Keerthi Raj Nagaraja (nagaraj1@purdue.edu)

12-04-2014

1 Face Recognition

In this part of the assignment, we use the concepts of dimensionality reduction through Principal Component Analysis (PCA) and Linear Discriminant Analysis (LDA) and the nearest-neighborhood classification for face recognition. The goal of this part is to classify an unknown face image given a database of labeled face images. The steps for recognition includes, vectorizing the images and normalizing them. Reducing the feature dimensions through PCA and LDA (two separate methods and hence different results). We then project the vectorized training as well as test images to the reduced lower dimensional sub-space. Finally, we use nearest neighborhood classification method to classify a test image based on euclidean distance in the sub-space.

1.1 PCA

The goal of the PCA is to reduce the image data dimension and make the computations more efficient. This is done by choosing the best orthogonal features (components) along which the data has the largest variances. For example, instead of representing a face image as a point in the 16384 (128x128) dimensional space, PCA allows the same face image to be represented in as small as a 10-dimensional subspace (user parameter) while still retaining the best features which represent the original data without significant error. Conceptually, this means that the eigen-vectors corresponding to the largest eigen values of the covariance matrix XX^T are the principal components that we are trying to extact. We proceed to do that with following steps:

(a) Vectorize and normalize the 128x128 training face images.

(b) Make the vectors as zero-mean vectors by subtracting global mean of all vectors since all future computations include subtracting this mean from the vectorized images. Let $\vec{x}_i$ be the vectorized and normalized zero-mean representation of the i^{th} image in the training data set. $i = [1..N]$

(c) Construct a matrix X out of these vectors by considering each vector as a column of the matrix. The size of X will be $16384 \times N$

$$X = \begin{bmatrix} \vec{x}_1 & \vec{x}_2 & ..... & \vec{x}_N \end{bmatrix}$$

(d) Since XX^T would be a matrix of size 16384×16384 which poses storage and computational issues. We compute eigen-vectors of X^TX (which is of size $N \times N$) and multiply them by X to get the eigen-vectors of XX^T. As N is very small compared to 16384, this trick is very efficient. In our case N is 630 which is equal to number of training images. We use SVD to find the eigen vectors. Let $\vec{u}_i$'s be the N eigen-vectors of X^TX. Then, eigen-vectors of XX^T will be $\vec{w}_i = X\vec{u}_i$, $i = [1..N]$

(e) Form the sub-space by choosing the first p eigen-vectors of XX^T and the sub-space matrix will be of the size $16384 \times p$

$$W = \begin{bmatrix} \vec{w}_1 & \vec{w}_2 & ..... & \vec{w}_p \end{bmatrix}$$

(f) Project training image vectors and test image vectors onto the p-dimensional sub-space. Let Y be the projected vectors of training vectors matrix X onto the sub-space and Z be the projected vectors of test vectors matrix T onto the subspace, then

$$Y = W^T X$$

and

$$Z = W^T T$$

Now, Y and Z are of size $p \times N$ where i^{th} column represents the i^{th} image in p-dimensional sub-space.

(f) Now given that PCA has done its job of reducing the dimensions of the data point, for every i^{th} vector in Z, we simply find the nearest vector j in Y and classify the i^{th} test image as the one belonging to the same class as j^{th} training image.

(g) To find the accuracy of the classification method, we use the provided labels of the test dataset and find the percentage of the images that have been labeled/classified correctly.

1.2 LDA

The goal of the LDA is again to reduce the image data dimension and make the computations more efficient. However, LDA is more robust as it finds the orthogonal directions in the original space which provides the maximal class separation. This is done by choosing the best orthogonal features (components) along which the within-class data scatter is minimum while the between-class data scatter is maximum. Conceptually, this means that we need to maximize the ratio between the between-class scatter and the within-class scatter. We proceed to do that with following steps:

(a) Vectorize and normalize the 128x128 training face images.

(b) Make the vectors as zero-mean vectors by subtracting global mean of all vectors since all future computations include subtracting this mean from the vectorized images. Let $\vec{x}_i$ be the vectorized and normalized zero-mean representation of the i^{th} image in the training data set. $i = [1..N]$

(c) Construct a matrix X out of these vectors by considering each vector as a column of the matrix. The size of X will be $16384 \times N$

$$X = \begin{bmatrix} \vec{x}_1 & \vec{x}_2 & ..... & \vec{x}_N \end{bmatrix}$$

(d) Calculate the class means for all the N classes and construct the class-mean matrix M of size $16384 \times K$ where K is the number of classes. In this case K=30

$$\vec{m}_i = \frac{1}{||C_i||}\sum_{i=1}^{||C_i||}\vec{x}_i$$

$$M = \begin{bmatrix}\vec{m}_1 & \vec{m}_2 & ..... & \vec{m}_K\end{bmatrix}$$

(e) Let S_B be the between-class scatter matrix given by,

$$S_B = \frac{1}{||C||}MM^T$$

The eigen vectors of S_B are found using the trick mentioned in PCA section by first finding the eigen vectors of M^TM and then multiplying them by M

(f) Let $Y = \begin{bmatrix}\vec{v}_1 & \vec{v}_2 & ..... & \vec{v}_K\end{bmatrix}$ be the eigen-vector matrix of S_B where vectors are sorted in descending order of their corresponding eigen vectors. Let D be the diagonal singular-value matrix of S_B

(g) Compute $Z = YD^{-1}$ which is of size $16384 \times K$. We then compute Z^TS_WZ as given below

$$Z^TS_WZ = \frac{1}{K}\sum_{i=1}^{K}\frac{1}{||C_i||}\sum_{j=1}^{||C_i||}(Z^T\vec{x}_k)(\vec{x}_k^TZ)$$

(h) We then calculate the eigen-vectors of Z^TS_WZ using SVD decomposition. So,

$$U = \begin{bmatrix}\vec{u}_1 & \vec{u}_2 & ..... & \vec{u}_K\end{bmatrix}$$

will be the eigen-vector matrix.

(i) Now, form the sub-space using the first p eigen-vectors. Let this sub-space be represented by W where

$$W = ZU = \begin{bmatrix}\vec{w}_1 & \vec{w}_2 & ..... & \vec{w}_p\end{bmatrix}$$

(j) Project training image vectors and test image vectors onto the p-dimensional sub-space. Let Y be the projected vectors of training vectors matrix X onto the sub-space and Z be the projected vectors of test vectors matrix T onto the subspace, then

$$Y = W^TX$$

and

$$Z = W^TT$$

Now, Y and Z are of size $p \times N$ where i^{th} column represents the i^{th} image in p-dimensional sub-space.

(k) Now given that LDA has done its job of reducing the dimensions of the data point, for every i^{th} vector in Z, we simply find the nearest vector j in Y and classify the i^{th} test image as the one belonging to the same class as j^{th} training image.

(l) To find the accuracy of the classification method, we use the provided labels of the test dataset and find the percentage of the images that have been labeled/classified correctly.

1.3 Results

The PCA and LDA methods for face detection have been evaluated based on the accuracies on test images. Accuracy is calculated as a fraction of test images that are being classified correctly.

