ECE 661 (Fall 2016) - Computer Vision - HW 11

Debasmit Das

December 11, 2016

1 Face Recogntion

This task here is to perform PCA and LDA on face images and then do nearest neighbor classification.

1.1 Principal Component Analysis (PCA)

To perform Face recognition using PCA, both train and test images need to be converted into a vectorized format. With the vectorized image, the covariance matrix $\mathbf{C}$ is computed.

1.1.1 Estimate Covariance Matrix

If we have N vectorized images, $\mathbf{x}_i$ being the i^{th} image. where $i = 1, 2...N$. To compute the co-variance matrix $\mathbf{C}$, the mean vector of N images will be given as -

$$\overline{\mathbf{m}} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}_i \tag{1}$$

and $\mathbf{X}$ is such that

$$\mathbf{X} = [\mathbf{x}_1 - \overline{\mathbf{m}} \quad \mathbf{x}_2 - \overline{\mathbf{m}}...\mathbf{x}_N - \overline{\mathbf{m}}] \tag{2}$$

$\mathbf{X}$ need to be normalized to take care of illumination, such that $\|\mathbf{x}_i\| = 1$ for all i. The covariance $\mathbf{C} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{X}\mathbf{X}^T$.
The eigen-vectors $\mathbf{w}_l$ of $\mathbf{C}$ corresponding to the K largest eigenvalues will constitute the PCA feature set, denoted as $\mathbf{W}_k$. Different K's should be tested. A computation trick is applied to compute the above eigen-vectors.

1.1.2 Computational Trick Used

If $\mathbf{w}$ is an eignevector of $\mathbf{C}$, it must satisfy,

$$\mathbf{X}\mathbf{X}^T\mathbf{w} = \lambda\mathbf{w} \tag{3}$$

Instead of computing eigen-vectors of $\mathbf{C}$, the eigen-vectors of $\mathbf{X}^T\mathbf{X}$ are to be computed first. Let that eigen-vector be denoted as $\mathbf{u}$. Then, we know that

$$\mathbf{X}^T\mathbf{X}\mathbf{u} = \lambda\mathbf{u}. \tag{4}$$

To get $\mathbf{w}$ from $\mathbf{u}$, we pre-multiply the previous equation by $\mathbf{X}$ to get

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

So, we need to just use $\mathbf{w} = \mathbf{X}\mathbf{u}$.

1.1.3 Projecting Data Space

Both the training and test data need to be projected into the eigen space, using $\mathbf{y_i} = \mathbf{W}^T\mathbf{x}_i$, where $\mathbf{W} = [\mathbf{w}_1 \quad \mathbf{w}_2 ... \mathbf{w}_K]$. Then the trained feature is used in nearest neighbor classification.

1.2 Linear Discriminant Analysis (LDA)

The goal of LDA is to find directions in the underlying space that maximally discrimnates between classes. For that we have to look into the between class scatter $\mathbf{S}_B$ and the within class scatter $\mathbf{S}_W$.

1.2.1 Defining $\mathbf{S}_B$ and $\mathbf{S}_W$

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

$$\mathbf{S}_B = \frac{1}{|\mathbb{C}|}\sum_{i=1}^{|\mathbb{C}|}(\overline{\mathbf{m}}_i - \overline{\mathbf{m}})(\overline{\mathbf{m}}_i - \overline{\mathbf{m}})^T \tag{6}$$

where $\mathbb{C}$ is the set of all classes and $\overline{\mathbf{m}}$ is the global mean. The within class scatter is defined as

$$\mathbf{S}_W = \frac{1}{|\mathbb{C}|}\sum_{i=1}^{|\mathbb{C}|}\frac{1}{|\mathbb{C}_i|}\sum_{k=1}^{|\mathbb{C}_i|}(\mathbf{x}_k - \overline{\mathbf{m}}_i)(\mathbf{x}_k - \overline{\mathbf{m}}_i)^T \tag{7}$$

where subset to images belonging to class i is referred to as $\mathbb{C}_i$ and $\overline{\mathbf{m}}_i$ is the mean image vector for each class.

1.2.2 LDA Objective

The objective of LDA is to find LDA eigenvectors $\mathbf{w}$ that maximize Fisher Discriminant Function.

$$J(\mathbf{w}) = \frac{\mathbf{w}^T\mathbf{S}_B\mathbf{w}}{\mathbf{w}^T\mathbf{S}_W\mathbf{w}} \tag{8}$$

To solve for the $\mathbf{w}$, Yu and yang's algorithm is used.

