ECE661 Computer Vision HW 11

Rih-Teng Wu

Email: wu952@purdue.edu

1. Introduction

In the first part of this homework, we use principal component analysis (PCA) and linear discriminant analysis (LDA) to perform face recognition. The PCA and LDA is employed to project the high dimension data to their eigen-space, then the nearest neighbor algorithm is used to classify the data.

In the second part of this homework, we use AdaBoost algorithm to perform car recognition. The cascaded AdaBoost is employed to reduce the False-Positive rate.

2. Principal Component Analysis (PCA)

To find the projection matrix that can project our high-dimension data to a low-dimension representation, PCA is introduced to perform this task. The procedure of PCA is described as follows.

Step 1: Vectorize each training image of size 128x128 to a 16384x1 vector $\vec{x}_i$, normalize each $\vec{x}_i$ to make it illuminant-invariant. Use gray-scale images.

Step 2: Calculate the global training mean:

$$\vec{m} = \frac{1}{N}\sum_i \vec{x}_i$$

Step 3: Form a matrix $X = [\vec{x}_1 - \vec{m} \quad \vec{x}_2 - \vec{m} \quad \dots\dots \quad \vec{x}_N - \vec{m}]$.

Step 4: Instead of directly calculating the eigen-vectors $\vec{w}$ of the covariance matrix XX^T, we calculate the eigen-vectors of matrix X^TX. Let $\vec{u}$ denote the eigen-vectors of matrix X^TX. To reconstruct the eigen-vectors $\vec{w}$ from $\vec{u}$, we use the following equation:

$$\vec{w} = X\vec{u}$$

Step 5: Normalize each eigen-vector in $\vec{w}$.

Step 6: Select the largest P eigen-vectors from the normalized $\vec{w}$. The projection matrix W_P is given as follows.

$$W_P = [\vec{w}_1 \quad \vec{w}_2 \quad \ldots\ldots \quad \vec{w}_P]$$

Step 7: Project all the training samples using the following equation:

$$\vec{y}_i = {W_P}^T(\vec{x}_i - \vec{m})$$

Step 8: Given a new test image, we first vectorize the test image, then project it using the equation given in Step 7. The projected vector is then classified based on the nearest neighbor within all the projected training vectors.

3. Linear Discriminant Analysis (LDA)

The objective of LDA is to find the eigen-vectors $\vec{w}_j$ that maximize the Fisher Discriminant Function:

$$J(\vec{w}_j) = \frac{{\vec{w}_j}^T S_B \vec{w}_j}{{\vec{w}_j}^T S_W \vec{w}_j}$$

Where S_B is the between-class scatter, S_W is the within-class scatter.

However, in most cases S_W is singular. Therefore, we need to use Yu and Yang's algorithm to find $\vec{w}_j$, and use $\vec{w}_j$ to form the projection matrix. The procedure of Yu and Yang's algorithm is described as follows.

Step 1: Vectorize each training image of size 128x128 to a 16384x1 vector $\vec{x}_i$, normalize each $\vec{x}_i$ to make it illuminant-invariant. Use gray-scale images.

Step 2: Calculate the global training mean:

$$\vec{m} = \frac{1}{N}\sum_i \vec{x}_i$$

Step 3: Calculate the class mean:

$$\vec{m}_k = \frac{1}{\|C_k\|} \sum_{i \epsilon C_k} \vec{x}_i$$

where C_k means the class of training images with identity k, $k = 1 \sim C$.

Step 4: Form matrix M:

$$M = [\vec{m}_1 - \vec{m} \quad \vec{m}_2 - \vec{m} \quad \ldots\ldots \quad \vec{m}_C - \vec{m}]$$

Step 5: Instead of directly calculating the eigen-vectors of $S_B = MM^T/C$, we calculate the eigen-vectors of matrix $M^T M/C$. Let $\vec{u}$ be the eigen-vectors of $M^T M/C$ in descending order, we reconstruct the eigen-vectors $\vec{V}$ of $S_B = MM^T/C$ by the following equation:

$$\vec{V} = M\vec{u}$$

Step 6: Form matrix $Y = [\vec{V}_1 \quad \vec{V}_2 \quad \ldots\ldots \quad \vec{V}_C]$, and form D_B which is the eigen-value matrix of S_B. (The eigen-values of MM^T/C and $M^T M/C$ are the same, except that the former have additional zeros.)

Step 7: Compute $Z = YD_B{}^{-1/2}$.

Step 8: Compute the eigen-vectors of $Z^T S_W Z$. We can use the same computation trick as described previously since $Z^T S_W Z$ has the following form:

$$Z^T S_W Z = (Z^T X_W)(Z^T X_W)^T$$

Where $X_W = [\vec{x}_{11} - \vec{m}_1, \vec{x}_{12} - \vec{m}_1, \ldots, \vec{x}_{1k} - \vec{m}_1 \ldots, \vec{x}_{C1} - \vec{m}_C, \ldots, \vec{x}_{Ck} - \vec{m}_C]$

Step 9: Organize the eigen-vectors U of $Z^T S_W Z$ in ascending order. Select the smallest P eigen-vectors from U. Denote the eigen-vector matrix after selection as $\hat{U}$. Then the projection matrix W_P is given as follows.

$$W_P^T = \hat{U}^T Z^T$$

Step 10: Normalize each eigen-vector in W_P.

Step 11: Project all the training samples using the following equation:

$$\vec{y}_i = W_P{}^T(\vec{x}_i - \vec{m})$$

Step 12: Given a new test image, we first vectorize the test image, then project it using the equation given in Step 11. The projected vector is then classified based on the nearest neighbor within all the projected training vectors.