1.3.1 PCA Results

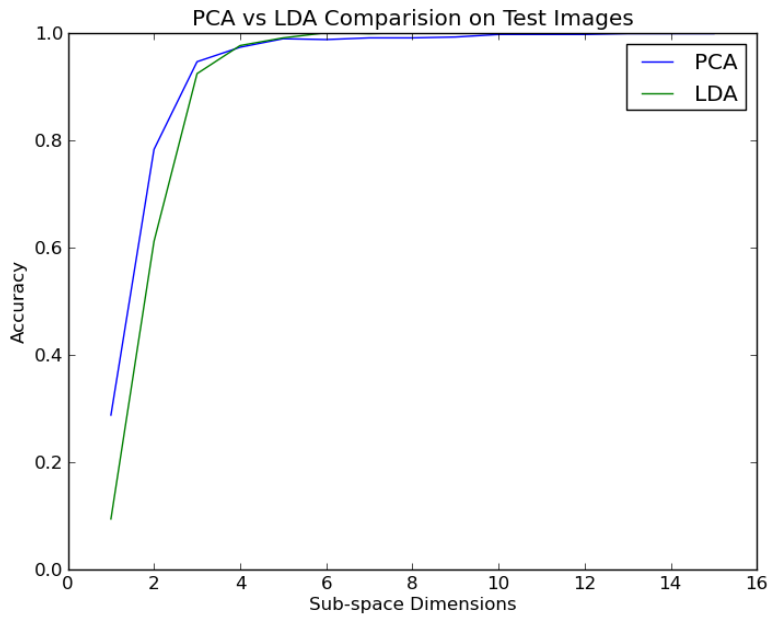
Eigen Vectors = 1, Accuracy = 0.287301587302
Eigen Vectors = 2, Accuracy = 0.78253968254
Eigen Vectors = 3, Accuracy = 0.946031746032
Eigen Vectors = 4, Accuracy = 0.973015873016
Eigen Vectors = 5, Accuracy = 0.988888888889
Eigen Vectors = 6, Accuracy = 0.987301587302
Eigen Vectors = 7, Accuracy = 0.990476190476
Eigen Vectors = 8, Accuracy = 0.990476190476
Eigen Vectors = 9, Accuracy = 0.992063492063
Eigen Vectors = 10, Accuracy = 0.996825396825
Eigen Vectors = 11, Accuracy = 0.996825396825
Eigen Vectors = 12, Accuracy = 0.996825396825
Eigen Vectors = 13, Accuracy = 0.998412698413
Eigen Vectors = 14, Accuracy = 0.998412698413
Eigen Vectors = 15, Accuracy = 0.998412698413

1.3.2 LDA Results

Eigen Vectors = 1, Accuracy = 0.0936507936508
Eigen Vectors = 2, Accuracy = 0.611111111111
Eigen Vectors = 3, Accuracy = 0.92380952381
Eigen Vectors = 4, Accuracy = 0.97619047619
Eigen Vectors = 5, Accuracy = 0.990476190476
Eigen Vectors = 6, Accuracy = 1.0
Eigen Vectors = 7, Accuracy = 0.998412698413
Eigen Vectors = 8, Accuracy = 1.0
Eigen Vectors = 9, Accuracy = 1.0

Eigen Vectors = 10, Accuracy = 1.0

Figure 1: PCA vs LDA comparision



1.4 Observations

(a) LDA converged to better accuracies faster than PCA since LDA uses most discriminating directions while choosing eigen vectors for the sub-space.

(b) LDA achieved 100% accuracy while PCA couldn't classify one of the test images correctly and achieved 99.84% accuracy.

2 Cascaded Adaboost Classifier for Car Detection

In this part of the assignment, we use the Viola and Jones Cascaded Adaboost classifier for car detection. The goal of this part is to classify whether the test image contains a car or not using a classifier given a database of labeled positive and negative training images. The steps for learning a classifier includes, extracting large number of HAAR-like features from the training images. Learning multiple weak classifiers to form a single adaboost classifier. Then, multiple such adaboost classifier stages are learnt until the desired false-positive rate is achieved on training dataset. Finally, we apply these learned classification rules to all the test images to classify them and evaluate the accuracy, True Positive, False Positive and False Negative rates.

2.1 Haar Feature Extraction

(a) We first compute the integral image for each of the input training image which will make the HAAR feature computations very efficient.

(b) The images are of 20×40 pixels size and starting from 1×2 and 2×1 haar rectangular windows to 20×40 haar rectangular windows, we compute 166000 haar feature values each corresponding to one particular window size and one particular location within the image. These features carry necessary discriminatory information about the presence or absence of the car in an image.

(c) More number of features can be extracted using different types of rectangular windows. However, due to large memory requirements, I have used only 166000 haar features for each image.

2.2 Weak Classifier

(a) We define a weak classifier to be one which classifies the data into two classes by a simple threshold on any one of the haar features.

(b) The goal is then to find the best feature-threshold pair among all the haar features and their possible thresholds which classifies the weighted train images with minimum error.

(c) Initially all the images(features) are weighted equally and a weak classifier is obtained. The weights of the training images that are misclassified will be increased and the next weak classifier work on these non-uniformly weighted training features to get the next best feature-threshold pair.

(d) The weak classifier is mathematically given by,

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

where, f is the best haar feature found, x is given image, p is the polarity and θ is the best threshold found for the haar feature f

(e) Each weak classifier is also associated with a trust value which indicates how good is the weak classifier. This is calculated based on its classification error, e.

$$\alpha = log(\frac{1}{\beta})$$

where $\beta = \frac{e}{1-e}$

The algorithm to find a weak classifier is explained in [1].

2.3 Strong Adaboost Classifier

(a) Each Adaboost classifier is made up of several weak classifiers and is boosted to be a strong classifier

(b) This process of boosting is done by adding a weak classifier to the pool of existing weak classifiers until the overall False positive rate of the strong adaboost classifier is below the required threshold. The FP rate for each stage is chosen to be 0.3

(c) The strong classifier is given by,

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

where, $h_t(x)$ is t^{th} weak classifier and α_t is the corresponding trust or belief in that weak classifier.

(d) To make sure all the positive training images are recognized as positive (TP Rate=1.0), we set the strong classifier threshold (T_s) to be the minimum among the values of $\sum_{t=1}^{T} \alpha_t h_t(x)$ for all positive training images.

2.4 Cascade of Adaboost Classifiers

(a) To achieve the important goal of very low false positive rate, Viola and Jones method uses cascade of multiple adaboost classifiers.

(b) Each stage is learnt using the training images which were classified as "True" and the goal of each stage is to eliminate false images as much as possible.

(c) Only images that passed through all the stages are considered to be true. Hence, while training the cascaded adaboost, we make sure True positive rate at each stage is 1.0 and each stage decreases false positive rate by at least 0.3. So, N adaboost stages can provide false positive rate of atleast $(0.3)^N$ which is an exponential decrease.

(d) The training process includes applying the strong adaboost classifier to the input training images and finding the number of false positives. The negative images which were classified correctly (True Negatives) are removed from the training set and the weights for the reduce training set are re-initialized to be uniform. Now, the reduced training set is used to learn multiple weak classifiers in the next stage until the false positive rate of atleast 0.3 is reached.

(e) The process of learning multiple adaboost stage continues until the overall false positive rate has reduced to 10^{-4} which is equivalent to zero classification error for the given training dataset of 710 positive and 1758 negative images.

2.5 Parameters for Training

The parameters used in the training process are,

(a) Number of positive images = 710

(b) Number of negative images = 1758

(c) Number of Haar features per image = 166000 (84000 Horizontal and 82000 Vertical features)

(d) Desired Minimum False Positve rate for each stage = 0.3 (Run 1) and 0.5 (Run 2)

(e) Desired Global False Positive rate for the entire cascade classifier = 10^{-4}

2.6 Results

The performance are evaluated based on two metrics.

(a) False Positive Rate = $\frac{\text{No. of misclassified negative images}}{\text{No. of negative images}}$

(b) False Negative Rate = $\frac{\text{No. of misclassified positive images}}{\text{No. of positive images}}$

These two metrics are obtained both while training and testing as a function of the number of stages. Below are the obtained results for two cases.