1.2.3 Yu and Yang's algorithm

Initially we have to carry out eigen decomposition of $\mathbf{S}_B$. The eigen values are diagonalized and need to be sorted in descending order. This will yield a matrix $\mathbf{V}$ consisting of the corresponding eignevectors. The first K eigen vectors constitute matrix $\mathbf{Y}$. Then, a matrix $\mathbf{Z} = \mathbf{Y}\mathbf{D}_B^{-0.5}$, where $\mathbf{D}_B$ is the upper-left $K \times K$ sub-matrix of the diagonalized eigen-values of $\mathbf{S}_B$. Or $\mathbf{D}_B = \mathbf{Y}^T\mathbf{S}_B\mathbf{Y}$. In fact , we use the following equation

$$\mathbf{D}_B = (\mathbf{Y}^T\mathbf{M})(\mathbf{Y}^T\mathbf{M})^T \tag{9}$$

where $\mathbf{M} = [\overline{\mathbf{m}}_1 - \overline{\mathbf{m}} \quad \overline{\mathbf{m}}_1 - \overline{\mathbf{m}} ... \overline{\mathbf{m}}_{|\mathbb{C}|} - \overline{\mathbf{m}}]$. The eigenvector matrix $\mathbf{U}$ is computed using eigen decomposition of $\mathbf{Z}^T\mathbf{S}_W\mathbf{Z}$. This is done using the following equation

$$\mathbf{Z}^T\mathbf{S}_W\mathbf{Z} = (\mathbf{Z}^T\mathbf{X}_W)(\mathbf{Z}^T\mathbf{X}_W)^T \tag{10}$$

where $\mathbf{X}_W = [\mathbf{x}_{11} - \overline{\mathbf{m}}_1 \quad ...\mathbf{x}_{1k} - \overline{\mathbf{m}}_k \quad ...\mathbf{x}_{|\mathbb{C}|1} - \overline{\mathbf{m}}_1 \quad ...\mathbf{x}_{|\mathbb{C}|k} - \overline{\mathbf{m}}_k]$

The eigen-vectors with the largest eigen-values are discarded. the matrix of LDA eigen-vectors $\mathbf{W}$ is given as follows -

$$\mathbf{W}^T = \hat{\mathbf{U}}^T\mathbf{Z}^T \tag{11}$$

We should also normalize $\mathbf{W}$

1.2.4 Project train and test images

Both train and test images need to be projected into the eigen-space before classification can be used.

$$\mathbf{y} = \mathbf{W}^T(\mathbf{x} - \overline{\mathbf{m}}) \tag{12}$$

1.3 Results and Discussion

We use accuracy as an evaluation metric for comparing LDA and PCA as a function of the dimension of eigenspace i.e. the number of eigen-vectors. The accuracy is given as follows- accuracy = (No. of correctly recognized images)/(Total no. of images). The accuracy is plotted as follows -

