University of Nevada, Reno

CS 485 — Computer Vision

Assignment #5

Joshua Gleason

Instructor:
Dr. George Bebis

April 28, 2011

Contents

List of Figures

1 Introduction

Comparison of extremely high dimensional data using direct methods has shown to be too computationally expensive and inaccurate. By transforming the data to a customized, low dimensional space; constructed solely to represent one particular type of data; a faster, more accurate comparison can be made. This new space can be formed using a method known as Principal Component Analysis (PCA). PCA determines a basis for a lower dimensional space by calculating the eigenvectors and eigenvalues of the covariance matrix. The covariance matrix is a special matrix constructed from data in a set containing one particular type of data. The eigenvectors which correspond to the largest eigenvalues are then kept to form a basis. It has been shown that using this method, the dimensionality of the data to be significantly reduced, while retaining nearly all of the information.

For the eigenface approach, the space which represents the vectors are is constructed with the sole intent of representing faces. This is done by taking a large database of face images, representing them as vectors, then using PCA to form a new basis. The basis vectors are known as eigenfaces because they are eigenvectors yet resemble faces. Some of these eigenfaces can be seen in Fig. 2. From now on, the space defined by the eigenfaces will be known as the eigenspace.

Every face in the database can be represented as a linear combination of the eigenfaces. The coefficients in this linear combination are the new vectors used to represent the various faces. Other images may also be transformed into the eigenspace, under the constraint that they are the same size as the original images from the database.

2 Datasets

For the experiments, two datasets are used, a training set containing 1204 faces from 867 subjects, and a testing set containing 1196 faces from 866 subjects. All 866 subjects from the testing dataset exist in the training database (i.e., there is one extra subject in the training set). A truncated training set is used in the second experiment where the images of the first 50 subjects are removed (i.e., the testing set contains 50 subjects that do not exist in the training set). Each dataset exists at two resolutions, 48×60 and 16×20, all experiments were done at both high and low resolutions.

3 Experiments

The first experiment attempts to obtain the identity of a face using the eigenspace approach. The eigenspace is constructed using the 1204 faces from the training set. The faces from the testing set are then transformed into the eigenspace and compared to the training faces. This comparison is done using the Malahanobis distance (Eq. 1) rather than the Euclidean distance. The Malahanobis distance allows variations along all axes to be treated as equally significant. If the correct identification is found among the top N matches, then the match was successful. By varying N from 1 to 50, and recording the accuracy across the entire testing set, the CMC graph is obtained. This is used to represent the matching power of the algorithm. This experiment is repeated three times using eigenspaces constructed to retain 80%, 90%, and 95% of the information from the training set.

$$\sum_{i=1}^{K} \frac{1}{\lambda_i}(w_i - w_i^l)^2 \tag{1}$$

The second experiment attempts to detect intruders in the test set. This is done by using a training set with the first 50 subjects removed. The eigenspace is then constructed using 95% information retention. Each face from the test set is compared to each face in the training set and the smallest Malahanobis distance score is found (Eq. 1). If this score is above a particular threshold the face is rejected as an intruder. By using many different thresholds, the ROC curve is found which represents this methods ability to reject intruders while retaining non-intruders.

4 Results

4.1 Face Identification

The CMC graph for the high-resolution dataset (Fig. 4) shows that using all three eigenspaces give similar results. The 90% eigenspace seems to yield the best results, however the 80% eigenspace gives nearly identical results once N becomes larger than 25. The 95% eigenspace is similar the the 90% eigenspace when N is less than 5, but quickly diverges and maintains a 4% less recognition rate for larger N. Based on these results, 90% gives the best results, but if speed is a factor, 80% gives very similar results as well.

These results seem to signify that most of the important information needed for high-level comparison is in the first coefficients. If too many are used then useless information is introduced and performance begin to deteriorate. Examples of correctly matched faces, along with top matches from the training set can be seen in Figs. 6, 8, and 10. The left-most image is the test face, and the following faces to the right are the best matched faces from the training set. The largest of these right-most faces indicate correct matches. Examples of incorrectly matched faces can be seen in Figs. 7, 9, and 11 along with the ten best matches where non of these are the correct match.

The CMC graph for the low-resolution dataset (Fig. 15) shows that perhaps 80% information retention is not good enough for low-resolution images. This graph also shows an improvement of approximately $2 - 5\%$ for the 90% and 95% eigenspaces. This result is counter-intuitive because people generally find it more difficult to identify a person from a low resolution image. A possible explanation of this phenomenon is that, at lower resolutions, only large defining features are kept. This allows insignificant information such as some lighting conditions and noise to be less invasive.

4.2 Intruder Detection

The ROC generated by testing many thresholds in the high-resolution dataset (Fig. 5) appears to accept approximately 25% of intruders while retaining 70% of non-intruders. Although this ratio is not great, this method definitely has some intruder detection abilities. By setting a high threshold, 90% of of non-intruders are accepted and 50% of intruders are accepted. By using this high threshold, it may be possible to make this a part of a cascade detection system which could use this as one of multiple steps in intruder detection.

At low-resolution (Fig. 16, the rejection/acceptance rate is nearly identical. This implies that using high-resolution images is not benificial and should therefore not be done because low-resolution images allow for faster calcutions with no observable negative effects.

5 Questions

1. What is dimensionality reduction and why is it important?

   (a) Dimensionality reduction reduces the number of coefficients required to represent a vector. This is important because it reduces the processing time for any computation on the vector.

2. What is the criterion used by PCA for determining a space of low dimensionality?

   (a) Find a basis that includes only the vectors which allow for the greatest information retention.

3. How is the lower dimensionality space computed using PCA?

   (a) Form the lower dimensionality basis from the top K eigenvectors of the covariance matrix C, where C is calculated using the following formula.
   Assume $x_1, x_2, ..., x_M$ are column vectors and $|x_i| = N$
      i. $\overline{x} = \frac{1}{M} \sum_{i=1}^{M} x_i$
      ii. $\Phi_i = x_i - \overline{x}$
      iii. form the matrix $A = [\Phi_1 \Phi_2 \cdots \Phi_M]$ (A is an $N \times M$ matrix)

      $$C = \frac{1}{M} \sum_{n=1}^{M} \Phi_n \Phi_n^T = \frac{1}{M} A A^T$$

4. What is the geometrical interpretation of PCA?

   (a) The eigenvectors used to form the basis represent the axes where the data in x varies the most. The axes where the data varies the least represents the least information and is therefore removed.

5. How do we choose the number of dimensions?

   (a) After determining the amount of data to preserve, PCA takes the top K eigenvectors of AA^T where K is the minimum number of eigenvalues λ_i of AA^T required to meet the following formula.

   $$\frac{\sum_{i=1}^{K} \lambda_i}{\sum_{i=1}^{N} \lambda_i} > T$$

   Where $0 < T \leq 1$ is the minimum amount of data to preserve.