4. Performance Evaluation and Parameter Setting for PCA and LDA

The performance evaluation of PCA and LDA is conducted using the following equation:

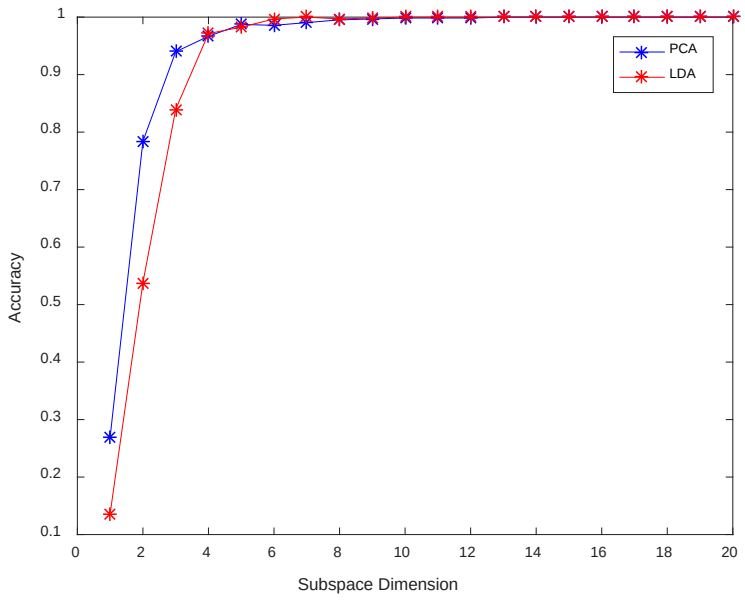
$$accuracy = \frac{\#\text{ of test images correctly classified}}{total\ \#\text{ of test images}}$$

The parameter setting for PCA and LDA is tabulated as follows.

Parameters	Description	Setting
p	The subspace dimensionality, which is the number of eigen-vectors used to project image	1~20

5. PCA and LDA Result and Observations

Result plot:



Result tabulation:

Subspace dimension	PCA accuracy	LDA accuracy
1	0.270	0.137
2	0.784	0.538
3	0.940	0.840
4	0.967	0.971
5	0.987	0.983
6	0.986	0.997
7	0.990	1.000
8	0.995	0.997
9	0.997	0.998
10	0.998	1.000
11	0.998	1.000
12	0.998	1.000
13	1.000	1.000
14	1.000	1.000
15	1.000	1.000
16	1.000	1.000
17	1.000	1.000
18	1.000	1.000
19	1.000	1.000
20	1.000	1.000

According to the above results, we have the following observations:

a. LDA is not always better than PCA.

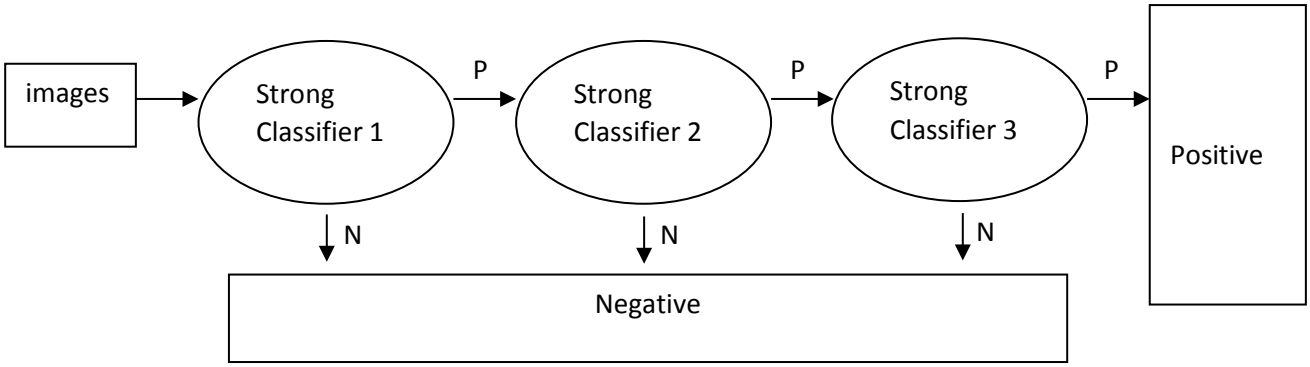
b. When subspace dimensionality is below 4, LDA performs worse than PCA. When subspace dimensionality is higher than 5, LDA achieves better accuracy than PCA.

c. LDA achieves 100% accuracy at subspace dimensionality of 7, while PCA achieves 100% accuracy at subspace dimensionality of 13.

PART II: Object Detection with Cascaded AdaBoost Classifier

1. Main Concept of Cascaded AdaBoost

The main concept of cascaded AdaBoost classifier is to design several strong classifiers, each strong classifier consists of multiple weak classifiers. By selecting the targeted false-positive rate and the true detection rate of each strong classifier, the final combined classifier can achieve a desirable low false-positive rate while keeping the true detection rate being acceptable. The figure below shows the configuration of the cascaded AdaBoost classifier.



2. Feature Generation

In AdaBoost algorithm, the weak classifier is simply built by the thresholding of feature. In this homework, we generate the Haar-like edge features, which has the following form:

In mathematical representation, we denote horizontal filter and vertical filter as $[0,1]$ and $[1,0]^T$, respectively. To reduce computation burden, we use horizontal filters of size 1x2, 1x4, …, 1x40 sliding over the whole image to generate features. Also, we use vertical filters of size 2x2, 4x2, …, 20x2 sliding over the whole image. The feature calculation utilizes the integral image, which reduces computation efforts as well. As a result, there is a total of 11,900 features employed in this homework.

