ECE 661: Homework 10: Face Recognition and Object Detection

Fall 2014

Solutions

PART I. FACE RECOGNITION

In the first part, the task is to perform face recognition with PCA and LDA and the nearest-neighborhood rule is applied for classification.

1. PCA (Principal Components Analysis)

To perform face recognition using PCA, both train and test images need to be vectorized before any other operations. With the vectorized image, the covariance matrix *C* for train images is first computed.

1.1 Estimate Covariance C of Training Image Set

Given *N* vectorized training images, $\vec{x_i}$ indicates the i^{th} image, where *i=0,1,2,…,N-1*. To compute the covariance matrix *C*, the mean vector with the *N* images can be computed following

$$\vec{m} = \frac{1}{N}\sum_{i=0}^{N-1}\vec{x}_i \,. \tag{1}$$

And *X* is defined as

$$X = \left[\vec{x}_1 - \vec{m}, \vec{x}_2 - \vec{m}, ..., \vec{x}_N - \vec{m}\right]. \tag{2}$$

Then *X* is normalized to achieve illumination invariance by subjecting it to the constraint that $\vec{x_i}^T\vec{x_i} = 1$. To compute covariance *C*, the following equation can be used:

$$C = \frac{1}{N}\sum_{i=0}^{N-1}\left\{XX^T\right\}. \tag{3}$$

The eigenvectors $\vec{w_i}$ of *C* corresponding to the *K* largest eigenvalues will constitute the PCA feature set, denoted as W_K. Different *K*s are tested in this homework. However, the direct Eigen decomposition of *C* can eat up significant computational resources, therefore, a computational trick is applied to compute the eigenvectors $\vec{w_i}$.

1.2 Compute Eigenvectors of C Using a Computational Trick

If $\vec{w}$ represents an eigenvector of *C*, then it must satisfy

$$XX^T\vec{w} = \lambda\vec{w}. \tag{4}$$

Instead of computing the eigenvectors of C, the eigenvectors of X^TX are first computed, denoted as $\vec{u}$. Their relations are shown in equation (5). The Eigen decomposition of X^TX is much easier than C, since X^TX is much smaller than C.

$$X^T X\vec{u} = \lambda\vec{u} \tag{5}$$

Then, to get $\vec{w}$ from $\vec{u}$, equation (5) is multiplied by X at both side, and reformatted as

$$\left(XX^T\right)X\vec{u} = \lambda X\vec{u}\ . \tag{6}$$

Since C equals XX^T, the eigenvectors of C can be computed following

$$\vec{w} = X\vec{u}\ . \tag{7}$$

The eigenvectors are then normalized to unit magnitude, so that the images can be back projected into the eigenspace. The eigenvectors of largest K eigenvalues are used in this homework.

1.3 Back Project both Train and Test Images for Testing

Both train and test images are back projected to the eigenspace following

$$y_i = W^T X_i\ , \tag{8}$$

where $W=[\overrightarrow{w_1}, \overrightarrow{w_2}, \ldots, \overrightarrow{w_K}]$. The trained feature with nearest distance is considered as the target matching for a test feature.

2. LDA (Linear Discriminant Analysis)

The goal of LDA is to find the directions in the underlying vector space that are maximally discriminating between the classes. In this case, two concepts are introduced, the between-class scatter S_B and the within-class scatter S_W along a direction.

2. 1 Definition of between-class scatter S_B and within-class scatter S_W

For multiple classes, the between-class scatter is defined as

$$S_B = \frac{1}{|\mathbb{C}|}\sum_{i=1}^{|\mathbb{C}|}\left(\vec{m}_i - \vec{m}\right)\left(\vec{m}_i - \vec{m}\right)^T \tag{9}$$

where $\mathbb{C}$ is the set of all classes and $\vec{m}$ is the global mean same as equation (1). The within-class scatter is defined as

$$S_W = \frac{1}{|\mathbb{C}|}\sum_{i=1}^{|\mathbb{C}|}\frac{1}{|\mathbb{C}_i|}\sum_{k=1}^{|\mathbb{C}_i|}\left(\vec{x}_k - \vec{m}_i\right)\left(\vec{x}_k - \vec{m}_i\right)^T \tag{10}$$

where the subset of images corresponding to class *i* is denoted as $\mathbb{C}_i$ and $\vec{m}_i$ is the mean image vector for class *i*.

2. 2 Goal of LDA

The goal of LDA is to find LDA eigenvectors *W* that can maximize the Fisher Discriminant Function,

$$J(\vec{w}) = \frac{\vec{w}^T S_B \vec{w}}{\vec{w}^T S_w \vec{w}}. \tag{11}$$

However, instead of directly solving this problem, Yu and Yang's algorithm is applied to find the *W*, since S_W can be singular in this case.

2. 3 Yu and Yang's Algorithm

The first step of Yu and Yang's algorithm is to perform an Eigen decomposition of S_B. The eigenvalues are diagonalized and sorted in descending order. This will also yields a matrix denoted as *V*, representing the corresponding eigenvectors. The same trick used in PCA is also applied here to perform the Eigen decomposition. The first *K* eigenvectors constitute matrix *Y*. Then a matrix *Z* is constructed following

$$Z = YD_B^{-1/2} \tag{12}$$

where D_B is the upper-left $K \times K$ sub-matrix of the diagonalized eigenvalues of S_B,

$$D_B = Y^T S_B Y. \tag{13}$$

To compute D_B, instead of equation (13), we can apply the following equation

$$D_B = \left(Y^T M\right)\left(Y^T M\right)^T \tag{14}$$

where $M = \left[\vec{m}_1 - \vec{m}, \vec{m}_2 - \vec{m}, ..., \vec{m}_{|\mathbb{C}|} - \vec{m}\right]$.