2.6.1 Run 1: With 0.3 as Desired False Positive rate for each stage

Number of Stages = 5

Testing Accuracy = 93.36 %

Testing TP = 0.8876

Testing FN = 0.1123

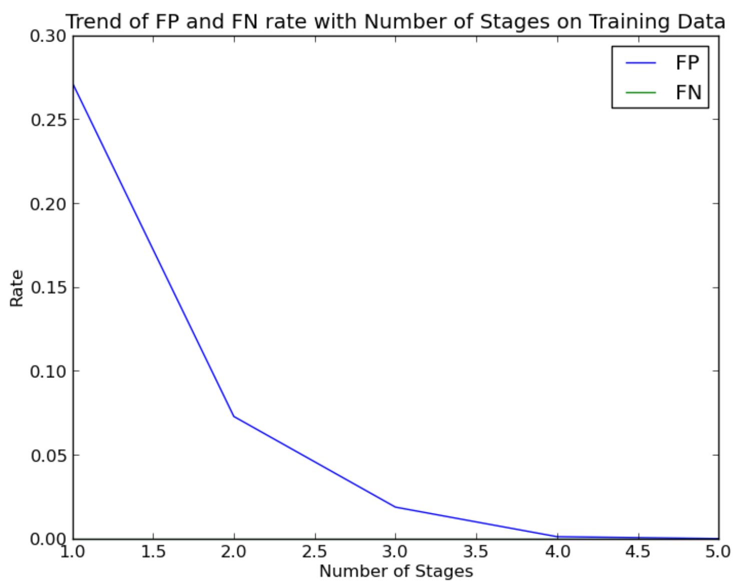
Testing FP = 0.0477

Testing TN = 0.9522

(a) Training Metrics

Stage	Number of Weak Classifiers	FP Rate for each stage	Cumulative FP rate	Number of negative training images forwarded to next stage
1	11	0.2718	0.2718	478
2	24	0.2677	0.0728	128
3	20	0.2578	0.0187	33
4	18	0.0606	0.0011	2
5	2	0.0	0.0	0

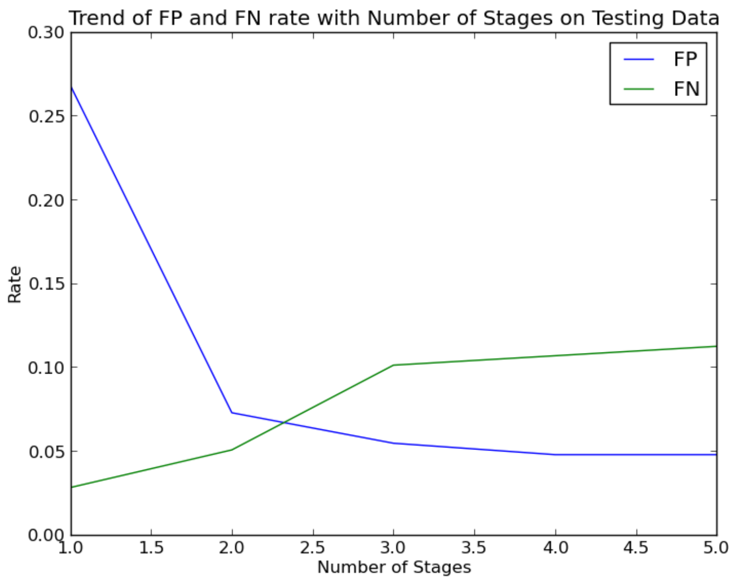
Figure 2: Trend of FP and FN rate with Number of Stages on Training Data



(b) Testing Metrics

Stage	Number of Weak Classifiers	FP Rate for each stage	Cumulative FP rate	Number of negative testing images forwarded to next stage
1	11	0.2681	0.2681	118
2	24	0.2711	0.0727	32
3	20	0.7496	0.0545	24
4	18	0.875	0.0477	21
5	2	1.0	0.0477	21

Figure 3: Trend of FP and FN rate with Number of Stages on Testing Data



2.6.2 Run 2: With 0.5 as Desired False Positive rate for each stage

Number of Stages = 8

Testing Accuracy = 94.49 %

Testing TP = 0.9213

Testing FN = 0.0786

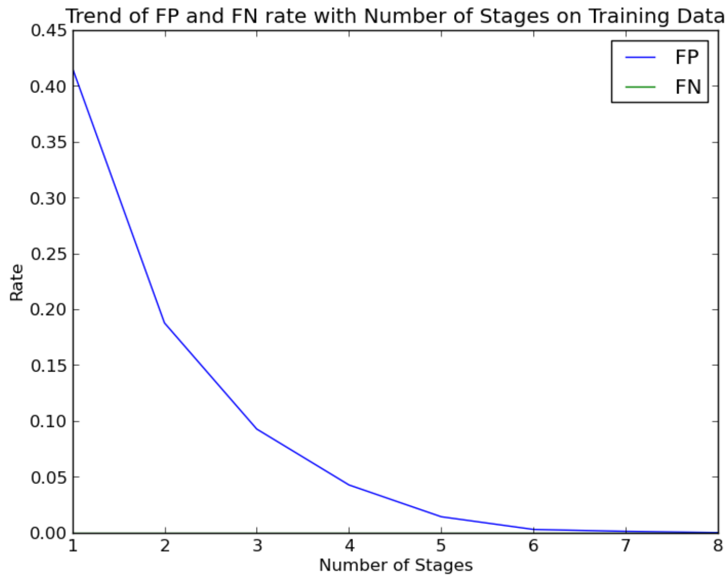
Testing FP = 0.0454

Testing TN = 0.9545

(a) Training Metrics

Stage	Number of Weak Classifiers	FP Rate for each stage	Cumulative FP rate	Number of negative training images forwarded to next stage
1	6	0.4163	0.4163	732
2	16	0.4508	0.1877	330
3	18	0.4939	0.0927	163
4	22	0.4601	0.0426	75
5	21	0.3333	0.0142	25
6	13	0.2	0.00284	5
7	5	0.4	0.00113	2
8	8	0.0	0.0	0

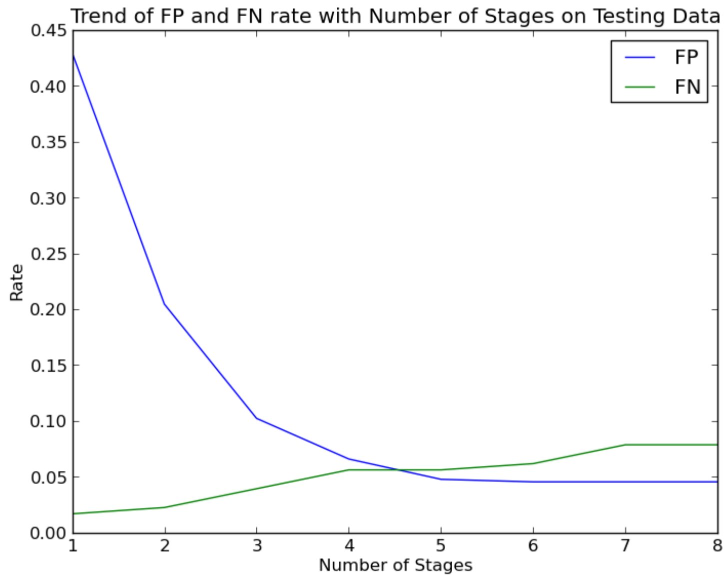
Figure 4: Trend of FP and FN rate with Number of Stages on Training Data



(b) Testing Metrics

Stage	Number of Weak Classifiers	FP Rate for each stage	Cumulative FP rate	Number of negative testing images forwarded to next stage
1	6	0.4295	0.4295	189
2	16	0.4761	0.2045	90
3	18	0.5	0.1022	45
4	22	0.6448	0.0659	29
5	21	0.7238	0.0477	21
6	13	0.9517	0.0454	20
7	5	1.0	0.0454	20
8	8	1.0	0.0454	20

Figure 5: Trend of FP and FN rate with Number of Stages on Testing Data



2.7 Observations

(a) The Adaboost classifier worked as expected and good results were observed on test images. The testing performance metrics slightly depend on the number of stages and hence on the value of desired false positive rate that we choose for each stage as observed from two different runs of the same program.