3. AdaBoost Classifier

(a) Find the best weak classifier

The procedure of finding the best weak classifier (feature) is described as follows.

Step 1: For each feature, sorting the feature value in ascending order. The error for selecting the feature value of the current example as the threshold is:

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

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, S^- is the sum of negative weights below the current example. The feature which gives us the minimum error is selected as the best weak classifier.

Step 2: The weights for each training image is initially equal assigned. After the t weak classifier is obtained, update the weights using the following equation:

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

$$\beta_t = \frac{\varepsilon_t}{1-\varepsilon_t}$$

Where $e_i = 0$ if the sample is correctly classified, $e_i = 1$ if the sample is misclassified. ε_t is the weighted error.

Step 3: The t weak classifier is defined as:

$$\mathrm{h}(\mathrm{x}, \mathrm{f}, \mathrm{p}, \theta) = \begin{cases} 1, \text{if } \mathrm{pf}(\mathrm{x}) < \mathrm{p}\theta \\ 0, \text{otherwise} \end{cases}$$

where x is image, f is feature, f(x) is feature value, θ is the threshold and p is polarity sign determined by $e = \min\left(S^+ + (\mathrm{T}^- - \mathrm{S}^-), S^- + (\mathrm{T}^+ - \mathrm{S}^+)\right)$. If $S^+ + (\mathrm{T}^- - \mathrm{S}^-)$ is less than $S^- + (\mathrm{T}^+ - \mathrm{S}^+)$, then $\mathrm{p} = -1$. Otherwise, $\mathrm{p} = 1$.

(b) Build Strong Classifier

The procedure of building the strong classifier is described as follows.

Step 1: Given n training images x_i , label the positive examples as 1 and the negative examples as 0.

Step 2: Initial image weights $w_{1,i} = \frac{1}{2M}, \frac{1}{2L}$ for negative and positive image respectively, where M and L are the number of negative and positive images respectively.

Step 3: For iteration $t = 1 \sim T$,

I. Normalize weight $w_{t,i} = w_{t,i} / \sum_i w_{t,i}$
II. For all the feature f, find the best weak classifier $h_t(x) = h(x, f, p, \theta)$ with the minimum weighted error ε_t.
III. Compute $\beta_t = \varepsilon_t / 1 - \varepsilon_t$, $\alpha_t = \log(1/\beta_t)$
IV. Update weights $w_{t+1,i} = w_{t,i} \beta_t^{1-e_i}$

Step 4: The final strong classifier is:

$$C(x) = \begin{cases} 1, \sum_t \alpha_t h_t(x) \geq \text{threshold} \\ 0, \text{otherwise} \end{cases}$$

Step 5: The stopping criterion is determined by the targeted false-positive rate and the true detection rate for each strong classifier. In this homework, the targeted true detection rate during training is 1, and the targeted false-positive rate during training is 0.5.

It should be noted that the threshold for the strong classifier can be adjusted based on our objective. Since we want our classifier to pass all the positive examples during training, the threshold is set to be the minimum value of $\sum_{t=1}^{T} \alpha_t h_t(x)$. During testing, we set the threshold to be $0.5 \times \sum_{t=1}^{T} \alpha_t$.

4. Performance Evaluation and Parameter Setting for AdaBoost

The performance evaluation of AdaBoost is conducted using the false-positive rate (FP) and the false-negative rate (FN):

$$FP = \frac{\#\text{ of misclassfied negative test images}}{total\ \#\text{ of negative test images}}$$

$$FN = \frac{\#\text{ of misclassfied positive test images}}{total\ \#\text{ of positive test images}}$$

The parameter setting for AdaBoost is tabulated as follows.

Parameters	Description	Setting
threshold_positive	The acceptable positive detection rate	1
threshold_FP	The acceptable False-Positive rate	0.5
S	Maximum number of strong classifiers	10
T	Maximum number of weak classifiers in each stage	100

5. AdaBoost Result and Observations

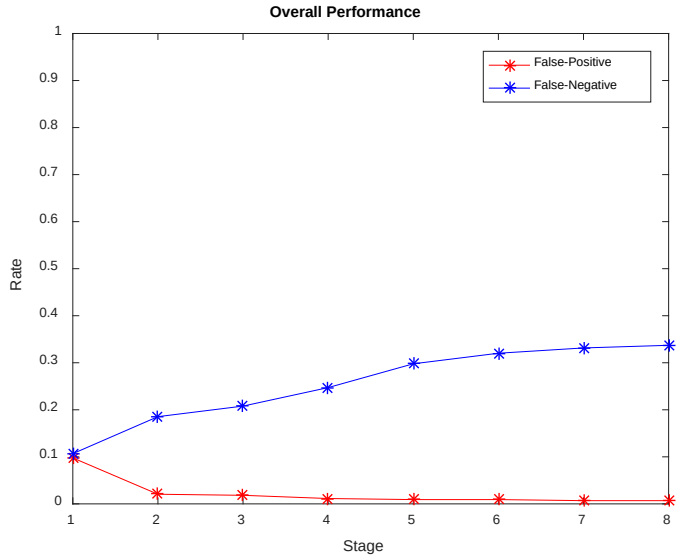
Training result:

The number of positive images, number of negative images, true detection rate , and the false positive rate in each stage is tabulated as follows.

stage	# of weak classifier	# of positive images before / after	Stage detection rate	# of negative images before / after	Stage false positive rate
1	8	710 -> 710	100.00%	1758 -> 755	42.95%
2	13	710 -> 710	100.00%	755 -> 376	49.80%
3	9	710 -> 710	100.00%	376 -> 181	48.14%
4	8	710 -> 710	100.00%	181 -> 80	44.20%
5	8	710 -> 710	100.00%	80 -> 37	46.25%
6	8	710 -> 710	100.00%	37 -> 12	32.43%
7	5	710 -> 710	100.00%	12 -> 2	16.67%
8	2	710 -> 710	100.00%	2 -> 0	0.00%

Test result:

The overall false-positive rate and the overall false-negative rate at each stage is shown in the following figure.