Then, the eigenvectors matrix *U* is computed by Eigen decomposition of $Z^T S_W Z$. This equation can be computed following

$$Z^T S_W Z = \left(Z^T X_W\right)\left(Z^T X_W\right)^T \tag{15}$$

where $X_W = \left[\vec{x}_{11} - \vec{m}_1, \vec{x}_{12} - \vec{m}_1, ..., \vec{x}_{1k} - \vec{m}_1, ..., \vec{x}_{|\mathbb{C}|1} - \vec{m}_{|\mathbb{C}|}, ..., \vec{x}_{|\mathbb{C}|k} - \vec{m}_{|\mathbb{C}|}\right]$. Therefore, the same computational trick in PCA can be performed here as well.

The eigenvectors with largest eigenvalues are discarded, since much inter-class discriminatory information is contained in the smallest eigenvectors of S_W. The matrix of

LDA eigenvectors that maximize the Fisher discriminant function defined in equation (11) can then be calculated following

$$W^T = \hat{U}^T Z^T . \tag{16}$$

At last, *W* need to be normalized as well.

2.4 Back Project both Train and Test Images for Testing

Both train and test images are back projected to the eigenspace following

$$y_i = W^T \left(X_i - \bar{m} \right), \tag{17}$$

where $W=[\overrightarrow{w_1}, \overrightarrow{w_2}, \ldots, \overrightarrow{w_K}]$. The trained feature with nearest distance is considered as the target matching for a test feature.

3. Comparison between PCA and LDA

The accuracy of both approaches is calculated following

$$accuracy(k) = \frac{\text{Number of correctly recognized images}}{\text{Total number of images}} . \tag{18}$$

Figure 1 represents the accuracy for both approaches. From Figure 1, it is observed that before reaching 100% recognition rate, LDA shows relatively higher accuracy than LDA in each number of eigenvectors. Also, LDA uses less number of eigenvectors to achieve 100% recognition accuracy. LDA uses 5 eigenvectors to achieve 100% accuracy, while PCA uses 13. From computational point of view, PCA may be optimal for low dimensional representation, while from discrimination point of view, LDA shows better performance.

There are also some example images for correct recognition and false recognition for both approaches shown in Section 4 from Figure 2 to Figure 5. Both PCA and LDA shows good performance under different illumination environments.

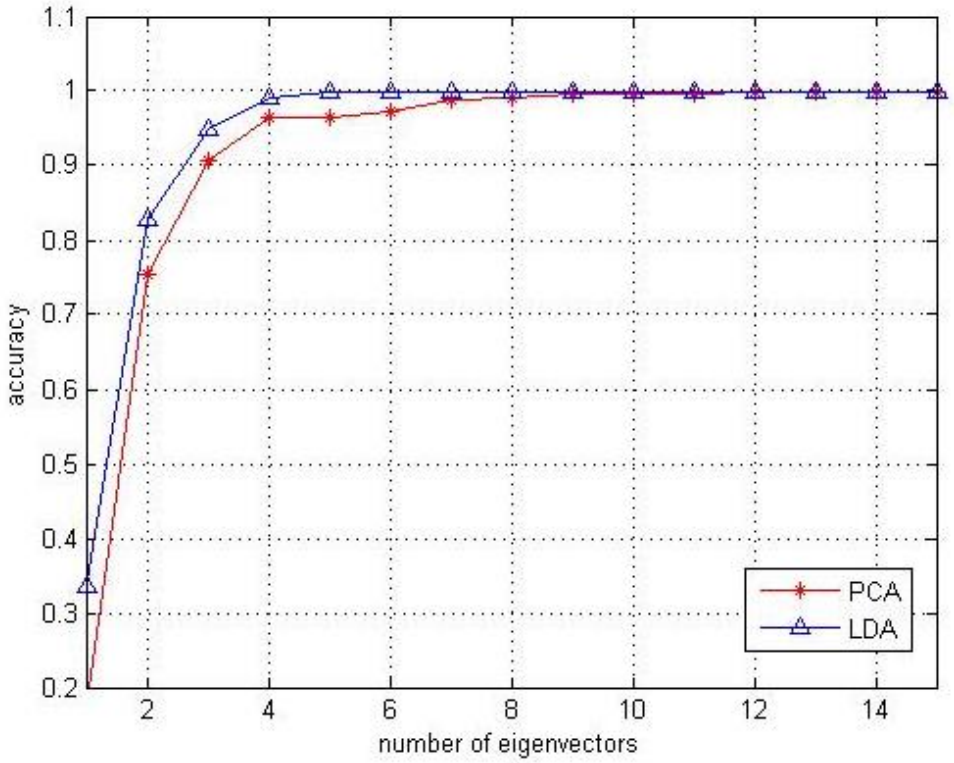

Figure 1 Recognition accuracy of PCA and LDA.

4. Examples of Recognition Results

Figure 2 is an example of correctly recognized face. Figure 2 (a) is the test image and Figure 2 (b) is the recognized image. The distance between this two images is $1.1491e^{-5}$.