6. What are the steps for face recognition using PCA?

   (a) Take each $N \times N$ face image and in the training set and turn it into an N^2 column vector x_i then use PCA to reduce the dimensionality of x_i from N^2 to K where Ω_i^l are the reduced vectors , Ψ is the mean face, and u_i are the basis eigenvectors.
      i. Normalize $\Gamma : \Phi = \Gamma - \Psi$
      ii. Project Ψ onto the eigenspace: $\Omega = [w_1 w_2 \cdots w_3]^T$ ($w_i = u_i^T \Phi$)
      iii. compute $e_r = min \left\| \Omega - \Omega_i^l \right\|$ using the Mahalanobis distance.
      iv. if $e_r < Threshold$ then Γ is a face, otherwise it is not.

6 Images

Figure 1: Average face

Figure 2: Top ten Eigenfaces

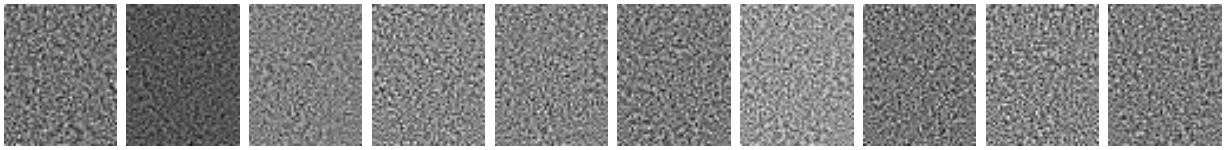

Figure 3: Bottom ten Eigenfaces

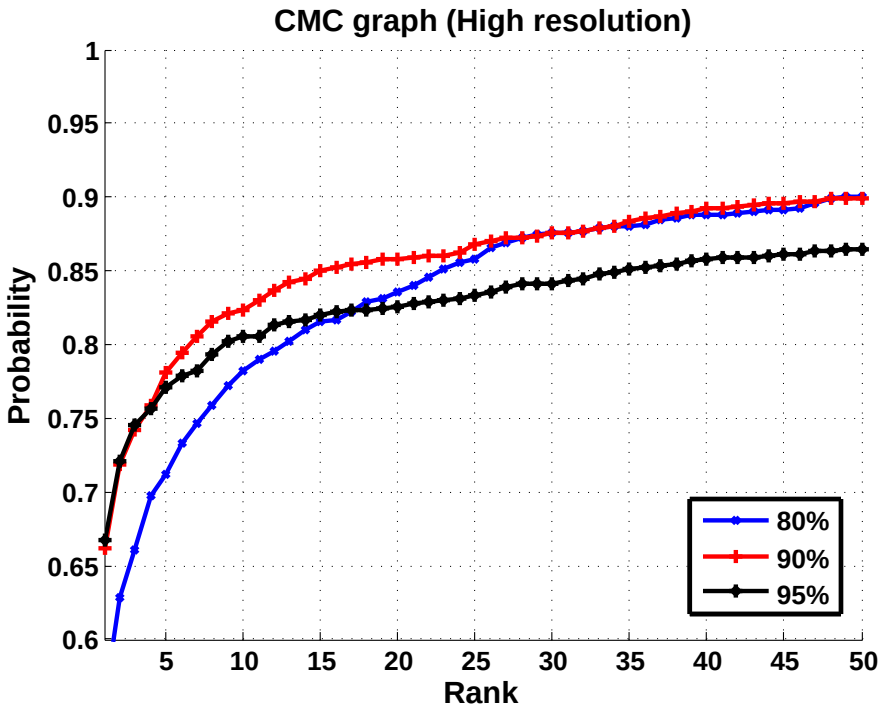

Figure 4: High resolution CMC for training at 80%, 90%, and 95% information retention.

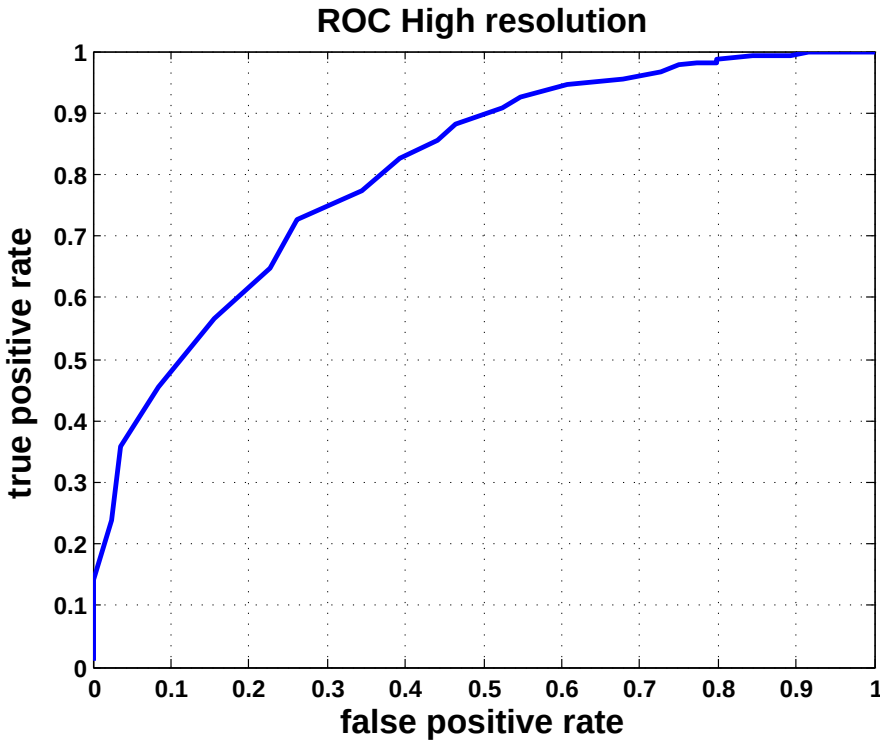

Figure 5: High resolution ROC.

Figure 6: Top matches for 3 correctly matched faces at 80% information retention.

Figure 7: Top matches for 3 incorrectly matched faces at 80% information retention.

Figure 8: Top matches for 3 correctly matched faces at 90% information retention.