The overall false-positive rate and the overall false-negative rate at each stage is tabulated as follows.

stage	# of weak classifier	Overall false-positive rate	Overall false-negative rate
1	8	0.098	0.107
2	13	0.020	0.185
3	9	0.018	0.208
4	8	0.011	0.247
5	8	0.009	0.298
6	8	0.009	0.320
7	5	0.007	0.331
8	2	0.007	0.337

According to the above results, we have the following observations:

a. Based on the configuration we have for the cascaded classifier, we expect to see the overall false-positive rate decreases as we use more strong classifiers. And we also expect to see the overall false-negative rate increases as we use more strong classifiers. The result we have is reasonable.

b. The final false-positive rate is 0.007, while the final false-negative rate is 0.337. The false-negative rate might be reduced if we use more features during the training stage. However, this will increase the computation effort.

Code hw11_RihTengWu_PCA_LDA.m

```
% ECE 661 HW 11 - Face Recognition, PCA and LDA
% Student: Rih-Teng Wu

clc
clear all
close all

% ==== parameters
p = 1:1:20; % subspace dimensionality, number of eigen-vectors used to
project image
num_class = 30; % number of different classes
num_data = 21;  % number of data in each class

% ==== Do PCA or LDA
Do_PCA = 1; % 0: dont do; 1: do
Do_LDA = 1;

if Do_PCA
    % ==== PCA training
    train_path = 'ECE661_2016_hw11_DB1\train\';
    cd 'ECE661_2016_hw11_DB1\train'
    train_file = dir('*.png');
    cd '..'
    cd '..'

    [y_train, W_K, mean_img] = trainPCA(train_path,train_file,p);

    % ==== PCA testing
    test_path = 'ECE661_2016_hw11_DB1\test\';
    cd 'ECE661_2016_hw11_DB1\test'
    test_file = dir('*.png');
    cd '..'
    cd '..'

    % ==== generate target label
    target = [];
    for i = 1:num_class
        temp = [i*ones(num_data,1)];
        target = [target; temp];
    end

    [PCA_accuracy] =
testPCA(test_path,test_file,y_train,W_K,mean_img,target);
    figure, plot(p,PCA_accuracy);

end


if Do_LDA
    % ==== LDA training
    train_path = 'ECE661_2016_hw11_DB1\train\';
    cd 'ECE661_2016_hw11_DB1\train'
```

```
    train_file = dir('*.png');
    cd '..'
    cd '..'

    [y_train, W_K, mean_img] =
trainLDA(train_path,train_file,p,num_class,num_data);

    % ==== LDA testing
    test_path = 'ECE661_2016_hw11_DB1\test\';
    cd 'ECE661_2016_hw11_DB1\test'
    test_file = dir('*.png');
    cd '..'
    cd '..'

    % ==== generate target label
    target = [];
    for i = 1:num_class
        temp = [i*ones(num_data,1)];
        target = [target; temp];
    end

    [LDA_accuracy] =
testPCA(test_path,test_file,y_train,W_K,mean_img,target);
    figure, plot(p,LDA_accuracy);
end

figure, plot(p,PCA_accuracy,'b*-',p,LDA_accuracy,'r*-');
legend('PCA','LDA'); xlabel('Subspace Dimension'); ylabel('Accuracy');
```

Code trainPCA.m

```
function [ y_train, W_K, mean_img ] = trainPCA( train_path,train_file,p )
% This function output the eigen-vector matrix and the mean of training
% images for PCA method
% Author: Rih-Teng Wu
% y_train: the projected training feature vector for all training images
% W_K: matrix containing the largest p eigen-vectors of covariance
% matrix C
% mean_img: mean of all training images
% train_path: path of training folder
% train_file: structure contains training image names
% p: the subspace dimensionality

img = imread([train_path train_file(1).name]);
[h,w,~] = size(img); % assume all training images have the same sizes
n = h*w;             % number of pixels in each image

% ==== normalize each image, calculate mean of training images, form X
(normalized images)
X = [];
for i = 1:length(train_file)
    file_path = [train_path train_file(i).name];
    img = imread(file_path);

    % ==== convert to gray-scale and form vector
    img_g = rgb2gray(img); % need to first convert to gray, then convert to
double
    img_g = double(img_g); % convert to double
    x_i = reshape(img_g',[n,1]);

    % ==== normalize image
    x_i_n = x_i./norm(x_i);

    % ==== form X
    X = [X x_i_n];
end

% ==== calculate mean of training images after normalization
mean_img = mean(X,2);

% ==== subtract from mean
X2 = X - repmat(mean_img,[1,size(X,2)]);

% ==== C = X*X', but first calculate the eigenvectors of X'*X
[u,D1,~] = eig(X2'*X2); % The eigen-values is in ascending order
[~,idx] = sort(diag(D1),'descend');
u2 = u(:,idx);        % get the eigenvectors in descending order

% ==== get the eigenvectors of C
W = X2*u2;

% ==== normalize eigenvectors W
W_n = W;
```

```
for i = 1:size(W,2)
    W_n(:,i) = W(:,i)./norm(W(:,i));
end

% ==== extract the p largest eigenvectors
for i = 1:length(p)
    W_K{i} = W_n(:,1:p(i));
end

% ==== compute the projected y_train
for i = 1:length(p)
    y_train{i} = W_K{i}'*X2;
end

end
```