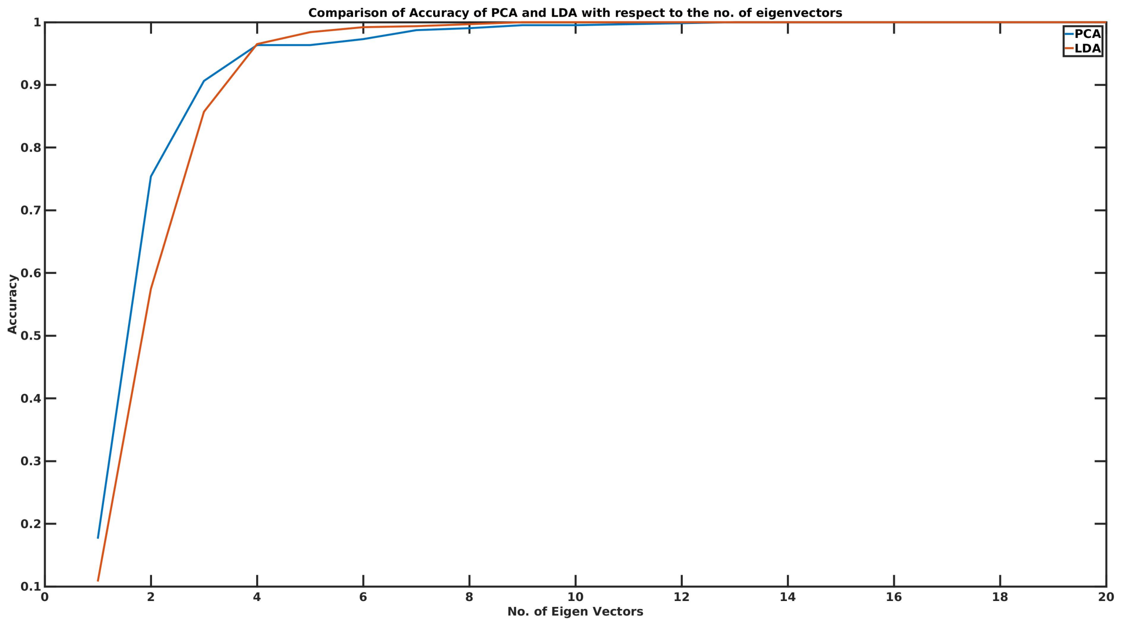

Figure 1: Comparing Performance of PCA and LDA

From the figure, we see that LDA for few eigen-vectors does not perform as well as PCA but as the no. of eigen-vectors increases LDA performs better than PCA and reaches 100 percent accuracy faster. PCA takes close to 10 dimensional eigen-space to reach 100 percent accuracy while for LDA it is a 6 dimensional space. So computationally speaking, PCA is good for low-dimensional representation while LDA is good for better discriminative performance. This is mainly because PCA is an unsupervised method and LDA is a supervised method and therefore LDA models the class separability.

2 Object Detection

Here, we use the Viola-Jones detector or the cascaded Adaboost for object detection. The object in our case is a car. In each cascade of our framework we have a desired a target false positive and true detection rate.

2.1 AdaBoost Classifier

AdaBoost stands for adaptive boosting where we aggregate weak classifiers to form a strong classifier. To do so we use the following steps.

2.1.1 Haar Feature Extraction

To build weak classifiers, the Haar features need to be extracted. Haar filters are box features and can therefore use integral images for fast operation. The integral image is calculated as follows-

$$\mathbf{II}(x, y) = \sum_{x_i \leq x, y_i \leq y} \mathbf{I}(x_i, y_i) \tag{13}$$

There will be different kind of Haar features, such as $[0, 1]$ and $[1, 0]^T$. To get all possible horizontal and vertical features we have extend each feature. The 1×2 feature is extended to 1×4, 1×6, 1×8 ..., horizontally. and similarly vertically. Together we will get around 166000 features.

2.1.2 Build Weak Classifier

If we assume that the final strong classifier is built with T weak classifiers, one weak classifier is denoted as h_t. To find the best T weak classifiers, all the features are evaluated T times. To find the t weak classifier, all the features are evaluated one by one.

For each feature it is applied to all the training data to find the best threshold that can classify the training data with optimal classification rate. Before the threshold is calculated, the current feature is first sorted ascendingly accroding to feature's value for each example. The threshold is then calculated as follows-

$$e = \min(S^+ + (T^- - S^-), S^- + (T^+ - S^+)) \tag{14}$$

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 for finding the best weak classifier. The feature with the smallest error is then selected as a weak classifier h_t.

After the t^{th} weak classifier is obtained, the weight for each training image is updated as following -

$$w_{t+1,i} = w_{t,i}\beta_t^{1-e_i} \tag{15}$$

where $e_i = 0$ when correctly classified and $e_i = 1$, otherwise. $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$. The error is calculated using the following $\epsilon_t = \min \sum_i w_{t,i}|h_t(\mathbf{x}_i) - y_i|$, where $\mathbf{x}_i$ is a training image and y_i is the corresponding label.

2.1.3 Build Strong Classifier

These T classifiers constitute a strong classifier. When performing the validation or testing process, this strong classifier is used as -

$$C(\mathbf{x}) = \begin{cases} 1, & \text{if } \sum_{t=1}^{T} \alpha_t h_t(\mathbf{x}) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0, & \text{otherwise} \end{cases} \tag{16}$$

where $\alpha_t = \log \frac{1}{\beta}$

2.2 Cascaded Adaboost Classifier

To integrate adaboost with cascaded algorithm, the process of building one strong classifier is repeated for several cascaded stages. At the beginning of each stage, the features 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 and false negative rate is calculated as the following.

$$\text{false positive rate} = \text{No. of misclassified negative images/No. of negative images} \tag{17}$$

$$\text{false negative rate} = \text{No. of misclassified positive images/No. of positive images} \tag{18}$$

2.3 Results and Discussion

We discuss both training and testing results in separate sections.