(b) More the number of stages, better the True Positive rate and accuracy on test images. With 8 stages, true positive rate increased by around 3.5%. However, False positive rate doesn't change much since in both the runs, 21 or 20 out of 440 negative images are misclassified at the end. The reason for change in TP rate is that with 0.3 as the desired FP rate for each stage, each stage is aggressive in detecting negative images and there is a higher chance that they classify some of the positive images as negative which may not be the case when the stages are not so aggressive with FP rate of 0.5.

(c) The training of the classifier requires more memory and computation time. Some measures had to be taken to ensure the program runs with given computer's memory constraints using techniques such as in-place matrix manipulation, 32-bit float data-types of numpy instead of python's default 64-bit float data-types, etc.

References

[1] Paul Viola, Michael J. Jones, *Robust Real-Time Face Detection.* International Journal of Computer Vision, 57(2), 137-154, 2004.

3 Appendix

3.1 PCA.py

```
# Importing libraries
import cv2
import cv
from math import *
from numpy import *
#from sympy import Symbol,cos,sin
from operator import *
from numpy.linalg import *
import time
import ctypes
from scipy.optimize import leastsq
from matplotlib import pyplot as plt
# Prints the numbers in float instead of scientific format
set_printoptions(suppress=True)

folder='Face Dataset/' # Dataset Folder
NUMBER_OF_SUBJECTS=30   # Number of Subjects in the dataset
NUMBER_OF_IMAGES_PER_SUBJECT=21 # Number of images per subject
MAX_NUMBER_OF_EIGEN_VECTORS=20 # The desired number of maximum eigen ...
    vectors that we want to test
#------------------------------------------------------------------------------------------------
# This function reads all the images in the given folder, converts ...
    each image to an array and
# returns a matrix of image vectors
def readImages(folder):
    imageVectors=[]
    for loopVar1 in range(NUMBER_OF_SUBJECTS):  # For all subjects
        for loopVar2 in range(NUMBER_OF_IMAGES_PER_SUBJECT): # For all ...
            Images for each subject
            img = ...
                cv2.imread(folder+str(loopVar1+1).zfill(2)+'_'+str(loopVar2+1).zfill(2)+'.png'
                0)  # Read the image
            img = asarray(img).flatten().tolist() # Flatten it as a vector
            imageVectors.append(img) # Append the vector to a matrix
    return imageVectors     # Return that matrix

#------------------------------------------------------------------------------------------------
# This function normalizes the image vectors and then subtracts mean ...
    from them.
# Returns the normalized zero-mean vectors
def normalizeVectors(vectors):
    for loopVar1 in range(len(vectors)):    # For each vector
        vectors[loopVar1]=vectors[loopVar1]/norm(vectors[loopVar1]) # ...
            Find the normalized vector

    meanVector=mean(array(vectors), 0)   # Find the mean-vector

    for loopVar1 in range(len(vectors)):    # For each vector
```

```
        vectors[loopVar1]=(array(vectors[loopVar1])-meanVector).tolist() ...
            # Subtract the mean
    meanVector=mean(array(vectors), 0)  # Just for verification, check ...
        if the mean is zero. It should be.
    print meanVector, norm(meanVector)
    return vectors      # Return the normalized zero-mean vectors

#-----------------------------------------------------------------------------
# This function takes in train and test vectors in subspace and ...
    classifies a test vector based on
# nearest neighbor classification method. Returns the classified ...
    labels for test vectors and accuracy.
def classify(trainFeatures, testFeatures):
    classifiedLabels=[]
    correctClassifications=0            # Counter for correct ...
        classifications
    for loopVar1 in range(shape(testFeatures)[1]):  # For each test vector
        testVector=(array(testFeatures[:,loopVar1])).flatten() # ...
            Convert it to an array
        querySubject=(loopVar1/NUMBER_OF_IMAGES_PER_SUBJECT)+1 # Find ...
            the true label of the test image
        minDistance=1e+10
        for loopVar2 in range(shape(trainFeatures)[1]): # For train vector
            trainVector=(array(trainFeatures[:,loopVar2])).flatten() # ...
                Convert it to an array
            distance=sqrt(sum(square(subtract(trainVector, ...
                testVector)))) # Find the euclidean distance
            if distance<minDistance:                # Check if it's ...
                the minimum so far
                minDistance=distance        # If yes, save the distance
                matchedSubject=(loopVar2/NUMBER_OF_IMAGES_PER_SUBJECT)+1 ...
                    # Save the predicted label
        if matchedSubject==querySubject:            # If the predicted ...
            label is same as true label
            correctClassifications+=1           # Increase the correct ...
                classifications count
        classifiedLabels.append(matchedSubject)
    accuracy=correctClassifications/float(shape(testFeatures)[1])   # ...
        Find the accuracy
    return classifiedLabels, accuracy   # Return the classified labels ...
        and the accuracy

#-----------------------------------------------------------------------------
# Main Code starts
trainimageVectors=readImages(folder+'train/')   # Read and vectorize ...
    all training images
testimageVectors=readImages(folder+'test/') # Read and vectorize all ...
    test images

trainimageVectors=normalizeVectors(trainimageVectors)   # Get ...
    normalized zero-mean vectors for train images
testimageVectors=normalizeVectors(testimageVectors) # Get normalized ...
    zero-mean vectors for test images

```

```
print len(trainimageVectors), type(trainimageVectors)   # Print the ... length of those vectors for debugging
print len(testimageVectors), type(testimageVectors)

X=matrix(trainimageVectors)     # Make a matrix out of all the ... training image vectors
print shape(X)              # X will be (16384 x N)
XXT=X*transpose(X)  # Fing X transpose X (Note the variables name are ... quite different as initial X in program is (Nx16384))
print shape(XXT)            # X transpose X will be (NxN)
U,D,V=linalg.svd(XXT)           # Find SVD of X transpose X
for num_eig_vec in range(1,MAX_NUMBER_OF_EIGEN_VECTORS+1):  # For ... eigen vectors from 1 to MAX, find a classifier
    eigenVectors=U[:,0:num_eig_vec]     # Pick top 'p' eigen vectors ... of X transpose X
    print shape(eigenVectors)
    W=transpose(X)*eigenVectors     # Find top 'p' eigen vectors of X ... X transpose by multiplying by X
    print shape(W)              # Size of W will be (16384 x p)
    featureVectors=transpose(W)*transpose(X) # Project the training ... images onto subspace
    print shape(featureVectors)     # Size of the new train feature ... vectors will be (px1)

    Xtest=matrix(testimageVectors)             # Make a matrix out of ... all the test image vectors
    testfeatureVectors=transpose(W)*transpose(Xtest)    # Project the ... test images onto subspace
    print shape(testfeatureVectors)     # Size of the new test feature ... vectors will be (px1)

    classifiedLabels, accuracy=classify(featureVectors, ... testfeatureVectors) # Classify the test data
    print 'Accuracy = ', accuracy, 'PCA - Eigen Vectors = ', ... num_eig_vec    # Print the accuracy
```