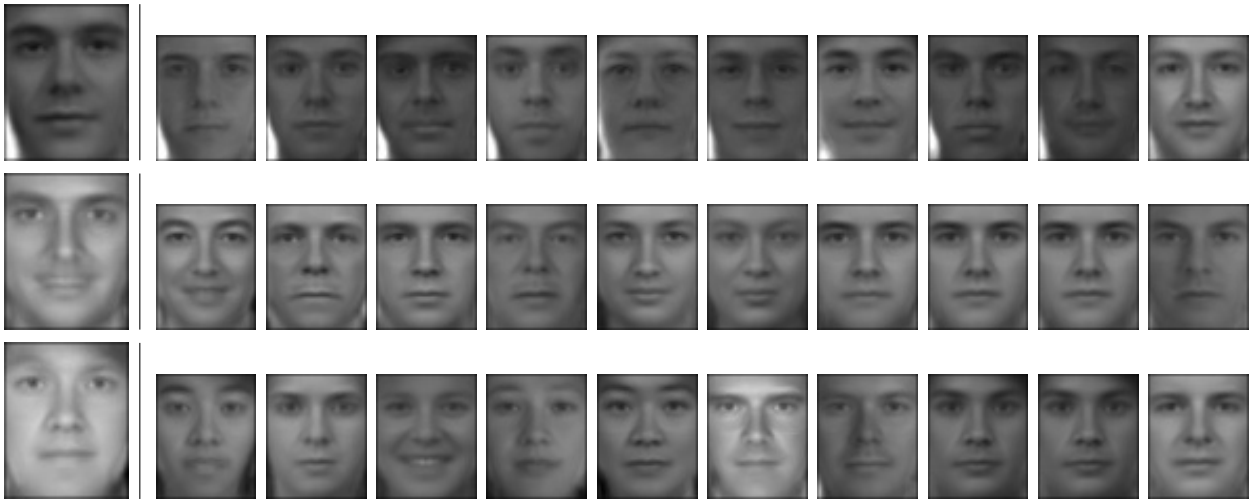

Figure 9: Top matches for 3 incorrectly matched faces at 90% information retention.

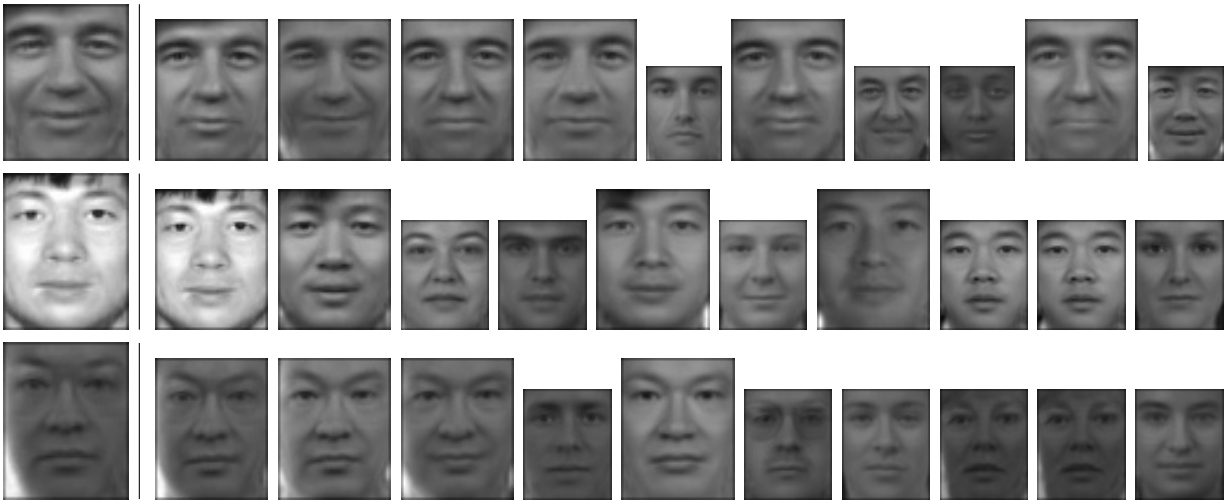

Figure 10: Top matches for 3 correctly matched faces at 95% information retention.

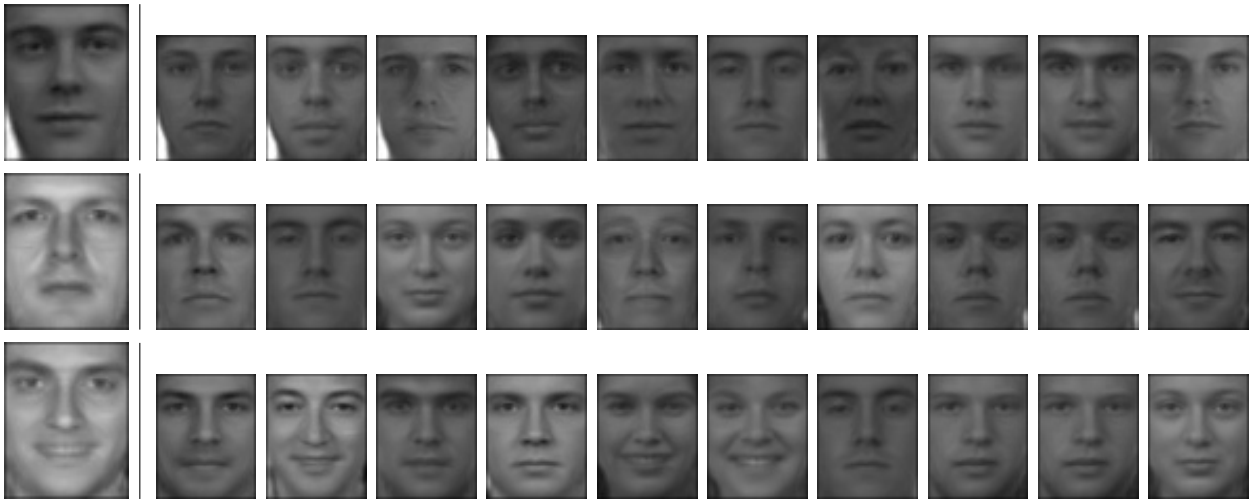

Figure 11: Top matches for 3 incorrectly matched faces at 95% information retention.

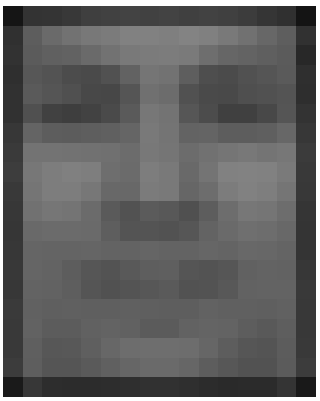

Figure 12: Average face (Low res)