2.3.1 Training Results

The number of classifiers used in each stage is summarized as follows in the table. -

Table 1: No. of classifiers in each stage of the cascade

Stage	1	2	3	4	5	6	7	8	9	10
No. of Classifiers	29	21	20	19	18	15	13	12	10	8

Next, we also visualize the accumulated false positive rate in the training process

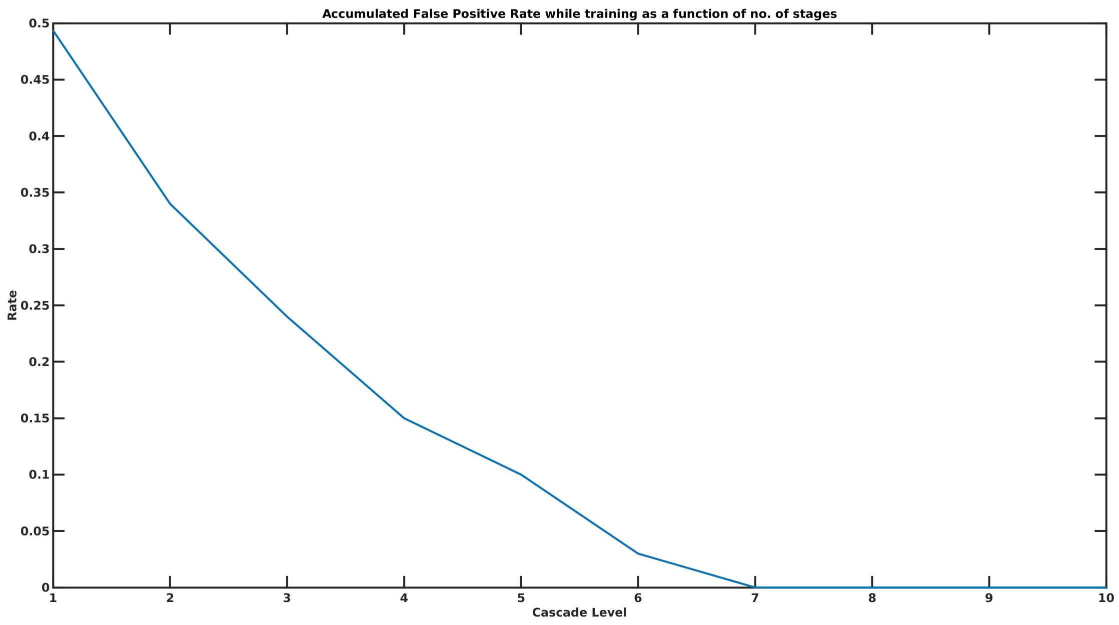

Figure 2: Accumulated False Positive Rate during training for cascade stages

2.3.2 Testing Results

We look at the testing results. We plot the accumulated false positive rate and false negative rate as the function of the number of stages on the test data.