3.2 LDA.py

```
# Importing libraries
import cv2
import cv
from math import *
from numpy import *
#from sympy import Symbol,cos,sin
from operator import *
from numpy.linalg import *
import time
import ctypes
from scipy.optimize import leastsq
from matplotlib import pyplot as plt
# Prints the numbers in float instead of scientific format
```

```
set_printoptions(suppress=True)

folder='Face Dataset/' # Dataset Folder
NUMBER_OF_SUBJECTS=30  # Number of Subjects in the dataset
NUMBER_OF_IMAGES_PER_SUBJECT=21 # Number of images per subject
MAX_NUMBER_OF_EIGEN_VECTORS=20 # The desired number of maximum eigen ...
    vectors that we want to test

#-----------------------------------------------------------------------
# This function reads all the images in the given folder, converts ...
    each image to an array and
# returns a matrix of image vectors
def readImages(folder):
    imageVectors=[]
    for loopVar1 in range(NUMBER_OF_SUBJECTS):
        for loopVar2 in range(NUMBER_OF_IMAGES_PER_SUBJECT):
            img = ...
                cv2.imread(folder+str(loopVar1+1).zfill(2)+'_'+str(loopVar2+1).zfill(2)+'.png'
                0) # Read two images
            img = asarray(img).flatten().tolist()
            imageVectors.append(img)
    return imageVectors

#-----------------------------------------------------------------------
# This function normalizes the image vectors and then subtracts mean ...
    from them.
# Returns the normalized zero-mean vectors
def normalizeVectors(vectors):
    for loopVar1 in range(len(vectors)):    # For each vector
        vectors[loopVar1]=vectors[loopVar1]/norm(vectors[loopVar1])  # ...
            Find the normalized vector

    meanVector=mean(array(vectors), 0)  # Find the mean-vector

    for loopVar1 in range(len(vectors)):    # For each vector
        vectors[loopVar1]=(array(vectors[loopVar1])-meanVector).tolist() ...
             # Subtract the mean
    meanVector=mean(array(vectors), 0)  # Just for verification, check ...
        if the mean is zero. It should be.
    print meanVector, norm(meanVector)
    return vectors          # Return the normalized zero-mean vectors

#-----------------------------------------------------------------------
# This function takes in a full set of vectors and returns the means ...
    of each class within the input vectors
def findClassMean(vectors):
    classmeanMatrix=[]
    vectors=matrix(array(vectors))  # Convert the received vectors to ...
        matrix
    for loopVar1 in range(NUMBER_OF_SUBJECTS):  # For each class (subject)
        classmeanMatrix.append(mean(array(vectors[(loopVar1*NUMBER_OF_IMAGES_PER_SUBJECT):((lo
            # Find the mean and append it to class mean array
    classmeanMatrix=matrix(array(classmeanMatrix))  # Convert the ...
        class mean array to a matrix
```

```
    return classmeanMatrix      # Return that matrix

#-----------------------------------------------------------------------------------------------
# This function takes in Z and the training vectors. Computes Z ...
    transpose S Z.
def computeZTSwZ(Z, vectors):
    ZT=transpose(Z)             # Find transpose of Z
    ZTSwZ=matrix(zeros((NUMBER_OF_SUBJECTS, NUMBER_OF_SUBJECTS)))   # ...
        Create a matrix for Z transpose S Z
    for loopVar1 in range(NUMBER_OF_SUBJECTS):      # For each class ...
        (subject)
        temp_matrix=matrix(zeros((NUMBER_OF_SUBJECTS, ...
            NUMBER_OF_SUBJECTS))) # Create a temporary matrix
        for loopVar2 in range(NUMBER_OF_IMAGES_PER_SUBJECT):    # For ...
            each image in each subject
            vector=matrix(array(vectors[(loopVar1*NUMBER_OF_IMAGES_PER_SUBJECT)+loopVar2])) ..
                # Get its vector
            temp_matrix+=(ZT*transpose(vector)*vector*Z)    # Find (Z ...
                transpose x) times (x transpose Z)
        temp_matrix=temp_matrix/NUMBER_OF_IMAGES_PER_SUBJECT    # ...
            Normalize the temporary matrix by class size
        ZTSwZ+=temp_matrix              # Update Z transpose S Z
    ZTSwZ=ZTSwZ/len(vectors)                # Normalize the Z ...
        transpose S Z matrix by number of classes
    return ZTSwZ                # Return Z transpose S Z

#-----------------------------------------------------------------------------------------------
# This function takes in train and test vectors in subspace and ...
    classifies a test vector based on
# nearest neighbor classification method. Returns the classified ...
    labels for test vectors and accuracy.
def classify(trainFeatures, testFeatures):
    classifiedLabels=[]
    correctClassifications=0            # Counter for correct ...
        classifications
    for loopVar1 in range(shape(testFeatures)[1]):  # For each test vector
        testVector=(array(testFeatures[:,loopVar1])).flatten()  # ...
            Convert it to an array
        querySubject=(loopVar1/NUMBER_OF_IMAGES_PER_SUBJECT)+1  # Find ...
            the true label of the test image
        minDistance=1e+10
        for loopVar2 in range(shape(trainFeatures)[1]): # For train vector
            trainVector=(array(trainFeatures[:,loopVar2])).flatten() # ...
                Convert it to an array
            distance=sqrt(sum(square(subtract(trainVector, ...
                testVector)))) # Find the euclidean distance
            if distance<minDistance:            # Check if it's the ...
                minimum so far
                minDistance=distance            # If yes, save the ...
                    distance
                matchedSubject=(loopVar2/NUMBER_OF_IMAGES_PER_SUBJECT)+1 ...
                    # Save the predicted label
        if matchedSubject==querySubject:            # If the predicted ...
            label is same as true label
```

```
            correctClassifications+=1           # Increase the correct ...
                classifications count
        classifiedLabels.append(matchedSubject)
    accuracy=correctClassifications/float(shape(testFeatures)[1])   # ...
        Find the accuracy
    return classifiedLabels, accuracy   # Return the classified labels ...
        and the accuracy

#-----------------------------------------------------------------------------------------
# Main Code starts
trainimageVectors=readImages(folder+'train/')   # Read and vectorize ...
    all training images
testimageVectors=readImages(folder+'test/') # Read and vectorize all ...
    test images

trainimageVectors=normalizeVectors(trainimageVectors)   # Get ...
    normalized zero-mean vectors for train images
testimageVectors=normalizeVectors(testimageVectors) # Get normalized ...
    zero-mean vectors for test images

print len(trainimageVectors), type(trainimageVectors)   # Print the ...
    length of those vectors for debugging
print len(testimageVectors), type(testimageVectors)

X=matrix(trainimageVectors)         # Make a matrix out of all the ...
    training image vectors
M=findClassMean(trainimageVectors)      # Find the class mean matrix, M
print shape(M)                      # Size of M will be (16384 x K)
MMT=(M*transpose(M))/NUMBER_OF_SUBJECTS     # Find M transpose M
print shape(MMT)                    # Size of M transpose M will be (K x K)
U,D,V=linalg.svd(MMT)                   # Find SVD of M transpose M
Y=transpose(M)*U                    # Find eigen vectors of M M transpose
print shape(Y)                      # Size of Y (eigen vector matrix) will ...
    be (16384 x K)
Diag=matrix(zeros((len(D),len(D))))     # Find Diagonal matrix D using ...
    a for-loop
for loopVar1 in range(len(D)):
    Diag[loopVar1, loopVar1]=D[loopVar1]
D=Diag
print shape(D)                      # D is of size (K x K)
Z=Y*inv(D)                  # Find Z
print shape(Z)                      # Z is of size (16384 x K)
ZTSwZ=computeZTSwZ(Z, trainimageVectors)    # Compute Z transpose S Z
print shape(ZTSwZ)                  # Z transpose S Z is of size (K x K)
U,D,V=linalg.svd(ZTSwZ)                 # Find SVD of Z transpose S Z
for loopVar1 in range(shape(U)[1]):     # Sort the columns in U so ...
    that eigen-vector with lowest eigen value is first
    temp_column=U[:,loopVar1]
    U[:,loopVar1]=U[:,shape(U)[1]-1-loopVar1]
    U[:,shape(U)[1]-1-loopVar1]=temp_column
print shape(U)                      # U is of size (K x K)

for num_eig_vec in range(1, MAX_NUMBER_OF_EIGEN_VECTORS+1): # For ...
    eigen vectors from 1 to MAX, find a classifier
```

```
    eigenVectors=U[:,0:num_eig_vec]      # Pick first 'p' eigen vectors ...
        of Z transpose S Z
    print shape(eigenVectors)
    W=Z*eigenVectors              # Find first 'p' eigen vectors which ...
        forms the subspace
    print shape(W)                # Size of W will be (16384 x p)
    featureVectors=transpose(W)*transpose(X)     # Project the training ...
        images onto subspace
    print shape(featureVectors)          # Size of the new train ...
        feature vectors will be (px1)

    Xtest=matrix(testimageVectors)           # Make a matrix out of all ...
        the test image vectors
    testfeatureVectors=transpose(W)*transpose(Xtest)     # Project the ...
        test images onto subspace
    print shape(testfeatureVectors)          # Size of the new test ...
        feature vectors will be (px1)

    classifiedLabels, accuracy=classify(featureVectors, ...
        testfeatureVectors) # Classify the test data
    print 'Accuracy = ', accuracy, 'LDA - Eigen Vectors = ', ...
        num_eig_vec     # Print the accuracy
```