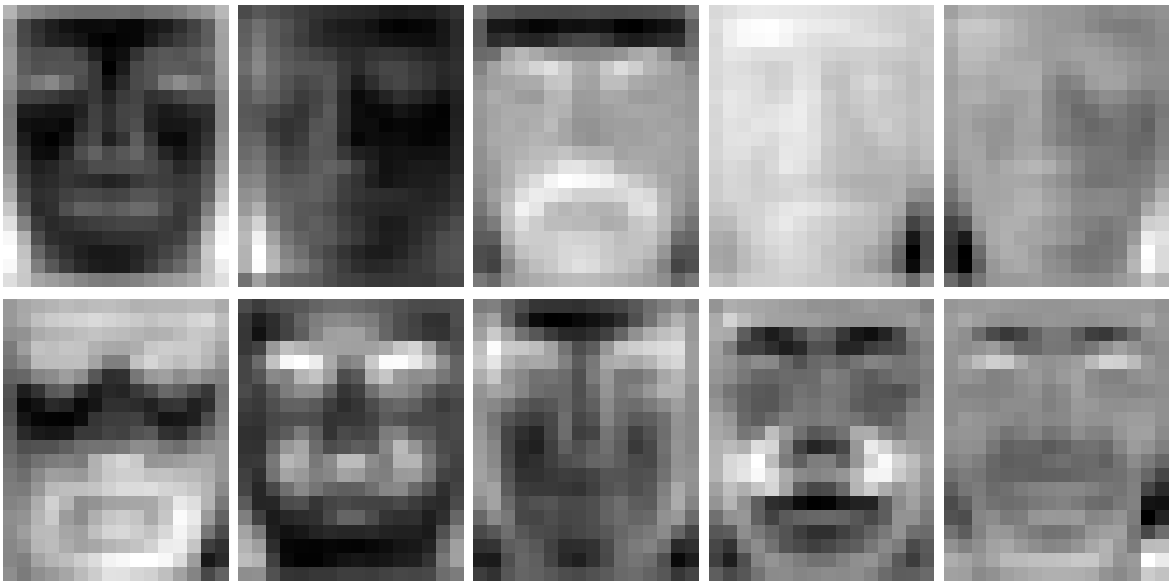

Figure 13: Top ten low resolution Eigenfaces

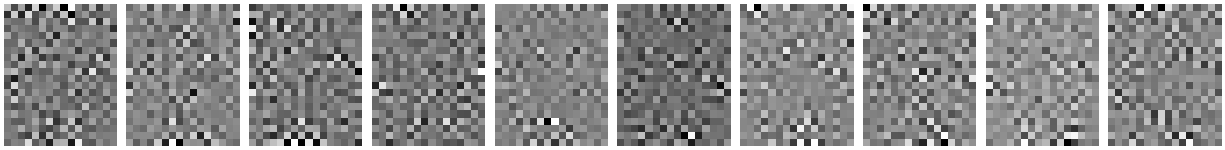

Figure 14: Bottom ten low resolution Eigenfaces

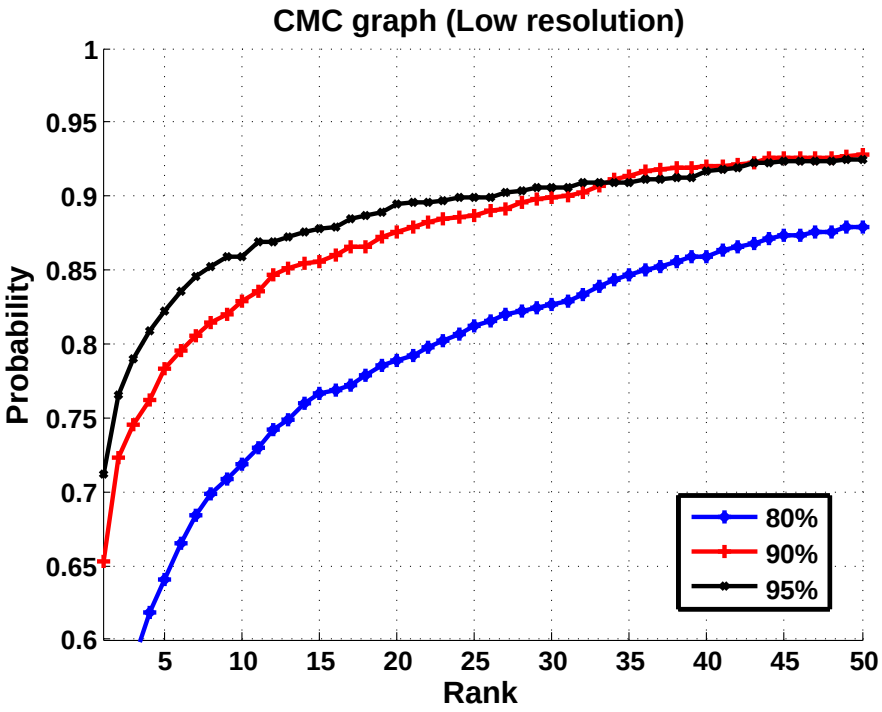

Figure 15: Low resolution CMC for training at 80%, 90%, and 95% information retention.

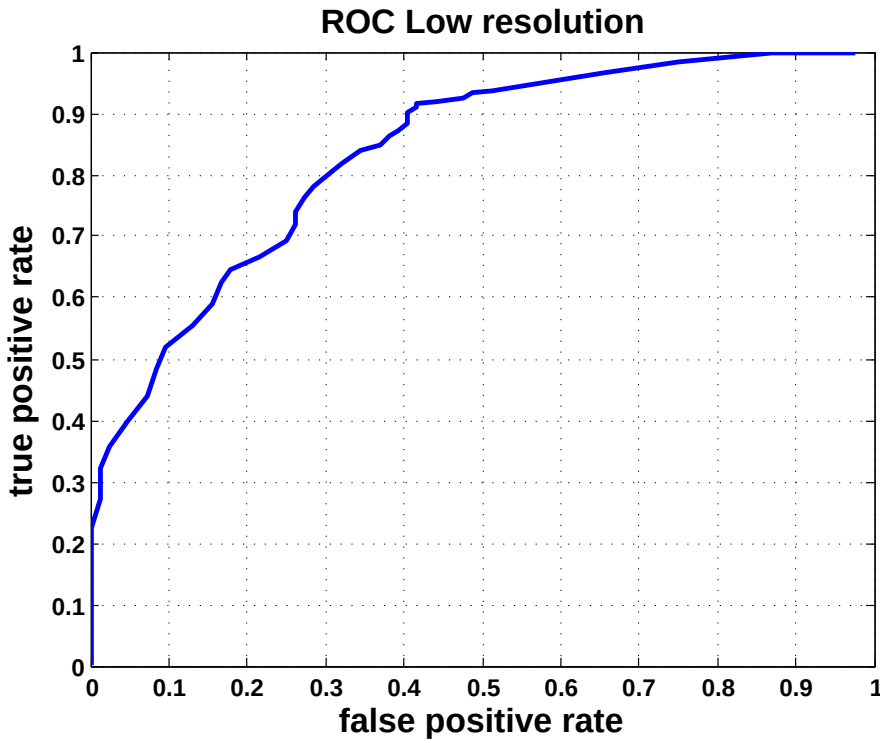

Figure 16: Low resolution ROC.