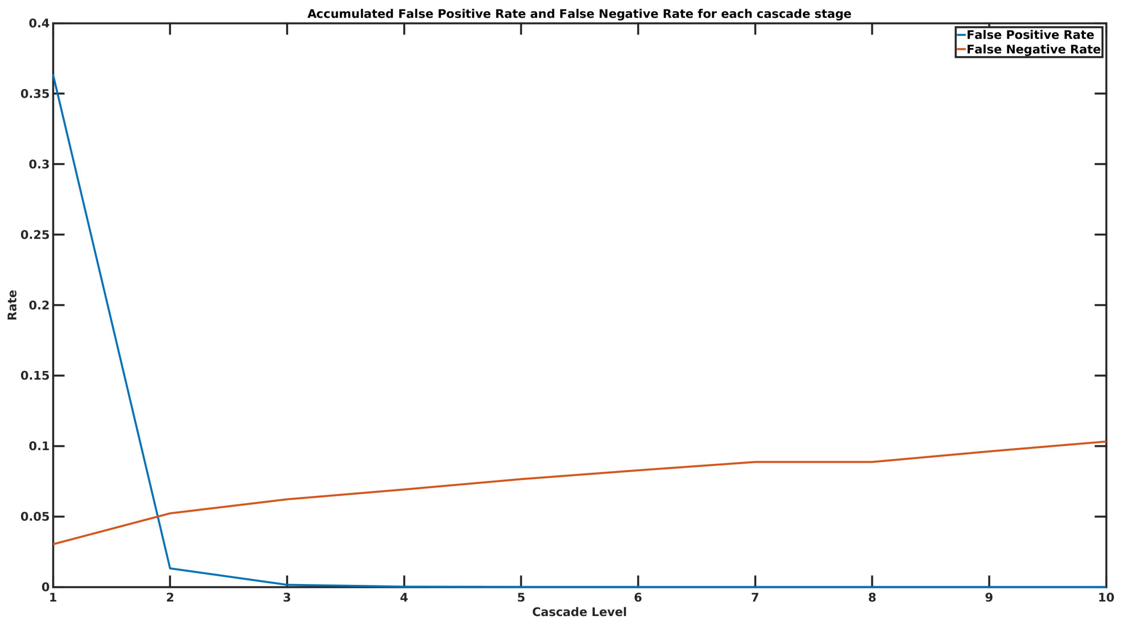

Figure 3: Accumulated False Positive Rate and False negative rate as a function of the stages for test data

So we see that the False Positive Rate decreases as expected. The False negative rate however rises. This is because of the fact that True Positive Rate decreases. Since False Negative Rate is $1 -$ True Positive Rate, False Negative Rate increases.

Code

The script is in MATLAB 2016a and is self-explanatory

Face Recognition

Main Script of PCA

```
% This is the main method for PCA

%Number of persons or the no. of classes
Npers=30;
%Number of trials per person
Ntrials=21;
%Path for training images
trPath = 'Face/train/';
tePath = 'Face/test/';
% load training images
[trainImg, ~, ~] = loadImg(trPath, Npers, Ntrials);
% load testing images
```

```
[testImg, ~, ~] = loadImg(tePath, Npers, Ntrials);
% load trained w
[w, Neig] = myPCA(trainImg); % Taking the number of eigen values and
% test using different number of eigenvectors, from small to large
accPCA = zeros(1, Neig);
TrainDataY=zeros(Npers*Ntrials,1);
for i=1:Npers*Ntrials
    TrainDataY(i,1)=ceil(i/Ntrials);
end


for i=1: Neig
partEig=w(:,1:i);

% project training images
 trainProj = zeros(i, Npers*Ntrials);
 for j = 1:Npers*Ntrials
 trainProj(:,j) = partEig' * trainImg(:,j);
 end

 % project testing images
 testProj = zeros(i, Npers*Ntrials);
 for j = 1:Npers*Ntrials
 testProj(:,j) = partEig' * testImg(:,j);
 end

TrainDataX=trainProj';
TestDataX=testProj';
%Training a K-Nearest neighbour K=1
mdl=fitcknn(TrainDataX, TrainDataY, 'NumNeighbors',1, 'distance', 'euclidean');
TestDataPred=mdl.predict(TestDataX);

%Testing using nearest neighbour and calculating accuracy
Diff=TestDataPred-TrainDataY;
accPCA(1,i)=nnz(~Diff);

end
% compute accuracy for different dim of eigen space
accPCA = accPCA / (Npers*Ntrials);
plot(accPCA);
```

PCA subroutine

```
function [normW, Neig]= myPCA(imgVec)
%Function to find PCA on image vectors

% Compute covariance matrix for sorting eigen values
[V,D]= eig(imgVec'*imgVec);
eigV = diag(D);
[~,idx] = sort(-1.0 .* eigV); % Thos is done for sorting
eigV = eigV(idx);
V = V(:,idx);
```

```
% For each image, we get the no. of eigenvectors which have
% eigen values greater than 1
Neig=0;
for i=1:size(imgVec,2)
if eigV(i) > 1
Neig = Neig + 1;
end
end

% We have to compute the weight matrix
w=imgVec*V;

%Next we normalize w
[r,c]=size(w);
normW=zeros(r,c);
for i=1:c
   normW(:,i)=w(:,i)/norm(w(:,i));
end
end
```