3.3 Adaboost.py

```
# Importing libraries
import cv2
import cv
from math import *
from numpy import *
#from sympy import Symbol,cos,sin
from operator import *
from numpy.linalg import *
import time
import ctypes
from scipy.optimize import leastsq
from matplotlib import pyplot as plt

# Prints the numbers in float instead of scientific format
set_printoptions(suppress=True)

folder='Car Dataset/'            # Dataset folder
NUMBER_OF_TRAIN_POS_IMAGES=710       # Number of postive training images
NUMBER_OF_TRAIN_NEG_IMAGES=1758      # Number of negative training images
TOTAL_NUMBER_OF_TRAIN_IMAGES=NUMBER_OF_TRAIN_POS_IMAGES+NUMBER_OF_TRAIN_NEG_IMAGES  ...
    # Total training images

NUMBER_OF_TEST_POS_IMAGES=178        # Number of postive test images
NUMBER_OF_TEST_NEG_IMAGES=440        # Number of negative test images
TOTAL_NUMBER_OF_TEST_IMAGES=NUMBER_OF_TEST_POS_IMAGES+NUMBER_OF_TEST_NEG_IMAGES ...
        # Total test images
```

```
TEST_POS_IMG_NUMBERING_OFFSET=710   # Numbering offset while image reading
TEST_NEG_IMG_NUMBERING_OFFSET=1758

REQUIRED_FP_RATE=0.0001         # Desired FP rate for overall classifier
FP_RATE_FOR_EACH_STAGE=0.5      # Desired FP rate for each stage

#
# This function reads in each image and calls another function to ...
    compute its HAAR features.
# Stores all the HAAR features of all the images in a matrix.
def readImageHaar(folder, NUMBER_OF_POS_IMAGES, NUMBER_OF_NEG_IMAGES, ...
    OFFSET):
    posfolder=folder+'positive/'            # Folder to read positive ...
        images from
    for loopVar1 in range(NUMBER_OF_POS_IMAGES):    # For each ...
        positive image
        img = ...
            cv2.imread(posfolder+str(loopVar1+1+OFFSET).zfill(6)+'.png', ...
            0) # Read the image
        scalingFactor=norm(img.flatten())       # Normalize the image
        if scalingFactor!=0:
            img=img/scalingFactor
        integralImg=zeros((img.shape[0]+1, img.shape[1]+1)) # Find the ...
            integral image
        integralImg[1:,1:]=cumsum(cumsum(img, axis=0, dtype=float64), ...
            axis=1, dtype=float64)
        features=calcHaarFeatures(integralImg)      # Calculate HAAR ...
            features using the integral image
        features.append(1)              # Append the true label along ...
            with HAAR features array
        sampleWeights[loopVar1, 0]=1/float(2*NUMBER_OF_POS_IMAGES)  # ...
            Assign uniform weight for this image
        haarFeatures[loopVar1,:]=array(features)        # Store the ...
            HAAR features along with label as a row in the matrix
        print 'Reading Pos Images', loopVar1    # printing the index ...
            of the image, for debugging

    negfolder=folder+'negative/'        # Folder to read negative ...
        images from
    for loopVar1 in range(NUMBER_OF_NEG_IMAGES):    # For each ...
        negative image
        img = ...
            cv2.imread(negfolder+str(loopVar1+1+OFFSET).zfill(6)+'.png', ...
            0) # Read the image
        scalingFactor=norm(img.flatten())       # Normalize the image
        if scalingFactor!=0:
            img=img/scalingFactor
        integralImg=zeros((img.shape[0]+1, img.shape[1]+1))      # Find ...
            the integral image
        integralImg[1:,1:]=cumsum(cumsum(img, axis=0, dtype=float64), ...
            axis=1, dtype=float64)
        features=calcHaarFeatures(integralImg)      # Calculate HAAR ...
            features using the integral image
```

```
        features.append(0)          # Append the true label along with ...
            HAAR features array
        sampleWeights[loopVar1+NUMBER_OF_POS_IMAGES, ...
            0]=1/float(2*NUMBER_OF_NEG_IMAGES) # Assign uniform weight ...
            for this image
        haarFeatures[loopVar1+NUMBER_OF_POS_IMAGES, :]=array(features) ...
            # Store the HAAR features along with label as a row in the ...
            matrix
        print 'Reading Neg Images', loopVar1    # printing the index ...
            of the image, for debugging

    return 0            # Returns nothing as it works with a global ...
        HAAR feature Matrix

#
# This function takes in an integral image and calculates 166000 haar ...
    features and returns them as a list
def calcHaarFeatures(integralImg):
    features=[]
    for loopVar1 in range(1, integralImg.shape[0]):         # For each ...
        type of horizontal HAAR window
        for loopVar2 in range(2, integralImg.shape[1], 2):

            for loopVar3 in range(0, integralImg.shape[0]-1):   # For ...
                each position of the window in the image
                for loopVar4 in range(0, integralImg.shape[1]-1):
                    if ...
                        ((loopVar4+(loopVar2/2))<integralImg.shape[1])and((loopVar4+loopVar2)<
                                # Check if the window is ...
                        within the image boundaries
                        A=[loopVar3, loopVar4]          # Get all 6 ...
                            corners of the HAAR rectangle
                        B=[loopVar3, loopVar4+(loopVar2/2)]
                        C=[loopVar3, loopVar4+loopVar2]
                        D=[loopVar3+loopVar1, loopVar4+loopVar2]
                        E=[loopVar3+loopVar1, loopVar4+(loopVar2/2)]
                        F=[loopVar3+loopVar1, loopVar4]
                        feature=-integralImg[A[0], ...
                            A[1]]+(2*integralImg[B[0], ...
                            B[1]])-integralImg[C[0], ...
                            C[1]]+integralImg[D[0], ...
                            D[1]]-(2*integralImg[E[0], ...
                            E[1]])+integralImg[F[0], F[1]]    # ...
                            Calculate the HAAR feature using 6 summations
                        features.append(feature)    # Append the ...
                            feature to the entire list of features

    for loopVar1 in range(2, integralImg.shape[0], 2):      # For each ...
        type of vertical HAAR window
        for loopVar2 in range(1, integralImg.shape[1]):

            for loopVar3 in range(0, integralImg.shape[0]-1):   # For ...
                each position of the window in the image
                for loopVar4 in range(0, integralImg.shape[1]-1):
```

```
                    if ...
                        ((loopVar4+loopVar2)<integralImg.shape[1])and((loopVar3+(loopVar1/2))<
                                        # Check if the window is ...
                        within the image boundaries
                        A=[loopVar3, loopVar4]          # Get all 6 ...
                            corners of the HAAR rectangle
                        B=[loopVar3, loopVar4+loopVar2]
                        C=[loopVar3+(loopVar1/2), loopVar4+loopVar2]
                        D=[loopVar3+loopVar1, loopVar4+loopVar2]
                        E=[loopVar3+loopVar1, loopVar4]
                        F=[loopVar3+(loopVar1/2), loopVar4]
                        feature=-(-integralImg[A[0], ...
                            A[1]]+integralImg[B[0], ...
                            B[1]]-(2*integralImg[C[0], ...