(a) (b)

Figure 2 Recognition accuracy of PCA and LDA.

Figure 3 is an example of false recognized face. Figure 3 (a) is the test image, Figure 3 (b) is the false recognized image and Figure 3 (c) is the correctly recognized image using more eigenvectors. The distance between (a) and (b) is the $4.618e^{-9}$ with number of eigenvectors of 1 and the distance between (a) and (c) is *0.0027* with number of eigenvectors of 3.

(a)

(b)

(c)

Figure 3 Recognition accuracy of PCA and LDA.

PART II. OBJECT DETECTION

In the second part, object detection using Cascaded AdaBoost classification is performed. In each stage of the cascade, a target false positive and true detection rate is required to achieve.

1. AdaBoost Classifier

AdaBoost stands for Adaptive Boosting, which means integrating a set of weak classifiers into a strong classifier. To train an AdaBoost classifier, it follows the steps listed below.

1. 1 Haar Feature Extraction

To build weak classifiers, the Haar features are extracted in this homework. Before feature extraction, the original image is required to convert to integral image at first.

An integral image is calculated following

$$I(x,y)=\sum_{x_i\le x,\mathrm{y}_i\le y} i(x_i,y_i), \tag{19}$$

which is the sum of the left-top corner pixels of each pixel.

There are different types of Haar features, in this homework, the edge features are used, expressed as [0,1] and $[1,0]^{\mathrm{T}}$. To get all possible horizontal and vertical features, these two features are extended accordingly. The 1×2 feature is extended to 1×2, 1×4, 1×6, 1×8, 1×10, …, horizontally and 2×2, 3×2, 4×2, 5×2, …, vertically. In this way, there are 166000 features in total.

All the features of each image are expressed as a column vector and all images' features constitute a matrix of feature.

1. 2 Build Weak Classifier

Assume the final strong classifier is built with T weak classifiers, one weak classifier is denoted as h_t. In this homework, one weak classifier is one row of the feature matrix, represented as $f(x)$, where $x = 1, 2, \ldots, 166000$.

To find the best T weak classifiers, all the features are evaluated T times. For instance, to find the t weak classifier, all the features are evaluated one by one.

For one feature $f(x)$, it is then applied to all the training data to find the best threshold that can classify the training data with an optimal classification rate. Before the threshold is calculated, the current feature is first sorted ascendingly according to the feature's value for each example. The threshold is then calculated following

$$e=\min\left(S^{+}+\left(T^{-}-S^{-}\right),\left(S^{-}+\left(T^{+}-S^{+}\right)\right)\right) \tag{20}$$

where T^{+} is the total sum of positive example weights, T^{-} is the total sum of negative example weights, S^{+} is the sum of positive weights below the current example and S^{-} is the sum of negative weights below the current example. The feature with minimum error is used as the threshold is classify all the training images.

The weight for each training image is initially equally assigned and updated in each iteration t finding the best weak classifier. The feature with smallest error is then selected as a weak classifier h_t.

After the t weak classifier is obtained, the weight for each training image is updated following

$$w_{t+1,i} = w_{t,i}\beta_t^{1-e_i} \tag{21}$$

where

$$e_i = \begin{cases} 0 & \text{,correctly classified} \\ 1 & \text{, otherwise} \end{cases} \text{ and } \beta_t = \frac{\varepsilon_t}{1-\varepsilon_t}.$$

The error is calculate following

$$\varepsilon_t = \min \sum_i w_{t,i} \left| h_t(x_i) - y_i \right| \tag{22}$$

where x_i is a training image, and y_i is the label for it.

1. 3 Build Strong Classifier

These T weak classifiers are then constitute a strong classifier. When performing the validation or testing process, this strong classifier can be used as

$$C(x) = \begin{cases} 1 & \text{, } \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2}\sum_{t=1}^{T} \alpha_t \\ 0 & \text{, otherwise} \end{cases} \tag{23}$$

where

$$\alpha_t = \log \frac{1}{\beta_t}.$$

2. Cascaded AdaBoost Classifier

To integrate Adaboost with Cascaded algorithm, the process of building one strong classifier using AdaBoost is repeated for several cascaded stages.

2. 1 One stage of Cascaded AdaBoost

In the beginning of each stage, the features used for this stage are updated according to the false recognized negative training images. Only those correctly recognized negative images and all the positive images are used in this stage. Then, a strong classifier is constructed following AdaBoost process. Instead of integrating T weak classifiers as a strong classifier, an additional condition is applied to determine the number of weak classifiers used. In this case, if the false positive rate under a certain strong classifier is smaller than 0.5, this strong classifier is considered as good enough and this stage is completed then.

The false positive rate is calculated following

$$\text{false positive rate} = \frac{\text{Number of misclassified negative images}}{\text{Number of negative images}} . \quad (24)$$

3. Testing Results

Besides false positive rate, false negative rate is also calculated for each cascaded stage,

$$\text{false negative rate} = \frac{\text{Number of misclassified positive images}}{\text{Number of positive images}} . \quad (25)$$

Test images are validated using classifiers constructed from different cascaded stages and the false positive rate and false negative rate is shown in Figure 4.