Code testPCA.m

```
function [ PCA_accuracy ] =
testPCA( test_path,test_file,y_train,W_K,mean_img,target )
% This function output the test accuracy using the nearest neighbor method
% Author: Rih-Teng Wu
% PCA_accuracy: the accuracy of test images
% test_path: the path of the test images
% test_file: contains the names of test images
% y_train: The training feature vector, of dimension p*N (N:number or
training images)
% W_K: matrix containing the largest p eigen-vectors of covariance
% mean_img: mean of all training images
% target: the target label of test images

img = imread([test_path test_file(1).name]);
[h,w,~] = size(img); % assume all test images have the same sizes
n = h*w;             % number of pixels in each image

% ==== normalize each image
X = [];
for i = 1:length(test_file)
    file_path = [test_path test_file(i).name];
    img = imread(file_path);

    % ==== convert to gray-scale and form vector
    img_g = rgb2gray(img); % need to first convert to gray, then convert to
double
    img_g = double(img_g); % convert to double
    x_i = reshape(img_g',[n,1]);

    % ==== normalize image
    x_i_n = x_i./norm(x_i);

    % ==== form X
    X = [X x_i_n];
end

% ==== subtract X from the mean of training images
X2 = X - repmat(mean_img,[1,size(X,2)]);

% ==== calculate the projected y_test
for i = 1:length(W_K)
    y_test{i} = W_K{i}'*X2;
end

% ==== perform nearnest neighbor and calculate accuracy
N = size(y_train{1},2);
PCA_accuracy = [];

for i = 1:length(W_K)
    y_train_temp = y_train{i};
    y_test_temp = y_test{i};
```

```
    predict_label = [];
    for j = 1:length(test_file)
        diff = y_train_temp - repmat(y_test_temp(:,j),[1 N]);
        temp = diff.^2;
        distacne = sqrt(sum(temp,1));

        [~,idx] = sort(distacne,'ascend');
        temp_label = target(idx(1));
        predict_label = [predict_label; temp_label];
    end

    % ==== calculate accuracy
    [row,~,~] = find(predict_label==target);
    accuracy = length(row)/length(test_file);
    PCA_accuracy = [PCA_accuracy; accuracy];
end

end
```

Code trainLDA.m

```
function [ y_train, W_K, mean_img ] =
trainLDA( train_path,train_file,p,num_class,num_data)
% This function output the eigen-vector matrix and the mean of training
% images for LDA method
% Author: Rih-Teng Wu
% y_train: the projected training feature vector for all training images
% W_K: the matrix containing the smallest p eigen-vectors of SW
% mean_img: mean of all training images
% train_path: path of training folder
% train_file: structure contains training image names
% p: the subspace dimensionality
% num_class: number of classes in training images
% num_data: number of training images in each class

img = imread([train_path train_file(1).name]);
[h,w,~] = size(img); % assume all training images have the same sizes
n = h*w;             % number of pixels in each image

% ==== normalize each image, calculate mean of training images, form X
(normalized images)
X = [];
for i = 1:length(train_file)
    file_path = [train_path train_file(i).name];
    img = imread(file_path);

    % ==== convert to gray-scale and form vector
    img_g = rgb2gray(img); % need to first convert to gray, then convert to
double
    img_g = double(img_g); % convert to double
    x_i = reshape(img_g',[n,1]);

    % ==== normalize image
    x_i_n = x_i./norm(x_i);

    % ==== form X
    X = [X x_i_n];
end

% ==== calculate the global mean of training images after normalization
mean_img = mean(X,2);

% ==== calculate the mean of each class
mean_class = [];
for i = 1:num_class
    mean_class_temp = mean(X(:,num_data*(i-1)+1:num_data*i),2);
    mean_class = [mean_class mean_class_temp];
end

% ==== calculate SB, constant is not important (will do normalization)
X_mean = mean_class - repmat(mean_img,[1 num_class]);
S_B = X_mean*X_mean';

% ==== Use computation trick (X'*X) to do eigen-decomposition on SB,
```

```
% calculate DB, Y, and Z
[u,D1,~] = eig(X_mean'*X_mean);
[~,idx] = sort(diag(D1),'descend');
u2 = u(:,idx);        % get the eigenvectors in descending order
D2_temp = D1(:,idx);
D2 = flipud(D2_temp); % get the eigenvalues in descending order
V = X_mean*u2;

Y = V(:,1:num_class); % get Y
DB = D2;              % get DB (The eigenvalues of X'*X and X*X' are the
same, only X*X' has 0 as an additional eigenvalue.
Z = Y*DB^(-1/2);      % get Z

% ==== compute U, by eigen-decomposition of Z'*SW*Z = (Z'X_W)(Z'X_W)'
% ==== first form X_W
mean_class_extend = [];
for i = 1:num_class
    temp = repmat(mean_class(:,i),[1 num_data]);
    mean_class_extend = [mean_class_extend temp];
end

X_W = X - mean_class_extend;

[U,D3,~] = eig((Z'*X_W)*(Z'*X_W)');
[~,idx] = sort(diag(D3),'ascend');
U = U(:,idx);        % get the eigenvectors in ascending order

% ==== extract the p smallest eigenvectors
for i = 1:length(p)
    W_T = U(:,1:p(i))'*Z';
    W{i} = W_T';
end

% ==== normalize eigenvectors W
for i = 1:length(W)
    W_temp = W{i};
    W_n = W_temp;
    for j = 1:size(W_temp,2)
        W_n(:,j) = W_n(:,j)./norm(W_n(:,j));
    end
    W_K{i} = W_n;
end

% ==== compute the projected y_train
X2 = X - repmat(mean_img,[1,size(X,2)]);

for i = 1:length(p)
    y_train{i} = W_K{i}'*X2;
end

end
```