                            C[1]])+integralImg[D[0], ...
                            D[1]]-integralImg[E[0], ...
                            E[1]]+(2*integralImg[F[0], F[1]])) # ...
                            Calculate the HAAR feature using 6 summations
                        features.append(feature)    # Append the ...
                            feature to the entire list of features

    return features         # Return the computed list of featues for ...
        an image

#-------------------------------------------------------------------------------
# This function learns a weak classifier based on the Haar Feature ...
    Matrix and corresponding weights (both global variables)
# Returns the weak classifier's feature index, threshold, polarity, ...
    trust and beta
def learnWeakClassifier():
    sampleWeights[:, :]=sampleWeights[:, :]/sum(sampleWeights[:, ...
        :])    # Normalize the weights

    TPlus=0
    TMinus=0
    for loopVar1 in range(haarFeatures.shape[0]):           # This ...
        loop finds TPlus and TMinus values
        if haarFeatures[loopVar1, haarFeatures.shape[1]-1]==1:
            TPlus+=sampleWeights[loopVar1, 0]
        else:
            TMinus+=sampleWeights[loopVar1, 0]

    globalErrors=[]
    globalPolarities=[]
    globalThresholds=[]             # This loop finds the best ...
        feature-threshold pair
    for loopVar1 in range(haarFeatures.shape[1]-1): # For each HAAR ...
        feature
        subMatrix=hstack((haarFeatures[:, loopVar1:(loopVar1+1)], ...
            haarFeatures[:, ...
            haarFeatures.shape[1]-1:haarFeatures.shape[1]], ...
            sampleWeights[:, :]))                       # Extract the ...
            feature column, its labels and weights
```

```
        subMatrix=matrix(sorted(array(subMatrix), ...
            key=itemgetter(0)))   # Sort the sub-matrix based on ...
            feature values
        SPlus=TPlus                       # Splus starts from TPlus
        SMinus=TMinus                         # SMinus starts from TMinus
        errors=[]
        polarities=[]

        for loopVar2 in range(subMatrix.shape[0]):  # This loop ...
            calculates SPlus and SMinus for each possible threshold
            if subMatrix[loopVar2, subMatrix.shape[1]-2]==1:    # If ...
                true label is 1, SPlus is decremented
                SPlus-=subMatrix[loopVar2, subMatrix.shape[1]-1]
            else:                             # If true label is 0, ...
                SMinus is decremented
                SMinus-=subMatrix[loopVar2, subMatrix.shape[1]-1]
            if (SPlus+TMinus-SMinus)<(SMinus+TPlus-SPlus):      # Find ...
                the polarity and the error
                errors.append(SPlus+TMinus-SMinus)
                polarities.append(1)
            else:
                errors.append(SMinus+TPlus-SPlus)
                polarities.append(-1)

        minerror=min(errors)             # Find the minimum error among ...
            all errors for each threshold
        globalErrors.append(minerror)
        globalPolarities.append(polarities[errors.index(minerror)]) # ...
            Find corresponding polarity
        globalThresholds.append(subMatrix[errors.index(minerror), ...
            0])    # Find corresponding threshold

    finalError=min(globalErrors)        # Find the minimum error among ...
        all errors for each feature
    featureIndex=globalErrors.index(finalError) # Find the best ...
        feature index
    featurePolarity=globalPolarities[globalErrors.index(finalError)] # ...
        Find the corresponding polarity
    featureThreshold=globalThresholds[globalErrors.index(finalError)] ...
        # Find the corresponding threshold
    # At this point, we are done finding the weak classifer

    beta=finalError/float(1-finalError)     # Find the beta value for ...
        the weak classifier
    print 'beta=', beta
    if beta==0 or beta<0:   # If beta is zero or less than zero ...
        (because of floating point issues), assign high trust value
        featureTrust=1e+8
    else:
        featureTrust=log(1/beta)    # Else, find the actual trust value

    # Now, use the weak classifier to classify the training images and ...
        increase weights of misclassified images
    for loopVar1 in range(haarFeatures.shape[0]):       # For each image
```

```
        if (featurePolarity*haarFeatures[loopVar1, featureIndex]) ≤ ...
            (featurePolarity*featureThreshold):
            if haarFeatures[loopVar1, haarFeatures.shape[1]-1]==1: # ...
                If predicted and true labels are same
                weightMultiple=beta         # Multiplying factor will ...
                    be 'beta'
            else:                       # If predicted and true labels are ...
                not same
                weightMultiple=1            # Weights doesn't decrease
        else:
            if haarFeatures[loopVar1, haarFeatures.shape[1]-1]==0:  # ...
                If predicted and true labels are same
                weightMultiple=beta         # Multiplying factor will ...
                    be 'beta'
            else:                       # If predicted and true labels are ...
                not same
                weightMultiple=1            # Weights doesn't decrease
        sampleWeights[loopVar1, 0]=sampleWeights[loopVar1, ...
            0]*weightMultiple    # Update the weights for next iteration

    return featureIndex, featurePolarity, featureThreshold, ...
        featureTrust, beta  # Return the weak classifier information

#------------------------------------------------------------------------------------------------
# This function learns a strong classifier based on the Haar Feature ...
    Matrix and corresponding weights (both global variables)
# This function calls 'learnWeakClassifier' function multiple times ...
    until the desired FP rate for each stage is achieved
# Returns the information about the strong classifier learned
def learnStrongClassifier():
    global haarFeatures, sampleWeights
    FP=1.0              # Before learning any weak classifier, the FP ...
        for this stage will be 1.0
    weakClassifierIndices=[]
    weakClassifierPolarities=[]
    weakClassifierThresholds=[]
    weakClassifierTrusts=[]

    while (FP ≥ FP_RATE_FOR_EACH_STAGE):    # Learn weak classifier ...
        until desired FP rate for this stage is achieved
        print '*****************Learning Weak ...
            Classifier******************'
        featureIndex, featurePolarity, featureThreshold, featureTrust, ...
            beta=learnWeakClassifier()   # Learn a weak classifier
        weakClassifierIndices.append(featureIndex)      # Store the ...
            new weak classifier's information
        weakClassifierPolarities.append(featurePolarity)
        weakClassifierThresholds.append(featureThreshold)
        weakClassifierTrusts.append(featureTrust)
        print 'Weak Classifier ', len(weakClassifierIndices)
        print featureIndex, featurePolarity, featureThreshold, ...
            featureTrust