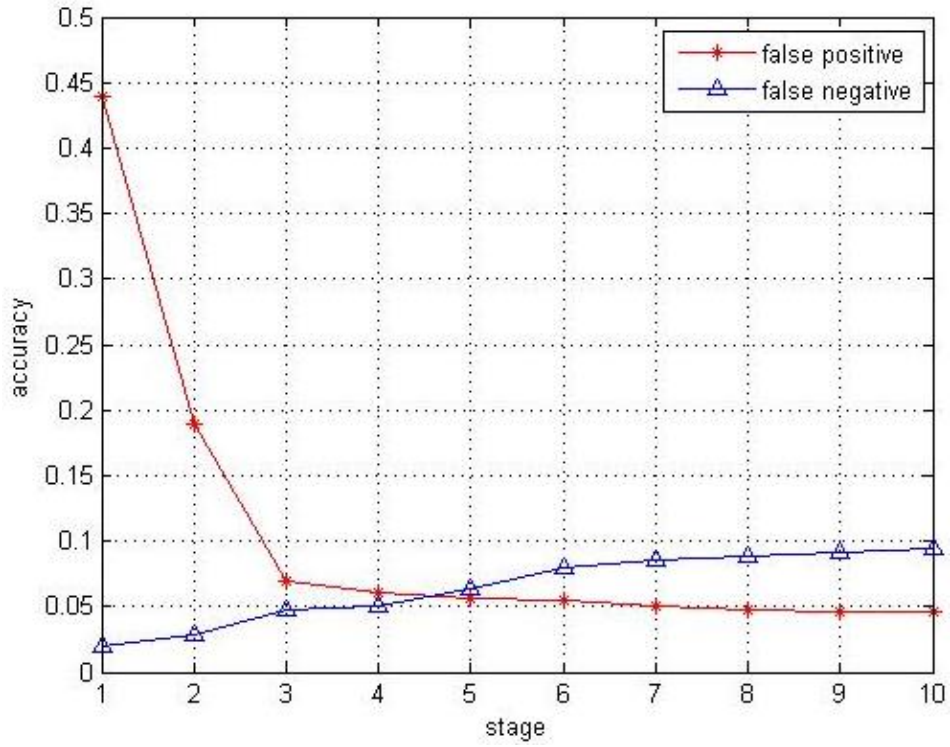

Figure 4 Accuracy for cascaded AdaBoost Classification.

4. Training Parameters and Results

The number of weak classifiers used in each cascaded stage is summarized in Table I.

Table I Number of weak classifiers for each cascade stage.

Stage	1	2	3	4	5	6	7	8	9	10
Number of weak classifiers	10	13	24	22	11	6	4	4	3	3

The Accumulative false positive rate for the cascade stages in training process is shown in Figure 5.

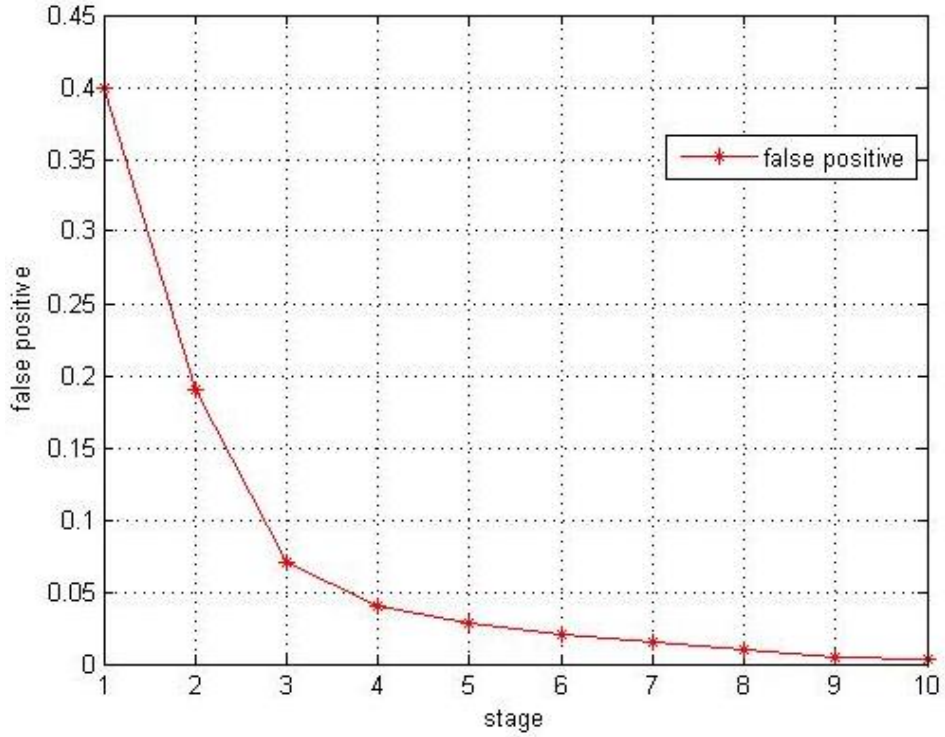

Figure 5 Accuracy for cascaded AdaBoost Classification.