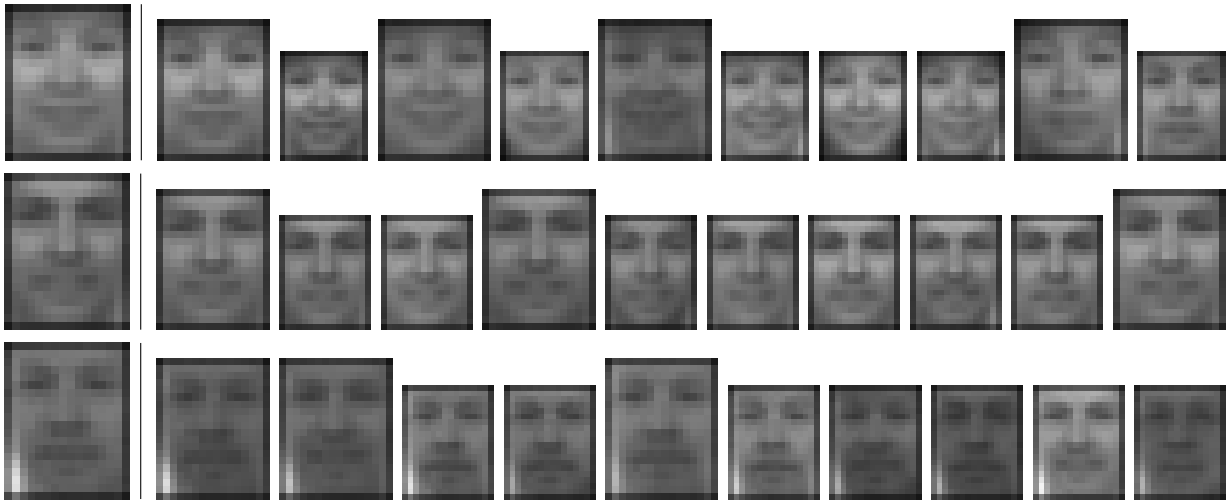

Figure 17: Top matches for 3 correctly matched faces at 80% (Low res).

Figure 18: Top matches for 3 incorrectly matched faces at 80% (Low res).

Figure 19: Top matches for 3 correctly matched faces at 90% (Low res).

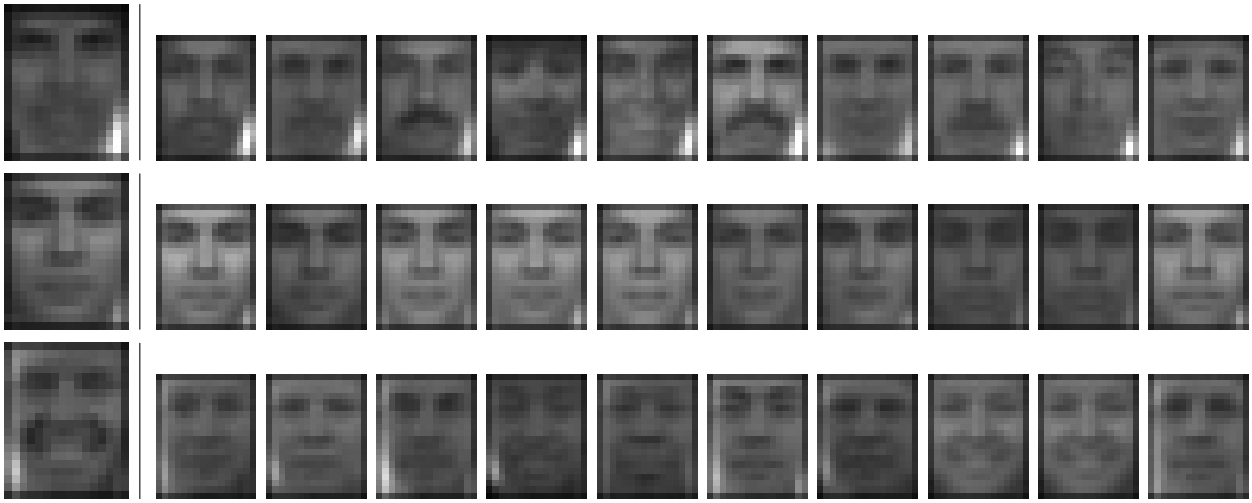

Figure 20: Top matches for 3 incorrectly matched faces at 90% (Low res).

Figure 21: Top matches for 3 correctly matched faces at 95% (Low res).

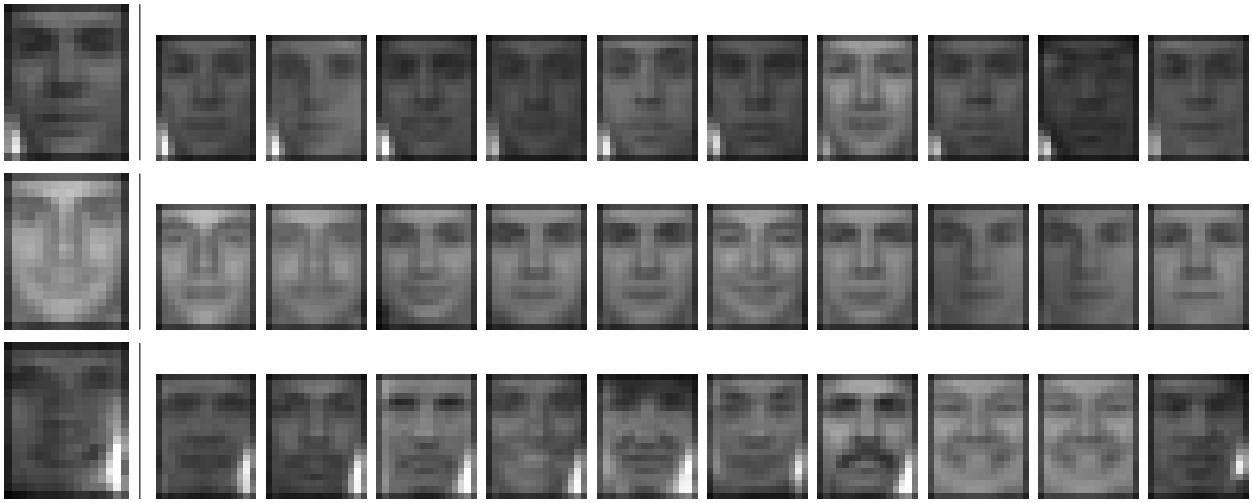

Figure 22: Top matches for 3 incorrectly matched faces at 95% (Low res).