Code hw11_RihTengWu_Adaboost.m

```
% ECE 661 HW 11 - Car Recognition, AdaBoost
% Student: Rih-Teng Wu

clc
clear all
close all

% ==== whether or not to do operations
Do_feature_generate = 0; % 1:do; 0: not to do
Do_training = 0;
Do_testing = 1;

% ==== user specified parameters
threshold_positive = 1; % acceptable positive detection rate
threshold_FP = 0.5;     % acceptable False-Positive rate
S = 10;                 % maximum number of strong classifiers
T = 100;                 % maximum number of weak classifiers in each stage

% ==== feature generation
if Do_feature_generate
    % ==== train feature
    train_path_p = 'ECE661_2016_hw11_DB2\train\positive\';
    cd 'ECE661_2016_hw11_DB2\train\positive'
    train_file_p = dir('*.png');
    cd '..'; cd '..'; cd '..';

    train_path_n = 'ECE661_2016_hw11_DB2\train\negative\';
    cd 'ECE661_2016_hw11_DB2\train\negative'
    train_file_n = dir('*.png');
    cd '..'; cd '..'; cd '..';

    [feature_train_p] = generateFeatureAdaBoost(train_path_p,train_file_p);
    [feature_train_n] = generateFeatureAdaBoost(train_path_n,train_file_n);

    % ==== test feature
    test_path_p = 'ECE661_2016_hw11_DB2\test\positive\';
    cd 'ECE661_2016_hw11_DB2\test\positive'
    test_file_p = dir('*.png');
    cd '..'; cd '..'; cd '..';

    test_path_n = 'ECE661_2016_hw11_DB2\test\negative\';
    cd 'ECE661_2016_hw11_DB2\test\negative'
    test_file_n = dir('*.png');
    cd '..'; cd '..'; cd '..';

    [feature_test_p] = generateFeatureAdaBoost(test_path_p,test_file_p);
    [feature_test_n] = generateFeatureAdaBoost(test_path_n,test_file_n);

    % ==== save data
    save('Train_positive.mat','feature_train_p','-v7.3');
    save('Train_negative.mat','feature_train_n','-v7.3');
    save('Test_positive.mat','feature_test_p','-v7.3');
```

```
    save('Test_negative.mat','feature_test_n','-v7.3');
end

% ==== AdaBoost training
if Do_training
    load('Train_positive.mat');
    load('Train_negative.mat');

    % ==== all features, [positive negative]
    feature_all = [feature_train_p feature_train_n];
    N_pos = size(feature_train_p,2);
    N_neg = size(feature_train_n,2);
    index = 1:1:N_pos+N_neg;
    new_idx = index; % new data index after classified by strong classifier

    for i = 1:S

        Strong =
performCascade(feature_all,N_pos,new_idx,i,threshold_positive,threshold_FP,T)
;

        % ==== update data index
        new_idx = Strong.UpdatedIndex;

        % ==== Stopping criteria (all negative examples are detected)
        neg_idx = find(new_idx > N_pos);

        % ==== record parameters
        Train_result(i) = Strong;

        if length(neg_idx) == 0
            break;
        end
        % ==== update N_pos
        N_pos = length(new_idx) - length(neg_idx);

    end
    save('Train_result.mat','Train_result','-v7.3');
end

% ==== AdaBoost testing
if Do_testing
    load('Test_positive.mat');
    load('Test_negative.mat');
    load('Train_result.mat');

    num_stage = length(Train_result); % number of strong classifier
    N_pos_all = size(feature_test_p,2);
    N_neg_all = size(feature_test_n,2);
    num_mis_pos = 0; % number of misclassfied positives
    num_cor_neg = 0; % number of correctly negatives

    for i = 1:num_stage
        stage = Train_result(i); % extract the ith strong classifier
```

```
        num_weak = stage.numberOfweak; % number of weak classifier
        weak_params = stage.parameters;% parameters of weak classifier

        f_idx = weak_params(1,1:num_weak); % best features
        theta = weak_params(2,1:num_weak); % the threshold for feature
        p = weak_params(3,1:num_weak);     % the polarity for inequality
        alpha = weak_params(4,1:num_weak); % alpha values for weak
classifiers

        predict_result =
AdaBoostTesting(feature_test_p,feature_test_n,f_idx,theta,p,alpha,num_weak);

        predict_label = predict_result.predict;
        N_pos_test = predict_result.N_pos;
        N_neg_test = predict_result.N_neg;

        % ==== calculate overall FP and FN at each stage
        mis_pos_temp = length(find(predict_label(1:N_pos_test)<1));
        num_mis_pos = num_mis_pos + mis_pos_temp;

        correct_neg_temp = length(find(predict_label(N_pos_test+1:end)< 1));
        num_cor_neg = num_cor_neg + correct_neg_temp;

        FP(i) = (N_neg_all-num_cor_neg)/N_neg_all;
        FN(i) = num_mis_pos/N_pos_all;

        % ==== update data
        remain_pos_idx = find(predict_label(1:N_pos_test)>0);
        feature_test_p = feature_test_p(:,remain_pos_idx);

        remain_neg_idx = find(predict_label(N_pos_test+1:end)>0);
        feature_test_n = feature_test_n(:,remain_neg_idx);
    end

    % ==== plot accumulative FP and FN
    figure, plot(1:1:num_stage,FP,'r*-',1:1:num_stage,FN,'b*-');
    legend('False-Positive','False-Negative');
    title('Overall Performance'); xlabel('Stage'); ylabel('Rate');
    ylim([0 1]);
end
```