Appendix A. Matlab® Code

I. Face Recognition with PCA and LDA

1. PCA approach

```
% Face recognition using PCA
% ECE661: Computer vision
% Fall 2014
% Ting Zhang
% zhan1013@purdue.edu

clc
clear all

% define parameters
Nperson = 30;
Ntrials = 21;
trainPath = 'ECE661_2014_hw10_DB1/train/';
testPath = 'ECE661_2014_hw10_DB1/test/';

% load training images
[trainImg, ~, ~] = loadImages(trainPath, Nperson, Ntrials);

% load testing images
[testImg, ~, ~] = loadImages(testPath, Nperson, Ntrials);

% load trained w
w_pca = load('w_pca');
w = w_pca.w_pca.w;
Neigen = w_pca.w_pca.Neigen;

% test using different number of eigenvectors, from small to large
accuracy = zeros(1, Neigen);

for i = 1:Neigen
    % get first i eigenvectors
    partEigen = w(:,1:i);

    % project training images
    trainProjected = zeros(i, Nperson*Ntrials);
    for j = 1:Nperson*Ntrials
        trainProjected(:,j) = partEigen' * trainImg(:,j);
    end

    % project testing images
    testProjected = zeros(i, Nperson*Ntrials);
    for j = 1:Nperson*Ntrials
        testProjected(:,j) = partEigen' * testImg(:,j);
    end

    % do recognition for each test image
    for j = 1:Nperson*Ntrials
```

```
        % compute all distance with the trained image
        distance = zeros(1,Nperson*Ntrials);
        for k = 1:Nperson*Ntrials
            distance(1,k) = norm(testProjected(:,j)-trainProjected(:,k))^2;
        end

        % get nearest as match
        [~,matchIdx] = min(distance);

        % determine accuracy for each test image
        % get test person id
        testPerson = floor((j-1)/Ntrials) + 1;
        % get matched person id
        matchPerson = floor((matchIdx-1)/Ntrials) + 1;
        if testPerson == matchPerson
            accuracy(1,i) = accuracy(1,i) + 1;
        end
        if testPerson ~= matchPerson
            j;
        end
    end % end of recognition for each test image
end % end of test for different number of eigen values

% compute accuracy
accuracy = accuracy / (Nperson*Ntrials);
save('PCA_accuracy.dat', 'accuracy', '-ASCII');

% plot accuracy
idx = 1:Neigen;
plot(idx(1:25),accuracy(1:25),'r*-');
axis([1 25 0.84 1]);

function [ normW, Neigen ] = myPCA( imgVec )
%myPCA Summary of this function goes here
%   Detailed explanation goes here

[~,col] = size(imgVec);

%compute covariance matrix C = XXt
%using XtX instead
[V,D] = eig(imgVec'*imgVec);
%sort eigenvalues from largest to smallest
eigenValue = diag(D);
[~,idx] = sort(-1.0 .* eigenValue);
eigenValue = eigenValue(idx);
V = V(:,idx);

%for each image, get number of eigenvectors with eigenvalue greater than 1
Neigen = 0;
for i = 1:col
    if eigenValue(i) > 1
        Neigen = Neigen + 1;
    end
end
```

```
%compute w = Xu
w = imgVec * V;

%normalize w
[row,col] = size(w);
normW = zeros(row,col);
for i = 1:col
    normW(:,i) = w(:,i) / norm(w(:,i));
end

end


function trainPCA(  )
%trainPCA Summary of this function goes here
%   Detailed explanation goes here

filePath = 'ECE661_2014_hw10_DB1/train/';
[imgVec, ~, ~] = loadImages(filePath, 30, 21);
[w, Neigen] = myPCA(imgVec);

w_pca.w = w;
w_pca.Neigen = Neigen;
save('w_pca','w_pca');

end

function [ normImgVec, imgVec, meanImg ] = loadImages( filePath, Nperson,
Ntrial )
%loadImages Summary of this function goes here
%   Detailed explanation goes here

%get image size
img = imread([filePath,'01_01.png']);
imgGray = rgb2gray(img);
[row,col] = size(imgGray);
%define output vectors
imgVec = zeros(row*col,Nperson*Ntrial); %each column is an image

%load images into 1D vectors
for i = 1:Nperson
    for j = 1:Ntrial
        img  =  imread([filePath,num2str2digit(i),'_',num2str2digit(j),'.png']);
        %figure;
        %imshow(img);
        imgGray = rgb2gray(img);
        [row,col] = size(imgGray);
        oneVec = reshape(imgGray',row*col,1);
        imgVec(:,(i-1)*Ntrial+j) = oneVec;
    end
end

%compute mean of all images
meanImg = mean(imgVec,2);
```

```
%normalize images using the mean
normImgVec = zeros(row*col,Nperson*Ntrial);
for i = 1:Nperson*Ntrial
    normImgVec(:,i) = (imgVec(:,i) - meanImg) / norm(imgVec(:,i) - meanImg);
end

end

%% convert num to 2 digit string
function str = num2str2digit(num)
if num<10
    str = ['0',num2str(num)];
else
    str = num2str(num);
end
end
```