7 Source Code

7.1 eigface.h

```
#ifndef JOSH_EIGFACE
#define JOSH_EIGFACE

#include <cv.h>
#include <cvaux.h>
#include <highgui.h>

using namespace cv;
using namespace std;

void readImagesFile(string filename, vector<string>& images, Size& size)
{
  ifstream fin(filename.c_str());
  string a;

  if (!fin.good())
    return;

  fin >> size.width >> size.height;
  getline(fin,a);

  while ( fin.good() )
  {
    getline(fin,a);
    if ( fin.good() )
      images.push_back(a);
  }
  fin.close();
}

// returns column vector of image (output is 32FC1)
void linearizeImage(const Mat& input, Mat& output)
{
  int rows = input.rows, cols = input.cols;
  int size = rows*cols, r, c, x;
  output = Mat(size,1,CV_32FC1);
  Mat in;

  if ( input.type() != CV_32FC1 )
    input.convertTo(in, CV_32FC1);
  else
    in = input;

  for ( c = 0, x = 0; c < cols; c++ )
    for ( r = 0; r < rows; r++, x++)
      output.at<float>(x,0) = in.at<float>(r,c);
}

// turns a column vector back to an image
void delinearizeImage(const Mat&,Mat&,const Size&, int outputType=CV_32FC1);
void delinearizeImage(const Mat& input, Mat& output, const Size& imgSize, int outputType
    )
{
  int rows = imgSize.height, cols = imgSize.width, r, c, x;
  Mat out, in;

  if ( outputType != CV_32FC1 )
    out = Mat(imgSize, CV_32FC1);
  else // out alias output now
  {
    output = Mat(imgSize, CV_32FC1);
    out = output;
  }

  if ( input.type() != CV_32FC1 )
    input.convertTo(in, CV_32FC1);
  else
    in = input;
```

```
  for ( c = 0, x = 0; c < cols; c++ )
    for ( r = 0; r < rows; r++, x++ )
      out.at<float>(r,c) = in.at<float>(x,0);

  if ( outputType != CV_32FC1 )
    out.convertTo(output, outputType);
}

void buildA(const Mat& mean, const vector<string>& img, Mat& A)
{
  int imgCount = img.size();
  int rows, cols;
  A = Mat(mean.rows, imgCount, CV_32FC1);

  int r, c, aR, aC;
  Mat face;

  for ( aC = 0; aC < imgCount; aC++ )
  {
    face = imread(img[aC],0);
    for ( c = 0, aR = 0; c < face.cols; c++ )
      for ( r = 0; r < face.rows; r++, aR++ )
        A.at<float>(aR,aC) = static_cast<float>(face.at<uchar>(r,c)) - mean.at<float>(aR
            ,0);
  }
}

void projectFace(const Mat& image, const vector<Mat>& eFaces, const Mat& mean,
  Mat& coeff)
{
  int dim = eFaces.size(), i;
  coeff = Mat(dim,1,CV_32FC1);

  Mat imgVector;

  linearizeImage(image, imgVector);

  imgVector -= mean;

  for ( i = 0; i < dim; i++ )
  {
    Mat t1, t2;
    transpose(eFaces[i],t1);

    t2=t1*imgVector;

    coeff.at<float>(i,0) = t2.at<float>(0,0);
  }
}

void backprojectFace(const Mat&, const vector<Mat>&, const Mat&, Size&, Mat&,
  int outputType=CV_32FC1);
void backprojectFace(const Mat& coeff, const vector<Mat>& eFaces,
  const Mat& meanFace, Size& imgSize, Mat& image, int outputType)
{
  int dim = eFaces.size(), i;

  Mat linImg = meanFace.clone();

  for ( i = 0; i < dim; i++ )
    linImg += eFaces[i]*coeff.at<float>(i,0);

  delinearizeImage(linImg, image, imgSize, outputType);
}

bool meanFace(const Size& imgSize, const vector<string>& img, Mat& mean)
{
  int imgCount = img.size();
  Mat face;

  // initialize mean face to all zeros
  mean = Mat(imgSize.height*imgSize.width,1,CV_32FC1,Scalar(0));
```

```
  int i, r, c, x;

  for ( i = 0; i < imgCount; i++ )
  {
    face = imread(img[i],0);
    for ( c = 0, x = 0; c < imgSize.width; c++ )
      for ( r = 0; r < imgSize.height; r++, x++ )
        mean.at<float>(x,0) += static_cast<float>(face.at<uchar>(r,c));
  }

  mean *= (1.0/imgCount);
}

#endif // JOSH_EIGFACE
```

7.2 train.cc

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <iomanip>
#include <cv.h>
#include <cvaux.h>
#include <highgui.h>
#include "eigface.h"

using namespace std;
using namespace cv;

//#define WRITE_EXTRAS

int findK(const Mat& eigValues, float thresh)
{
  if ( thresh >= 1 )
    return eigValues.rows;

  int i;
  float total = 0, sum = 0;
  // find sum of eigen values
  for ( i = 0; i < eigValues.rows; i++ )
    total += eigValues.at<float>(i,0);

  for ( i = 0; i < eigValues.rows; i++ )
  {
    sum += eigValues.at<float>(i,0);
    if ( sum / total >= thresh )
      return i+1;
  }

  return eigValues.rows;
}

bool saveEigenfaces( string filename, const vector<Mat>& eFaces, const Mat& eValues,
  const vector<Mat>& coeffs, const vector<string>& images, const Mat& mean,
  const Size& imgSize)
{
  ofstream fout(filename.c_str());

  int pixels = imgSize.height * imgSize.width,
      dim = eFaces.size(),
      imgCount = images.size(),
      i, j;

  // fout.precision(100);

  // write useful information
  fout << imgSize.height << ' ' << imgSize.width << ' ' << dim << ' ' << imgCount <<
      endl;

  // write mean face
  for ( i = 0; i < pixels; i++ )
```

```
    fout << ' ' << mean.at<float>(i,0);
  fout << endl;

  // write eigan values
  for ( i = 0; i < dim; i++ )
    fout << ' ' << eValues.at<float>(i,0);
  fout << endl;

  // write eigenfaces
  for ( i = 0; i < dim; i++ )
  {
    for ( j = 0; j < pixels; j++ )
      fout << ' ' << eFaces[i].at<float>(j,0);
    fout << endl;
  }

  // write coefficients
  for ( i = 0; i < imgCount; i++ )
  {
    fout << ' ' << images[i];
    for ( j = 0; j < dim; j++ )
      fout << ' ' << coeffs[i].at<float>(j,0);
    fout << endl;
  }

  fout.close();
}

// argv[1] : training images
// argv[2] : percentage acc
// argv[3] : output file
int main(int argc, char *argv[])
{
  if ( argc < 4 )
    return -1;

  // declare variables
  Size imgSize;
  vector<string> imgList;
  float thresh;
  int k, i;

  Mat mean, A, Atrans, lambda, eigVectors;

  vector<Mat> u;
  vector<double> w;

  // get desired reconstruction error
  thresh = atof(argv[2]);

  // read input file
  readImagesFile(argv[1],imgList,imgSize);

  // compute mean face
  meanFace(imgSize,imgList,mean);

  // build the A matrix and calculate transpose
  buildA(mean, imgList, A);
  transpose(A,Atrans);

  // calculate eigen vals/vectors
  eigen(Atrans*A, lambda, eigVectors);

  // find minimum k to satisfy threshold
  k = findK(lambda, thresh);
  cout << "Eigenvectors Kept: " << k << endl;

  // take top k eigen vectors, normalize and store into array
  for ( i = 0; i < k; i++ )
  {
    static Mat t1, t2, t3, v;
    t1 = eigVectors.row(i).clone();
    transpose(t1,v);
```

```
    t2 = A*v;
    normalize(t2, t3);

    u.push_back(t3.clone());
  }

#ifdef WRITE_EXTRAS
  // save mean face
  {
    Mat meany;
    delinearizeImage(mean,meany,imgSize,CV_8UC1);
    imwrite("results/mean_face.jpg",meany);
  }