Code generateFeatureAdaBoost.m

```
function [ feature ] = generateFeatureAdaBoost(path,file)
% This function output the Haar-like feature used for AdaBoost learning based
on
% Integral images
% Author: Rih-Teng Wu
% feature: the desirable output feature
% path: the path of files
% file: contains the names of the images

img = imread([path file(1).name]);
[h,w,~] = size(img);    % assume all training images have the same sizes
num_img = length(file); % number of images

% ==== filter sizes
h_size = 2:2:w; % sizes of horizontal edge filter, 1x2,1x4,...1x40
v_size = 2:2:h; % sizes of horizontal edge filter, 2x2,4x2,...20x2

% ==== initial feature
feature = zeros(11900,num_img);

% ==== generate Haar-like feature
for i = 1:num_img
    file_path = [path file(i).name];
    img = imread(file_path);

    % ==== convert to gray-scale and double
    img_g = rgb2gray(img); % need to first convert to gray, then convert to
double
    img_g = double(img_g); % convert to double

    % ==== obtain integral image
    img_int = integralImage(img_g); % there is zero padding at the top and
left of the image
                                    % zero padding will be helpful

    % ==== compute horizontal Haar-like feature
    feature_temp = [];

    for j = 1:length(h_size)
        width = h_size(j);

        for k = 1:h
            for m = 1:w-width+1
                corner_0 = [k m;k m+width/2;k+1 m;k+1 m+width/2]; % 1,2,3,4
corners
                corner_1 = [k m+width/2;k m+width;k+1 m+width/2;k+1 m+width];

                rec_0 = getRec(img_int,corner_0);
                rec_1 = getRec(img_int,corner_1);
                diff = rec_1 - rec_0;

                feature_temp = [feature_temp; diff];
```

```
            end
        end
    end

    % ==== compute vertical Haar-like feature
    for j = 1:length(v_size)
        height = v_size(j);

        for k = 1:h-height+1
            for m = 1:w-2+1
                corner_1 = [k m;k m+2;k+height/2 m;k+height/2 m+2]; % 1,2,3,4
corners
                corner_0 = [k+height/2 m;k+height/2 m+2;k+height m;k+height
m+2];

                rec_1 = getRec(img_int,corner_1);
                rec_0 = getRec(img_int,corner_0);
                diff = rec_1 - rec_0;

                feature_temp = [feature_temp; diff];
            end
        end
    end

    feature(:,i) = feature_temp;

end

end
```

Code getRec.m

```
function [ Rec ] = getRec( img_int,corner )
% This function output the sum of pixels within the rectangle
% with four specified corner points based on integral image
% Author: Rih-Teng Wu
% Rec: sum of pixels withing the rectangle
% img_int: integral image
% corner: four specified corner points

One = img_int(corner(1,1),corner(1,2)); % corner 1
Two = img_int(corner(2,1),corner(2,2)); % corner 2
Three = img_int(corner(3,1),corner(3,2)); % corner 3
Four = img_int(corner(4,1),corner(4,2)); % corner 4

Rec = Four + One - (Two+Three);

end
```

Code performCascade.m

```
function [ Strong ] =
performCascade( feature_all,N_pos,new_idx,stage,thres_pos,thres_FP,T)
% This function output the strong classfier based on AdaBoost method
% Author: Rih-Teng Wu
% Strong: output structure that contains information of the aggregated
% weak classifiers
% feature_all: training features for all training images
% N_pos: number of remaining positive samples
% new_idx: the data index after passing the previous strong classifier
% stage: indicate the current stage of building strong classifer
% thres_pos: acceptable positive detection rate
% thres_FP:  acceptable False-Positive rate
% T: maximum number of weak classifiers in each stage

% ==== calculate remaining samples
N_img = length(new_idx); % number of total remaining images
N_neg = N_img - N_pos;

% ==== update feature
feature = feature_all(:,new_idx);

% ==== initial weights and labels
weights = zeros(N_img,1);
labels = zeros(N_img,1); % 1: positive; 0: negative

for i = 1:N_img
    if i <= N_pos
        weights(i) = 1/2/N_pos;
        labels(i) = 1;
    else
        weights(i) = 1/2/N_neg;
    end
end

% ==== Initial parameters
Strong_result = zeros(N_img,1); % classification result of the strong
classifier
alpha = zeros(T,1);             % alpha value for each weak classifier
h = zeros(4,T);                 % selected feature, threshold, polarity,
alpha for each weak classifier (used for testing later on)
h_result = zeros(N_img,T);      % classification result of each weak
classifier
accuracy_pos = [];              % accuracy of positive examples for each
strong classifier
FP_neg = [];                    % False positive of negative examples for
each strong classifier


for t = 1:T
    t
    % ==== normalize weights
    weights = weights./sum(weights);
```

```
    % ==== get best weak classifier
    best_weak = getBestWeak(feature,weights,labels,N_pos);

    % ==== get the parameters of best weak classifier
    error = best_weak.minError;
    h(1,t) = best_weak.feature;
    h(2,t) = best_weak.theta;
    h(3,t) = best_weak.p;
    h_result(:,t) = best_weak.classification;

    % ==== compute beta
    beta = error/(1-error);
    alpha(t,1) = log(1/beta);
    h(4,t) = alpha(t,1);

    % ==== update weights
    weights = weights.*beta.^(1-xor(labels,h_result(:,t)));

    % ==== compute strong classifier result
    strong_temp = h_result(:,1:t)*alpha(1:t,1);
    % ==== instead of set the threshold to 0.5*sum(alpha(1:t,1));
    % ==== we adjust the threshold to make all positive examples are
    % ==== correctly classified
    threshold = min(strong_temp(1:N_pos));

    for i = 1:N_img
        if strong_temp(i) >= threshold
            Strong_result(i) = 1;
        else
            Strong_result(i) = 0;
        end
    end

    % ==== calculate accuracy of positive examples and False positive
    acc_temp = sum(Strong_result(1:N_pos))/N_pos;
    accuracy_pos = [accuracy_pos; acc_temp];

    FP_temp = sum(Strong_result(N_pos+1:end))/N_neg;
    FP_neg = [FP_neg; FP_temp];

    accuracy_pos(t)
    FP_neg(t)

    % ==== stopping criteria
    if (accuracy_pos(t) >= thres_pos) && (FP_neg(t) <= 0.5)
        break;
    end
end

% ==== get remaining positive and negative examples
% ==== if use thres_pos other than 1, this part need to be modified

[sort_neg,sort_neg_idx] = sort(Strong_result(N_pos+1:end),'ascend');
```

```
% ==== output
if sum(sort_neg)>0

    for i = 1:N_neg
        if sort_neg(i)>0
            remain_neg_idx = sort_neg_idx(i:end);
            Strong.UpdatedIndex = [1:1:N_pos,remain_neg_idx'+N_pos];
            break;
        end
    end
else
    Strong.UpdatedIndex = [1:1:N_pos];
end

Strong.numberOfweak = t; % number of weak classifiers in each stage
Strong.parameters = h; % parameters used to test
end
```