2. LDA approach

```
% Face recognition using LDA
% ECE661: Computer vision
% Fall 2014
% Ting Zhang
% zhan1013@purdue.edu

clc
clear all

% define parameters
Nperson = 30;
Ntrials = 21;
trainPath = 'ECE661_2014_hw10_DB1/train/';
testPath = 'ECE661_2014_hw10_DB1/test/';

% load training images
[~,trainImg, meanTrain] = loadImages(trainPath, Nperson, Ntrials);

% load testing images
[~,testImg, meanTest] = loadImages(testPath, Nperson, Ntrials);

% get trained data
[vecU, Z] = myLDA(trainImg,meanTrain,Nperson,Ntrials);

% test using different number of eigenvalues
Neigen = 30;
accuracy = zeros(1, Neigen);

for i = 1:Neigen
    % compute part eigenvector U
    partVecU = vecU(:,1:i);
    W = Z * partVecU;

    % normalize W
```

```
    for j = 1:i
        W(:,j) = W(:,j) / norm(W(:,j));
    end

    % project training images
    trainProjected = zeros(i, Nperson*Ntrials);
    for j = 1:Nperson*Ntrials
        trainProjected(:,j) = W' * (trainImg(:,j)-meanTrain);
    end

    % project testing images
    testProjected = zeros(i, Nperson*Ntrials);
    for j = 1:Nperson*Ntrials
        testProjected(:,j) = W' * (testImg(:,j)-meanTest);
    end

    % do recognition for each test image
    for j = 1:Nperson*Ntrials

        % compute all distance with the trained image
        distance = zeros(1,Nperson*Ntrials);
        for k = 1:Nperson*Ntrials
            distance(1,k) = norm(testProjected(:,j)-trainProjected(:,k))^2;
        end

        % get nearest as match
        [~,matchIdx] = min(distance);

        % determine accuracy for each test image
        % get test person id
        testPerson = floor((j-1)/Ntrials) + 1;
        % get matched person id
        matchPerson = floor((matchIdx-1)/Ntrials) + 1;
        if testPerson == matchPerson
            accuracy(1,i) = accuracy(1,i) + 1;
        end
    end % end of recognition for each test image

end

% compute accuracy
accuracy = accuracy / (Nperson*Ntrials);
save('LDA_accuracy.dat', 'accuracy', '-ASCII');

% plot accuracy
idx = 1:Neigen;
plot(idx(1:25),accuracy(1:25),'r*-');
axis([1 25 0.84 1]);

function [ vecU, Z ] = myLDA( imgVec, mean, Nperson, Ntrials )
%myLDA Summary of this function goes here
%   Detailed explanation goes here

% define image size
imgSize = 128*128;
```

```
% compute mean for each class
sumImg = zeros(imgSize,Nperson*Ntrials);

for i = 1:Nperson*Ntrials
    classIdx = floor((i-1)/Ntrials) + 1;
    sumImg(:,classIdx) = sumImg(:,classIdx) + imgVec(:,i);
end
meani = sumImg / Ntrials;

% build mi-m
meani_m = zeros(imgSize, Nperson);
for i = 1:Nperson
    meani_m(:,i) = meani(:,i) - mean;
end

% compute SB
SB = meani_m * meani_m';
% ensure SB is not singular
[vecSB,valSB] = eig(meani_m' * meani_m);
[~,idx] = sort(-1 .* diag(valSB));
V = meani_m * vecSB;

Nfeatures = 30;
% build Y, DB, Z
Y = V(:,1:Nfeatures);
DB = Y' * meani_m * meani_m' * Y;
Z = Y * DB^(-0.5);

% build xk-mi
xk_meani = zeros(imgSize, Ntrials);
for i = 1:Nperson*Ntrials
    classIdx = floor((i-1)/Ntrials) + 1;
    xk_meani(:,i) = imgVec(:,i) - meani(:,classIdx);
end
% compute Zt*Sw*Z = Z' * (xk-meani) * (xk-meani)' * Z
Zt_xk_meani = Z' * xk_meani;
% eigendecompostion to get U
[vecU,valU] = eig(Zt_xk_meani*Zt_xk_meani');
% diagnolize eigenvalues of U
DU = diag(valU);

end
```

II. Object Detection with Cascaded AdaBoost Classifier

1. Feature Extraction

```
function [ features, Npos, Nneg ] = getHaar( filePath )
%getHaar Summary of this function goes here
%   Detailed explanation goes here

% load images
row = 20;
col = 40;
posFilePath = [filePath 'positive/'];
negFilePath = [filePath 'negative/'];
posImg = loadImagesAdaBoost(posFilePath, row, col);
negImg = loadImagesAdaBoost(negFilePath, row, col);

% get total number of images
Nimg = size(posImg,3) + size(negImg,3);
Npos = size(posImg,3);
Nneg = size(negImg,3);

Nfeatures = 166000;
features = zeros(Nfeatures, Nimg);
for i = 1:Nimg
    if i <= size(posImg,3)
        % convert to integral image
        intImg = zeros(row+1,col+1);
        intImg(2:row+1,2:col+1) = cumsum(cumsum(posImg(:,:,i)),2);
        % compute features
        features(:,i) = computeFeature(intImg);
    else
        % convert to integral image
        intImg = zeros(row+1,col+1);
        intImg(2:row+1,2:col+1) = cumsum(cumsum(negImg(:,:,i-
size(posImg,3))),2);
        % compute features
        features(:,i) = computeFeature(intImg);
    end
end

features_adaboost.features = features;
features_adaboost.Npos = Npos;
features_adaboost.Nneg = Nneg;

save('features_adaboost_test.mat', 'features_adaboost', '-mat', '-v7.3');
%save('features_adaboost_train.mat', 'features_adaboost', '-mat', '-v7.3');
end

%% load images
function imgs = loadImagesAdaBoost(filePath, row, col)

% get images in 'filePath'
files = dir([filePath '*.png']);
imgs = zeros(row,col,length(files));
```

```
for i = 1: length(files)
    img = imread([filePath files(i).name]);
    imgGray = double(rgb2gray(img));
    imgs(:,:,i) = imgGray;
end

end

%% compute Haar features
function feature = computeFeature(I, row, col)

feature = zeros(166000,1);

%extract horizontal feature
cnt = 1;
for h = 1:20
    for w = 1:20
        for i = 1:21-h
            for j = 1:41-2*w
                rect1=[i,j,w,h];
                rect2=[i,j+w,w,h];
                feature(cnt)=sumRect(I, rect2)-sumRect(I, rect1);
                cnt=cnt+1;
            end
        end
    end
end

for h = 1:10
    for w = 1:40
        for i = 1:21-2*h
            for j = 1:41-w
                rect1=[i,j,w,h];
                rect2=[i+h,j,w,h];
                feature(cnt)=sumRect(I, rect1)-sumRect(I, rect2);
                cnt=cnt+1;
            end
        end
    end
end


end

%%
function [rectsum] = sumRect(I, rect_four)

% given four corner points in the integral image
% to calculate the sum of pixels inside the rectangular.

row_start = rect_four(1);
col_start = rect_four(2);
width = rect_four(3);
height = rect_four(4);
```

```
one = I(row_start, col_start);
two = I(row_start, col_start+width);
three = I(row_start+height, col_start);
four = I(row_start+height, col_start+width);

rectsum = four + one - (two + three);
end
```

