvalues\_sum = eigenvalues.sum();

float currentEigenTotal = 0;

int count;

char filename[100];

// find number of vectors to the percentatge of info given by PCA\_percentage

for(count = 0; currentEigenTotal / eigenvalues\_sum < PCA\_percentage && count < eigenvalues.rows(); count++)

{

currentEigenTotal += eigenvalues.row(count)(0);

}

//reduce dimansionality from # of eigenface cols to value given by count

MatrixXf reducedEigenfaces(averageFace.rows(), count);

reducedEigenfaces = eigenfaces.block(0, 0, averageFace.rows(), count);

// project faces on reduced eigenfaces

vector<pair<string, VectorXf>> projectedTrainingFaces, projectedTestFaces;

for(int i=0; i<trainingFaces.size(); i++)

{

pair<string, VectorXf> temp(trainingFaces[i].first, projectOntoEigenspace(trainingFaces[i].second, averageFace, reducedEigenfaces));

projectedTrainingFaces.push\_back(temp);

}

for(int i=0; i<testFaces.size(); i++)

{

pair<string, VectorXf> temp(testFaces[i].first, projectOntoEigenspace(testFaces[i].second, averageFace, reducedEigenfaces));

projectedTestFaces.push\_back(temp);

}

// find correct and incorrect classifications

VectorXf projectedTestFace;

int correct = 0;

int incorrect = 0;

bool querySaved = false;

ofstream output;

vector<float> N\_Performances(50, 0);

sprintf(filename, "%s-%i-NImageNames.txt", resultsFilepath, (int)(PCA\_percentage\*100));

output.open(filename);

// check each query face for if it can be classified correctly

for(int i=0; i<testFaces.size(); i++)

{

projectedTestFace = projectedTestFaces[i].second;

// <image id, distance>

vector< pair<string, float> > queryPairs;

// check if correct or incorrect

querySaved = false;

// find distances from each training face to this test face and sort them in accending order

for(int j=0; j<trainingFaces.size(); j++)

{

pair<string, float> newPair(trainingFaces[j].first, (projectedTestFace - projectedTrainingFaces[j].second).norm());

queryPairs.push\_back(newPair);

}

sort(queryPairs.begin(), queryPairs.end(), compare);

for(int n=0; n<50; n++)

{

if(amongNMostSimilarFaces(queryPairs, n+1, projectedTestFaces[i].first)) // make func

{

N\_Performances[n] += 1;

// save correct match if N = 1

if(correct < 3 && !querySaved && n == 0)

{

output << "Correct Test Image " << correct << " ID: "

<< testFaces[i].first;

output << " Correct Training Image " << correct << " ID: "

<< queryPairs[0].first;

output << endl << endl;

correct++;

querySaved = true;

}

}

else

{

// save incorrect match if N = 1

if(incorrect < 3 && !querySaved && n == 0)

{

output << "Incorrect Test Image " << incorrect << " ID: " << testFaces[i].first;

output << " Incorrect Training Image " << incorrect << " ID: " << queryPairs[0].first;

output << endl << endl;

incorrect++;

querySaved = true;

}

}

}

}

output.close();

// write data for CMC curve to output file

sprintf(filename, "%s-%i.txt", resultsFilepath, (int)(PCA\_percentage\*100));

output.open(filename);

for(int n = 0; n < 50; n++)

{

output << n+1 << "\t" << (N\_Performances[n] / (float)testFaces.size()) << endl;

}

output.close();

}

void classifierThreshold(const char\* resultsFilepath, VectorXf averageFace, MatrixXf eigenfaces, VectorXf eigenvalues, vector<pair<string, VectorXf> > trainingFaces, vector<pair<string, VectorXf> > queryFaces, float PCA\_percentage)

{

// PCA dimensionality reduction

float eigenValuesSum = eigenvalues.sum();

float currentEigenTotal = 0;

int count;

char fileName[100];

// find number of vectors to the percentatge of info given by PCA\_percentage

for(count = 0; currentEigenTotal / eigenValuesSum < PCA\_percentage && count < eigenvalues.rows(); count++)

{

currentEigenTotal += eigenvalues.row(count)(0);

}

cout << "Dimensionality reduced from " << eigenfaces.cols() << " to " << count << endl;

//reduce dimansionality from # of eigenface cols to value given by count

MatrixXf reducedEigenFaces(averageFace.rows(), count);

reducedEigenFaces = eigenfaces.block(0,0,averageFace.rows(),count);

// project faces on reduced eigenfaces

vector<pair<string, VectorXf> > projectedTrainingFaces, projectedQueryFaces;

for(int i = 0; i < queryFaces.size(); i++)

{

pair<string, VectorXf> temp(queryFaces[i].first, projectOntoEigenspace(queryFaces[i].second, averageFace, reducedEigenFaces));

projectedQueryFaces.push\_back(temp);

}

// find true positives and false positives

VectorXf projQueryFace;

int TPCount, FPCount;

TPCount = 0;

FPCount = 0;

pair<int, int> temp(0, 0);

vector< pair<int, int> > counts(1800, temp);

for (int i = 0; i < projectedQueryFaces.size(); i++)

{

cout << "\rQuery Face: " << i;

projQueryFace = projectedQueryFaces[i].second;

vector<pair<string, float> > queryPairs;

for (int t = 0; t < trainingFaces.size(); t++)

{

pair<string, float> newPair(trainingFaces[t].first, (projQueryFace - trainingFaces[t].second).norm());

queryPairs.push\_back(newPair);

}

sort(queryPairs.begin(), queryPairs.end(), compare);

cout << "\t" << queryPairs[0].second << endl;

//for(int threshold = 380; threshold < 1500; threshold += 5) // high res

for (int threshold = 50; threshold < 600; threshold += 2) // low res

{

if(queryPairs[0].second <= threshold)

{

// true positive case

if (atoi(projectedQueryFaces[i].first.c\_str()) > 50)

{

counts[threshold].first++;

}

// false positive case

else

{

counts[threshold].second++;

}

}

}

}

// output results

sprintf(fileName, "%s-%i.txt", resultsFilepath, (int)(PCA\_percentage\*100));

ofstream output;

output.open(fileName);

//for(int threshold = 380; threshold < 1500; threshold += 5) // high res

for(int threshold = 50; threshold < 600; threshold += 2) // low res

{

float TPRate = (float)counts[threshold].first / (float)trainingFaces.size();

float FPRate = (float)counts[threshold].second / ((float)queryFaces.size() - (float)trainingFaces.size());

output << threshold << "\t" << TPRate << "\t" << FPRate << endl;

}

output.close();

}