Code getBestWeak.m

```
function [ best_weak ] = getBestWeak( feature,weights,labels,N_pos )
% This function output the best weak classifer based on sorting method
% Author: Rih-Teng Wu
% best_weak: output structure that contains information of the best
% weak classifier
% feature: training features for the remaining training images
% weights: the updated weights from the previous weak classifier
% labels: the target labels
% N_pos: number of remaining positive samples

N_feat = size(feature,1); % number of features
N_img = size(feature,2);  % number of remaining images

% ==== calculte T_plus and T_minus
T_plus = repmat(sum(weights(1:N_pos)),[N_img 1]);
T_minus = repmat(sum(weights(N_pos+1:end)),[N_img 1]);

% ==== iterate over all features
best_weak.minError = inf; % initial minimum error

for i = 1:N_feat
    feat_temp = feature(i,:);

    % ==== sorting in ascending order
    [sort_feat,sort_idx] = sort(feat_temp,'ascend');
    sort_weight = weights(sort_idx);
    sort_label = labels(sort_idx);

    % ==== calculate S_plus and S_minus
    S_plus = cumsum(sort_weight.*sort_label); % sum of positive weights for
all threshold
    S_minus = cumsum(sort_weight) - S_plus;   % sum of negative weights for
all threshold

    % ==== determin threshold
    err_1 = S_plus + (T_minus - S_minus);
    err_2 = S_minus + (T_plus - S_plus);
    min_error_temp = min(err_1,err_2);
    [min_error, min_idx] = min(min_error_temp);

    % ==== get cloassification result
    classify_result = zeros(N_img,1);

    if err_1(min_idx) <= err_2(min_idx)
        p = -1;
        classify_result(min_idx+1:end) = 1; % label 1 to the examples above
current example
        classify_result(sort_idx) = classify_result;
    else
        p = 1;
        classify_result(1:min_idx) = 1; % label 1 to the examples below
current example
        classify_result(sort_idx) = classify_result;
```

```
    end

    % ==== get best weak classifier
    if min_error < best_weak.minError
         best_weak.minError = min_error;
         best_weak.p = p;
         best_weak.feature = i;
         best_weak.classification = classify_result;
         % ==== get threshold value for best feature
         if min_idx == 1
             best_weak.theta = sort_feat(1) - 0.01; % 0.01 is just a constant
to make sure every feature value is larger than theta
         elseif min_idx == N_feat
             best_weak.theta = sort_feat(N_feat) + 0.01; % 0.01 is just a
constant to make sure every feature value is smaller than theta
         else
             best_weak.theta = mean([sort_feat(min_idx),sort_feat(min_idx-
1)]);
         end
    end
end

end
```

Code AdaBoostTesting.m

```
function [ result ] =
AdaBoostTesting( feature_p,feature_n,f_idx,theta,p,alpha,num_weak)
% This function output the classification result in each stage based on
AdaBoost
% testing
% Author: Rih-Teng Wu
% result: the output of predicted labels, # of positives, # of negatives
% feature_p: features of positive test samples
% feature_n: features of negative test samples
% f_idx: best features
% theta: the threshold for feature
% p: the polarity for inequality
% alpha: alpha values for weak classifiers
% num_weak: number of weak classifier in this stage

feature_all = [feature_p feature_n];
N_pos = size(feature_p,2);
N_neg = size(feature_n,2);
N_img = N_pos + N_neg;

weak_result = zeros(N_img,num_weak);
% ==== get weak classifier results
for i = 1:num_weak
    feat_temp = feature_all(f_idx(i),:);

    for j = 1:N_img
        if p(i)*feat_temp(j) <= p(i)*theta(i)
            weak_result(j,i) = 1;
        end
    end
end

% ==== get strong classifier result
strong_temp = weak_result*alpha';
threshold = 0.5*sum(alpha);
Strong_result = zeros(N_img,1);

for i = 1:N_img
    if strong_temp(i) >= threshold
        Strong_result(i) = 1;
    end
end

% ==== output
result.predict = Strong_result;
result.N_pos = N_pos;
result.N_neg = N_neg;

end
```