2. Training Process

```
% Object detection training process using Cascaded AdaBoost Classification
% ECE661: Computer vision
% Fall 2014
% Ting Zhang
% zhan1013@purdue.edu

clc
clear all

% get features
featureFile = load('features_adaboost.mat');
features = featureFile.features_adaboost.features;
Npos = featureFile.features_adaboost.Npos;
Nneg = featureFile.features_adaboost.Nneg;


S = 20;
idx = 1: Npos+Nneg;

for i = 1:S
    idx = myCascade(features, Npos, idx, i);

    % stop is all negatives are detection correctly
    if length(idx)==Npos
        break;
    end
end

function [idx] = myCascade(featuresAll, Npos, idxPrevious, stage)
%myCascade Summary of this function goes here
%   Detailed explanation goes here

% update negative number
Nneg = length(idxPrevious) - Npos;
Ntotal = Npos + Nneg;

% update features
features = featuresAll(:,idxPrevious);

% initialize weights to equally assigned
weight = zeros(Ntotal,1);
% initialize labels for posiive and negative samples
```

```
label = zeros(Ntotal,1);
for i = 1:Ntotal
    if i <= Npos
        weight(i) = 0.5 / Npos;
        label(i) = 1;
    else
        weight(i) = 0.5 / Nneg;
    end
end

%% adaboost process
T = 40;
strongClaResult = zeros(Ntotal,1);
alpha = zeros(T,1);
ht = zeros(4,T);
hResult = zeros(Ntotal,T);

for t = 1:T
    % normalize  weights
    weight = weight ./ sum(weight);
    % get the best weak classifier and the detection result
    h = getClassifier(features, weight, label, Npos);
    % store result
    ht(1,t) = h.currentMin;
    ht(2,t) = h.p;
    ht(3,t) = h.featureIdx;
    ht(4,t) = h.theta;
    hResult(:,t) = h.bestResult;
    % get min error
    err = h.currentMin;
    % get trust fact alphat = 0.5 * ln((1-et)/et)
    alpha(t) = log((1-err)/err);

    % update weight
    weight = weight .* (err/(1-err)) .^ (1-xor(label,h.bestResult));

    % strong classifier
    strongCla = hResult(:,1:t) * alpha(1:t,:);
    threshold = min(strongCla(1:Npos));

    for i = 1:Ntotal
        if strongCla(i) >= threshold
            strongClaResult(i) = 1;
        else
            strongClaResult(i) = 0;
        end
    end

    % compute positive accuracy
    posAccuracy(t) = sum(strongClaResult(1:Npos)) / Npos;
    % compute negative accuracy
    negAccuracy(t) = sum(strongClaResult(Npos+1:end)) / Nneg;

    if posAccuracy(t)==1 && negAccuracy(t) <= 0.5
        break;
```

```
    end

    fprintf('t = %d\n', t);
end

%% update for next cascaded iteration

% sort negative, if there is false deteciton, there will be 1 at the end
[sortedNeg, idxNeg] = sort(strongClaResult(Npos+1:end));
% get false detection negative index
for i = 1:Nneg
    if sortedNeg(i) > 0
        idxNeg = idxNeg(i:end);
        break;
    end
end

% get sample index for next cascaded iteration
idx = [1:Npos, Npos+idxNeg'];

% save trained data
save(['strongCla_',num2str(stage),'.mat'],'strongCla','-mat', '-v7.3');
save(['negAccuracy_',num2str(stage),'.mat'],'negAccuracy','-mat', '-v7.3');
% polarity, theta  for each classifier
save(['ht_',num2str(stage),'.mat'],'ht','-mat', '-v7.3');
% alpha for each weak classifier
save(['alpha_',num2str(stage),'.mat'],'alpha','-mat', '-v7.3');
% indices for classifier h's feature
save(['idxForNext',num2str(stage),'.mat'],'idx','-mat', '-v7.3');
% threshold for whole strong classifier --- may not be used
save(['threshold_',num2str(stage),'.mat'],'threshold','-mat', '-v7.3');
end


function h = getClassifier(features, weight, label, Npos)
%getClassifier Summary of this function goes here
%   Detailed explanation goes here

% define parameters
Nfeatures = size(features,1);
Nimgs = size(features,2);
h.currentMin = inf;

tPos = repmat(sum(weight(1:Npos,1)), Nimgs,1);
tNeg = repmat(sum(weight(Npos+1:Nimgs,1)), Nimgs,1);

% search each feature as a classifier
for i = 1: Nfeatures
    % get one feature for all images
    oneFeature = features(i,:);
    % sort feature to thresh for postive and negative
    [sortedFeature, sortedIdx] = sort(oneFeature, 'ascend');
    % sort weights and labels
    sortedWeight = weight(sortedIdx);
    sortedLabel = label(sortedIdx);
```

```
    % select threshold
    sPos = cumsum(sortedWeight .* sortedLabel);
    sNeg = cumsum(sortedWeight) - sPos;
    errPos = sPos + (tNeg - sNeg);
    errNeg = sNeg + (tPos - sPos);

    % choose the threshold with small error
    allErrMin = min(errPos, errNeg);
    [errMin, idxMin] = min(allErrMin);

    % result
    result = zeros(Nimgs,1);
    if errPos(idxMin) <= errNeg(idxMin)
        p = -1;
        result(idxMin+1:end) = 1;
        result(sortedIdx) = result;
    else
        p = 1;
        result(1:idxMin) = 1;
        result(sortedIdx) = result;
    end

    % get best parameters
    if errMin < h.currentMin
        h.currentMin = errMin;
        if idxMin==1
            h.theta = sortedFeature(1) - 0.5;
        elseif idxMin==Nfeatures;
            h.theta = sortedFeature(Nfeatures) + 0.5;
        else
            h.theta = (sortedFeature(idxMin)+sortedFeature(idxMin-1))/2;
        end

        h.p = p;
        h.featureIdx = i;
        h.bestResult = result;
    end

end % end of search each feature

end
```