  // save largest/smallest eigenfaces
  for ( i = 0; i < 10; i++ )
  {
    static Mat t1, t2;
    delinearizeImage(u[i],t1,imgSize);
    normalize(t1,t2,0,255,CV_MINMAX,CV_8UC1);
    ostringstream sout;
    sout << "results/eigenfaces/largest" << i+1 << ".jpg";
    imwrite(sout.str(), t2);
  }

  for ( i = 0; i < 10; i++ )
  {
    static Mat t1, t2, t3, v;
    int index = imgList.size()-i-1;
    t1 = eigVectors.row(index).clone();
    transpose(t1,v);
    t2 = A*v;
    normalize(t2,t3);
    delinearizeImage(t3,t1,imgSize);
    normalize(t1,t2,0,255,CV_MINMAX,CV_8UC1);
    ostringstream sout;
    sout << "results/eigenfaces/smallest" << i+1 << ".jpg";
    imwrite(sout.str(), t2);
  }
#endif // WRITE_EXTRAS

  // calculate coefficients for all images
  vector<Mat> coeffs;

  for ( i = 0; i < imgList.size(); i++ )
  {
    Mat image = imread(imgList[i],0);
    Mat m;
    projectFace(image,u,mean,m);
    coeffs.push_back(m.clone());
  }

  // save results to file
  saveEigenfaces(argv[3],u,lambda,coeffs,imgList,mean,imgSize);

  // exit
  return 0;
}
```

7.3 test.h

```
#ifndef TEST_H_JOSH
#define TEST_H_JOSH

#define USE_MAHALANOBIS

bool loadEigenfaces( string filename, vector<Mat>& eFaces, Mat& eValues, vector<Mat>&
    coeffs,
  vector<string>& images, Mat& mean, Size& imgSize )
{
  ifstream fin(filename.c_str());
```

```
  if ( !fin.good() )
    return false;

  int dim, imgCount, pixels, i, j;

  // read useful information
  fin >> imgSize.height >> imgSize.width >> dim >> imgCount;

  pixels = imgSize.height*imgSize.width;

  // resize vectors for speed
  mean = Mat(pixels,1,CV_32FC1);

  eValues = Mat(dim,1,CV_32FC1);
  eFaces.resize(dim);
  for ( i = 0; i < dim; i++ )
    eFaces[i] = Mat(pixels,1,CV_32FC1);

  images.resize(imgCount);
  coeffs.resize(imgCount);
  for ( i = 0; i < imgCount; i++ )
    coeffs[i] = Mat(dim,1,CV_32FC1);

  // read mean face
  for ( i = 0; i < pixels; i++ )
    fin >> mean.at<float>(i,0);

  // read eigenvalues
  for ( i = 0; i < dim; i++ )
    fin >> eValues.at<float>(i,0);

  // read eigenfaces
  for ( i = 0; i < dim; i++ )
    for ( j = 0; j < pixels; j++ )
      fin >> eFaces[i].at<float>(j,0);

  // read coefficients
  for ( i = 0; i < imgCount; i++ )
  {
    fin >> images[i];
    for ( j = 0; j < dim; j++ )
      fin >> coeffs[i].at<float>(j,0);
  }

  fin.close();

  return true;
}

template<class T>
void my_swap(T& a, T& b)
{
  T temp = a;
  a = b;
  b = temp;
}

// complicated but fast for relatively small N (doesn't need to sort entire list)
void topNMatches(const Mat& input, const vector<Mat>& coeffs, const Mat& eValues, int N,
  vector<int>& matching, vector<double>& matchValues)
{
  matchValues.resize(N,0);    // holds the match score
  matching.resize(N,-1);      // holds the index

  int setSize = coeffs.size(), i, j, k, val1;
  double e, val2;
  Mat diff, t;

  for ( i = 0; i < setSize; i++ )
  {

#ifdef USE_MAHALANOBIS
    diff = input - coeffs[i];
```

```
    diff = diff.mul(diff);
    diff = diff.mul(1.0/eValues);
    e = norm(diff,NORM_L1);
#else   // use L2 norm
    e = norm(input,coeffs[i],NORM_L2);
#endif  // USE_MAHALANOBIS

    for ( j = 0; j < N; j++ )
    {
      if ( matching[j] < 0 || e < matchValues[j] )
        break;
    }

    if ( j < N )  // insert element here
    {
      if ( matching[j] < 0 )  // special case
      {
        matching[j] = i;
        matchValues[j] = e;
      }
      else
      {
        val1 = i;
        val2 = e;
        for ( k = j; k < N; k++ )
        {
          if ( matching[k] < 0 ) // unfilled position, place here
          {
            matching[k] = val1;
            matchValues[k] = val2;
            break;
          }
          else if ( val2 < matchValues[k] ) // swap down
          {
            swap(val1,matching[k]);
            swap(val2,matchValues[k]);
          }
          else
            break;
        }
      }
    }
  }
}

int matchTest(long ID, const vector<int>& indexMatches, const vector<long>& trainIDs)
{
  int count = 0;
  for ( int i = 0; i < indexMatches.size(); i++ )
    if ( ID == trainIDs[indexMatches[i]] )
      count++;
  return count;
}

// returns the ID based on file path
long getID( const string pathname )
{
  // find the leftmost '/' (or from beginning if none)
  int lastFSlash = -1;
  for ( int l = pathname.length()-1; l >= 0; l-- )
    if ( pathname[l] == '/' )
    {
      lastFSlash = l;
      break;
    }

  // read the first 5 after the last '/'
  long id = 0;
  int tens = 1; // never gets over 10,000 so no need for long
  for ( int j = lastFSlash+5; j > lastFSlash; j-- )
  {
    id += (long)(pathname[j]-'0')*tens;
    tens *= 10;
```

```
  }
  return id;
}

// figure out the unique face ID based on the filename
void buildIDs(const vector<string>& images, vector<long>& idList)
{
  int size = images.size();
  idList.resize(size);