Main Script of LDA

```
% This is the main method for LDA
%Number of persons or the no. of classes
Npers=30;
%Number of trials per person
Ntrials=21;
%Path for training images
trPath = 'Face/train/';
tePath = 'Face/test/';
% load training images
[~,trainImg, meanTrain] = loadImg(trPath, Npers, Ntrials);
% load testing images
[~,testImg, meanTest] = loadImg(tePath, Npers, Ntrials);

% get trained data
[vecU, Z] = myLDA(trainImg,meanTrain,Npers,Ntrials);
% test using different number of eigenvalues
Neig = 30;
accLDA = zeros(1, Neig);

TrainDataY=zeros(Npers*Ntrials,1);
for i=1:Npers*Ntrials
    TrainDataY(i,1)=ceil(i/Ntrials);
end

for i = 1:Neig
 % 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
 trainProj = zeros(i, Npers*Ntrials);
 for j = 1:Npers*Ntrials
 trainProj(:,j) = W' * (trainImg(:,j)-meanTrain);
 end

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

TrainDataX=trainProj';
TestDataX=testProj';
%Training a Nearest neighbour
mdl=fitcknn(TrainDataX, TrainDataY, 'NumNeighbors',1, 'distance', 'euclidean');
TestDataPred=mdl.predict(TestDataX);

%Testing using nearest neighbour and calculating accuracy
Diff=TestDataPred-TrainDataY;
accLDA(1,i)=nnz(~Diff);

end
% compute accuracy for different dim of eigen space
accLDA = accLDA / (Npers*Ntrials);
plot(accLDA);
```

LDA subroutine

```
function [ vecU, Z ] = myLDA( imgVec, mean, Npers, 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,Npers*Ntrials);
for i = 1:Npers*Ntrials
 % This is for selecting index for each class
 classIdx = floor((i-1)/Ntrials) + 1;
 sumImg(:,classIdx) = sumImg(:,classIdx) + imgVec(:,i);
end

meani = sumImg / Ntrials;

% build mi-m after subtracting from mean
```

```
meani_m = zeros(imgSize, Npers);
for i = 1:Npers
 meani_m(:,i) = meani(:,i) - mean;
end

% compute SB i.e the between class variance
SB = meani_m * meani_m';
% ensure SB is not singular using Yu and Wang's method
[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:Npers*Ntrials
 classIdx = floor((i-1)/Ntrials) + 1;
 xk_meani(:,i) = imgVec(:,i) - meani(:,classIdx);
end

% compute the intermediate variable
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
```

Subroutine for loading Images

```
function [ normImgVec, imgVec, meanImg ] = loadImg( filePath, Npers, Ntrial )

%get image dimensions
[r,c] = size(rgb2gray(imread([filePath,'01_01.png'])));

%define output vectors
imgVec = zeros(r*c,Npers*Ntrial); %each column is an image

%load images as feature vectors
for i = 1:Npers
 for j = 1:Ntrial
 img = imread([filePath,num2str2digit(i),'_',num2str2digit(j),'.png']);
 [r,c] = size(rgb2gray(img));
 oneVec = reshape(rgb2gray(img)',r*c,1);
 imgVec(:,(i-1)*Ntrial+j) = oneVec;
 end
```

```
end

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

%normalize images
% This standardization is required for eigen decomposition
normImgVec = zeros(r*c, Npers*Ntrial);
for i = 1:Npers*Ntrial
 normImgVec(:,i) = (imgVec(:,i) - meanImg) / norm(imgVec(:,i) - meanImg);
end
end
```

Subroutine for Converting Image numbers to strings

```
function str = num2str2digit(num)
% This function is used for converting
% indices in images to strings
if num<10
str = ['0',num2str(num)];
else
str = num2str(num);
end
end
```

Car Detection

Main Script for Training

```
clc
clear all

% Get features from stored data file
featFile = load('features_adaboost_train.mat');
feat=featFile.features_adaboost.features;
Npos=featFile.features_adaboost.Npos;
Nneg=featFile.features_adaboost.Nneg;

S=10;
idx=1:Npos+Nneg;

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

    if length(idx)==Npos
        break;
    end
end
```

Script for Cascading

```
function [idx] = myCascade(featuresAll, Npos, idxPrev, stage)

%update negative no.
Nneg = length(idxPrev) - Npos;
Ntotal = Npos + Nneg;

%update features
feats = featuresAll(:,idxPrev);

% Initialize weights to equally probability
weight = zeros(Ntotal,1);

% Initialize labels for both positive and negative examples
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

% This is the adaboost process
T=40;
strongClaResult = zeros(Ntotal, 1);
alpha = zeros(T,1);
ht = zeros(4,T);
hRes = zeros(Ntotal, T);

% The adaboost iterative process
for t = 1:T
% normalize weights
weight = weight ./ sum(weight);
% get the best weak classifier and the detection result
h = getClassifier(feats, 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;
hRes(:,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 = hRes(:,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;

%This is when the adaboost stops searching for features
if posAccuracy(t)==1 && negAccuracy(t) <= 0.5
break;
end

end

% Presenting update for the 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'];
% 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
```