3. Testing Process

```
% Object detection using Cascaded AdaBoost Classification
% ECE661: Computer vision
% Fall 2014
% Ting Zhang
% zhan1013@purdue.edu

clc
clear all

% get test features
```

```
testFeatureFile = load('features_adaboost_test.mat');
testFeatures = testFeatureFile.features_adaboost.features;
Npos = testFeatureFile.features_adaboost.Npos;
Nneg = testFeatureFile.features_adaboost.Nneg;

S = 10;

% for computing accuracy for each stage
fp = zeros(S,1);
fn = zeros(S,1);

% test for each stage
for i = 1:S
    fprintf('Test stage: %d\n', i);

    % load classifier infor
    htFile = load(['ht_' num2str(i) '.mat']);
    ht = htFile.ht;
    alphaFile = load(['alpha_' num2str(i) '.mat']);
    alpha = alphaFile.alpha;
    strongThFile = load(['threshold_' num2str(i) '.mat']);
    strongTh = strongThFile.threshold;

    % get t
    t = 0;
    for j = 1:size(ht,2)
        if ht(:,j)==0
            break;
        end
        t = t + 1;
    end

    % build polarity
    p = ht(2,1:t);
    % build each weak classifier threshold
    theta = ht(4,1:t);
    % build selected features
    fIdx = ht(3,1:t);
    % build alpha
    alpha = alpha(1:t,1);

    % do classification
    result = adaBoostClassify(testFeatures, alpha, p, theta, fIdx, t,
strongTh);

    % compute accuracy
    fn(i) = (Npos - sum(result(1:Npos))) / Npos;
    fp(i) = sum(result(Npos+1:end)) / Nneg;
end

% plot result of training
noStage = 10;
fp_rate = zeros(noStage,1);
for i=1:noStage
    fp_rate = fp(1:i);
```

```
    for j=2:i
        if fp_rate(j)==0
            fp_rate(j)=fp_rate(j-1);
            break;
        end
    end
    falsepos_acc = cumprod(fp_rate);
end
fn_rate = zeros(noStage,1);
for i=1:noStage
    fn_rate = fn(1:i);
    for j=2:i
        if fn_rate(j)==0
            fn_rate(j)=fn_rate(j-1);
            break;
        end
    end
    falseneg_acc = cumprod(fn_rate);
end

% save result
save('false_negative_rate.mat','fn','-mat', '-v7.3');
save('false_positive_rate.mat','fp','-mat', '-v7.3');

save('false_negative_rate_1.mat','falseneg_acc','-mat', '-v7.3');
save('false_positive_rate_1.mat','falsepos_acc','-mat', '-v7.3');
% plot result

function [ result ] = adaBoostClassify( featuresAll, alpha, p, theta, fIdx, T,
strongTh )
%adaBoostClassify Summary of this function goes here
%   Detailed explanation goes here

% get number of test images
Nimgs = size(featuresAll,2);

% result for each weak classifier
weakResult = zeros(Nimgs,T);

% classify using every weak classiier
for t = 1:T
    % get classifier feature
    feature = featuresAll(fIdx(t),:);

    % do classification for each test image
    for i = 1:Nimgs
        if p(t)*feature(i) <= p(t)*theta(t)
            weakResult(i,t) = 1;
        end
    end
end

% build strong classifier
strongCla = weakResult(:,1:T) * alpha(1:T,:);
```

```
% compute strong classifier thershold
strongThreshold = 0.5 * sum(alpha(1:T,1));

% get final classification result
result = zeros(Nimgs,1);

for i = 1:Nimgs
    %if strongCla(i) >= strongTh
    if strongCla(i) >= strongThreshold
        result(i) = 1;
    end
end

end
```