  for ( int i = 0; i < size; i++ )
    idList[i] = getID(images[i]);
}

#endif // TEST_H_JOSH
```

7.4 testa.cc

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <iomanip>
#include <cv.h>
#include <cvaux.h>
#include <highgui.h>
#include "eigface.h"
#include "test.h"

using namespace std;
using namespace cv;

//#define WRITE_EXTRAS

// argv[1] : training data
// argv[2] : test image file
// argv[3] : N
int main(int argc, char *argv[])
{
  if ( argc < 4 )
    return -1;

  int N = atoi(argv[3]);

  vector<Mat> eFaces, trainCoeffs, testCoeffs;
  vector<string> trainImgs, testImgs;
  Mat mean, lambda;
  Size imgSize;
  vector<long> trainIDs, testIDs;
  vector<vector<int> > trainMatches;
  int trainCount, testCount, i ,j;
  vector<int> matchValue;
  vector<vector<double> > topNError;

  // load training data
  if ( !loadEigenfaces(argv[1],eFaces,lambda,trainCoeffs,trainImgs,mean,imgSize) )
    return -1;

  // build ID list from image names
  buildIDs(trainImgs,trainIDs);

  // get training set size
  trainCount = trainImgs.size();

  // read test images folder
  {
    Size tempSize;
    readImagesFile(argv[2], testImgs, tempSize);
    if ( tempSize != imgSize )
    {
      cout << "Error: Training and Testing images have different dimensions" << endl;
      return -1;
    }
```

```
  }

  // build ID list from image names
  buildIDs(testImgs,testIDs);

  // get testing set size
  testCount = testImgs.size();

  testCoeffs.resize(testCount);
  trainMatches.resize(testCount);
  matchValue.resize(testCount);
  topNError.resize(testCount);

  int matchCount = 0;

  // build list of test set coefficients
  for ( i = 0; i < testCount; i++ )
  {
    static Mat input;
    input = imread(testImgs[i],0);
    projectFace(input, eFaces, mean, testCoeffs[i]);
    topNMatches(testCoeffs[i], trainCoeffs, lambda, N, trainMatches[i], topNError[i]);

    matchValue[i] = matchTest(testIDs[i], trainMatches[i], trainIDs);
    if ( matchValue[i] > 0 )
      matchCount++;
  }

  // output accuracy to terminal
  cout << (float)matchCount / testCount << endl;

// write top matches for 3 valid and 3 invalid (optional)
#ifdef WRITE_EXTRAS
  int max = matchValue[0], count=0;
  int usedIDs[] = {0,0,0,0,0,0};
  for ( i = 0; i < testCount; i++ )
    if ( matchValue[i] > max )
      max = matchValue[i];

  // write best 3 matches with top N matches ensuring 3 different IDs
  for ( i = max; i >= 0 && count < 3; i-- )
    for ( j = 0; j < testCount && count < 3; j++ )
      if ( matchValue[j] == i &&
           usedIDs[0] != testIDs[j] &&
           usedIDs[1] != testIDs[j])
      {
        usedIDs[count] = testIDs[j];
        cout << i << endl;
        count++;
        // save image j from test set with best matches
        string path;
        {
          ostringstream sout;
          string a = argv[1];
          sout << "results/correct" << a[a.length()-2] << a[a.length()-1] << '/'
               << count << "/";
          path = sout.str();
        }
        Mat tImg;
        backprojectFace(testCoeffs[j],eFaces,mean,imgSize,tImg,CV_8UC1);
        imwrite(path+(string)"testImg.jpg",tImg);

        // write top N matches
        for ( int k = 0; k < N; k++ )
        {
          ostringstream sout;
          sout << path << setfill('0') << setw(2) << k+1 << '_'
               << setw(4) << topNError[j][k]*100000;
          if ( testIDs[j] == trainIDs[trainMatches[j][k]] )
            sout << "_c.jpg";
          else
            sout << "_i.jpg";
          backprojectFace(trainCoeffs[trainMatches[j][k]],eFaces,mean,imgSize,tImg,
```

```
                CV_8UC1);
            imwrite(sout.str(),tImg);
          }
        }

    // write 3 non-matching with top N matches ensuring all have a different ID
    for ( i = 0; i < testCount && count < 6; i++ )
    {
      if ( matchValue[i] <= 0 &&
        usedIDs[0] != testIDs[i] &&
        usedIDs[1] != testIDs[i] &&
        usedIDs[2] != testIDs[i] &&
        usedIDs[3] != testIDs[i] &&
        usedIDs[4] != testIDs[i] )
      {
        usedIDs[count] = testIDs[i];
        count++;
        // save image i from test set with best matches
        string path;
        {
          ostringstream sout;
          string a = argv[1];
          sout << "results/incorrect" << a[a.length()-2] << a[a.length()-1] << '/'
               << count-3 << "/";
          path = sout.str();
        }
        Mat tImg;
        backprojectFace(testCoeffs[i],eFaces,mean,imgSize,tImg,CV_8UC1);
        imwrite(path+(string)"testImg.jpg",tImg);

        for ( int k = 0; k < N; k++ )
        {
          ostringstream sout;
          sout << path << setfill('0') << setw(2) << k+1 << '_'
               << setw(4) << topNError[i][k]*100000 << "_" << trainIDs[trainMatches[i][k]]
               << ".jpg";
          backprojectFace(trainCoeffs[trainMatches[i][k]],eFaces,mean,imgSize,tImg,CV_8UC1
              );
          imwrite(sout.str(),tImg);
        }
      }
    }

  #endif

    return 0;
  }
```

7.5 testb.cc

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <iomanip>
#include <cv.h>
#include <cvaux.h>
#include <highgui.h>
#include "eigface.h"
#include "test.h"

using namespace std;
using namespace cv;

// argv[1] : training data
// argv[2] : test image file
// argv[3] : N
int main(int argc, char *argv[])
{
  if ( argc < 4 )
    return -1;

  int N = 1; //atoi(argv[3]);
```

```
double thresh = atof(argv[3]);

vector<Mat> eFaces, trainCoeffs, testCoeffs;
vector<string> trainImgs, testImgs;
Mat mean, lambda;
Size imgSize;
vector<long> trainIDs, testIDs;
vector<vector<int> > trainMatches;
int trainCount, testCount, i ,j;
vector<int> matchValue;
vector<vector<double> > topNError;

// load training data
if ( !loadEigenfaces(argv[1],eFaces,lambda,trainCoeffs,trainImgs,mean,imgSize) )
  return -1;

// build ID list from image names
buildIDs(trainImgs,trainIDs);

// get training set size
trainCount = trainImgs.size();

// read test images folder
{
  Size tempSize;
  readImagesFile(argv[2], testImgs, tempSize);
  if ( tempSize != imgSize )
  {
    cout << "Error: Training and Testing images have different dimensions" << endl;
    return -1;
  }
}

// build ID list from image names
buildIDs(testImgs,testIDs);

// get testing set size
testCount = testImgs.size();

testCoeffs.resize(testCount);
trainMatches.resize(testCount);
matchValue.resize(testCount);
topNError.resize(testCount);

int matchCount = 0;

int fp = 0,
    tp = 0,
    intruder = 0,
    nonintruder = 0;

// build list of test set coefficients
for ( i = 0; i < testCount; i++ )
{
  static Mat input;
  input = imread(testImgs[i],0);
  projectFace(input, eFaces, mean, testCoeffs[i]);
  topNMatches(testCoeffs[i], trainCoeffs, lambda, N, trainMatches[i], topNError[i]);

  if ( testIDs[i] < 146 ) // intruder
    intruder++;
  else
    nonintruder++;

  if ( topNError[i][0] < thresh )
    if ( testIDs[i] < 146 ) // false positive
      fp++;
    else  // true positive
      tp++;
}

// output results to terminal
cout << (float)tp/nonintruder << ' ';
```

```
    cout << (float)fp/intruder << endl;

    return 0;
}
```