Script for Haar features

```
function [feats, Npos, Nneg] = getHaar(filePath)
```

```
% This is is for getting the features from all images

%The size of the images are fixed
r=20; % The no. of rows
c=40; % The no. of columns

%Setting the File paths
posFilePath = [filePath 'positive/'];
negFilePath = [filePath 'negative/'];
disp(posFilePath);
posImg = loadImagesAdaboost(posFilePath, r, c);
negImg = loadImagesAdaboost(negFilePath, r, c);

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

Nfeats=166000;
feats=zeros(Nfeats, Nimg);

for i=1:Nimg
    intImg=zeros(r+1,c+1);
    disp(i);
    if i<=size(posImg,3)
        intImg(2:r+1,2:c+1) = cumsum(cumsum(posImg(:,:,i)),2);
    else
        intImg(2:r+1,2:c+1) = cumsum(cumsum(negImg(:,:,i-size(posImg,3))),2);
    end
    feats(:,i)=computeFeature(intImg);
end

features_adaboost.features = feats;
features_adaboost.Npos = Npos;
features_adaboost.Nneg = Nneg;
% For saving test image features
save('features_adaboost_test.mat', 'features_adaboost', '-mat', '-v7.3');
% For saving training image features
%  save('features_adaboost_train.mat', 'features_adaboost', '-mat', '-v7.3');
end
```

Script for Loading Images

```
function imgs = loadImagesAdaboost(filePath, r, c)

% get the images in 'filePath'
files = dir([filePath '*.png']);
imgs=zeros(r,c,length(files));

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

```
end
end
```

Script for Computing Haar Features

```
function feat = computeFeature(I, r, c)

feat=zeros(166000,1);

%extract Horizontal features
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];
feat(cnt)=sumBox(I, rect2)-sumBox(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];
feat(cnt)=sumBox(I, rect1)-sumBox(I, rect2);
cnt=cnt+1;
end
end
end
end
```

Script for Integral Image technique

```
function [boxSum] = sumBox(I, box4)

%Given 4 corners in the integral image we have
% to calculate the sum of pixels inside the box.

row_s=box4(1);
col_s=box4(2);
w=box4(3);
h=box4(4);

A = I(row_s, col_s);
B = I(row_s, col_s + w);
C = I(row_s+h, col_s);
D = I(row_s+h, col_s+w);
```

```
boxSum = A + D - (B+C);
end
```

Script for getting strong classifier from weak classifier

```
function h = getClassifier(features, weight, label, Npos)
% This function is used for getting the classifier
% To 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; % Setting the polarity to negative
 result(idxMin+1:end) = 1;
 result(sortedIdx) = result;
 else
 p = 1; % Setting the polarity to positive
 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
```

Main Script for Testing

```
% This is the script for testing
clc
clear all

% Obtaining 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; % This is the no. of stages
% for computing accuracy for each stage
fp = zeros(S,1);
fn = zeros(S,1);
% test for each stage
for i = 1:S
 % 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 for each stage that is the no. of classifiers for each stage
 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);

 % compute false postive rate and false negative rate
 fn(i) = (Npos - sum(result(1:Npos))) / Npos;
 fp(i) = sum(result(Npos+1:end)) / Nneg;
end

%This is for checking cumulative FPR and TPR for cascaded stage.
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
end
falseneg_acc=(1-cumprod(1-fn_rate));
% Plot the accumulated FNR and FPR after that
```

Script for Classification

```
function [ result ] = adaBoostClassify( featuresAll, alpha, p, th, fIdx, T)
% th is the threshold and p is the polarity
%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)*th(t)
 weakResult(i,t) = 1; %else it will be negative class
 end
 end
end
% build strong classifier by weighted average of weak classifiers
strongCla = weakResult(:,1:T) * alpha(1:T,:);
% compute strong classifier thershold
strongTh = 0.5 * sum(alpha(1:T,1));
% get final classification result
result = zeros(Nimgs,1);
for i = 1:Nimgs
 if strongCla(i) >= strongTh
 result(i) = 1;
 end
end
end
```