```

```
        # We use the set of weak classifiers learned to find strong ...
            classifier threshold and FP rate
        weightedDecisionsforPositives=[]
        TotalPositives=0                    # This loop finds the strong ...
            classifier threshold so that TP=1.0
        for loopVar1 in range(haarFeatures.shape[0]):    # For each image
            if haarFeatures[loopVar1, haarFeatures.shape[1]-1]==1:  # ...
                If true label is 1 (positive image)
                weightedDecision=0
                TotalPositives+=1                 # Increment the number ...
                    of total positives
                for loopVar0 in range(len(weakClassifierIndices)):  # ...
                    For each weak classifier
                    if ...
                        (weakClassifierPolarities[loopVar0]*haarFeatures[loopVar1, ...
                        weakClassifierIndices[loopVar0]]) ≤ ...
                        (weakClassifierPolarities[loopVar0]*weakClassifierThresholds[loopVar0]
                        # Find weak classifier's decision
                        weightedDecision+=weakClassifierTrusts[loopVar0]*1 ...
                            # Find weighted summation of such decisions
                weightedDecisionsforPositives.append(weightedDecision)  ...
                    # Store the decision of strong classifier for all ...
                    positive images
        strongClassifierThreshold=min(weightedDecisionsforPositives)    ...
            # Strong classifier threshold will be minimum among all ...
            weighted decisions. This makes sure TP for each strong ...
            classifier is 1.0

        print 'Strong Classifier Threshold', strongClassifierThreshold

        falsePositives=0
        TotalNegatives=0
        TrueNegativeIndices=[]          # This loop finds the number ...
            false positives and the true negatives
        for loopVar1 in range(haarFeatures.shape[0]):    # For each image
            if haarFeatures[loopVar1, haarFeatures.shape[1]-1]==0:  # ...
                If true label is 0 (negative image)
                TotalNegatives+=1
                weightedDecision=0
                for loopVar0 in range(len(weakClassifierIndices)):  # ...
                    For each weak classifier
                    if ...
                        (weakClassifierPolarities[loopVar0]*haarFeatures[loopVar1, ...
                        weakClassifierIndices[loopVar0]]) ≤ ...
                        (weakClassifierPolarities[loopVar0]*weakClassifierThresholds[loopVar0]
                            # Find weak classifier's decision
                        weightedDecision+=weakClassifierTrusts[loopVar0]*1 ...
                            # Find weighted summation of such decisions

                if weightedDecision ≥ strongClassifierThreshold: # If ...
                    summation is greater than threshold
                    falsePositives+=1           # Declare it as a ...
                        false positive
                else:                           # Otherwise
```

```
                    TrueNegativeIndices.append(loopVar1)    # Save it ...
                        as true negative in order to discard it later
        FP=falsePositives/float(TotalNegatives)     # Find FP rate for ...
            this stage
        print 'FP=', FP
        if beta==0 or beta=='nan': # If the latest weak classifier had ...
            zero error, then we proceed with next stage, so that the ...
            weights which are currently all zeros (because of beta) ...
            will be re-initialized in the next stage
            break

    #--------------------------------------------------------------------#
    # This part of code removes the haar features of True Negatives ...
        from the matrix in-place so that the matrix is not duplicated ...
        and memory shortage issues doesn't occur. Idea is to move all ...
        the unwanted rows to the end of the matrix and resize the matrix.
    IndicestoRemove=[x for x in TrueNegativeIndices if ...
        x<(haarFeatures.shape[0]-len(TrueNegativeIndices))] # Find ...
        rows to remove
    IndicestoReplace=[]

    # This loop finds the rows at the end of the matrix which can be ...
        used for replacement
    IndextoReplace=(haarFeatures.shape[0]-len(TrueNegativeIndices))
    for loopVar1 in range(len(IndicestoRemove)):    # For each row to ...
        be removed
        replaceFound=0
        while(replaceFound==0):         # Find a replacement row at ...
            the end of the matrix
            if not(IndextoReplace in TrueNegativeIndices):
                IndicestoReplace.append(IndextoReplace) # Remember ...
                    that replacement row
                replaceFound=1
            IndextoReplace+=1

    # This loop exchanges rows to be removed with rows at the end of ...
        the matrix
    for loopVar1 in range(len(IndicestoRemove)):        # For each row ...
        to be removed
        rowtoKeep=array(haarFeatures[IndicestoReplace[loopVar1], ...
            :])    # Get the row to be kept but it is at the end of ...
            the matrix
        rowtoDelete=array(haarFeatures[IndicestoRemove[loopVar1], ...
            :])   # Get the row to be removed
        haarFeatures[IndicestoReplace[loopVar1], :]=rowtoDelete # Put ...
            the row to be removed at the other row's place
        haarFeatures[IndicestoRemove[loopVar1], :]=rowtoKeep    # Put ...
            the row to be kept at the removed row's place

        tempValue=sampleWeights[IndicestoReplace[loopVar1], 0]      # ...
            Similarly, exchange the weights array as well
        sampleWeights[IndicestoReplace[loopVar1], ...
            0]=sampleWeights[IndicestoRemove[loopVar1], 0]
        sampleWeights[IndicestoRemove[loopVar1], 0]=tempValue
```

```

    # At this point all the rows to be removed are at the end of the ...
        matrix and other retained rows are swapped.
    # We simply resize the HAAR feature matrix and corresponding ...
        weights array so that the reduced feature set is used by next ...
        stage
    haarFeatures.resize((haarFeatures.shape[0]-len(TrueNegativeIndices), ...
        166001), refcheck=False )
    sampleWeights.resize((sampleWeights.shape[0]-len(TrueNegativeIndices), ...
        1), refcheck=False)
    #------------------------------------------------------------------#

    # This loop re-initializes the weights based on Total positives ...
        and False positives in the reduced feature
    for loopVar1 in range(haarFeatures.shape[0]):   # For each image
        if haarFeatures[loopVar1, haarFeatures.shape[1]-1]==1:
            sampleWeights[loopVar1, 0]=1/float(2*TotalPositives)    # ...
                Re-initialize weight
        else:
            sampleWeights[loopVar1, 0]=1/float(2*falsePositives)    # ...
                Re-initialize weight

    return [weakClassifierIndices, weakClassifierPolarities, ...
        weakClassifierThresholds, weakClassifierTrusts, FP, ...
        strongClassifierThreshold]     # Return all information of ...
        this learned strong classifier

#------------------------------------------------------------------------------------------
# This function learns a cascaded adaboost classifier by simply ...
    calling 'learnStrongClassifier'
# until the desired global FP rate is achieved. Returns all the strong ...
    classifiers learned.
def learnCascadeClassifier():
    strongClassifiers=[]
    globalFalsePositiveRate=1.0     # The global false positive rate ...
        will be 1.0 initially
    while (globalFalsePositiveRate > REQUIRED_FP_RATE): # Repeat until ...
        desired overall FP rate is achieved
        print '**********************Learning Strong ...
            Classifier**************************'
        strongClassifer=learnStrongClassifier()     # Learn a strong ...
            classifier (one stage)
        globalFalsePositiveRate*=strongClassifer[4] # Update global ...
            false positive rate

        print 'GFPR=', globalFalsePositiveRate
        strongClassifiers.append(strongClassifer)   # Store the ...
            learned strong classifier's information
        print strongClassifer, len(strongClassifiers)   # Print the ...
            number of strong classifiers learned so far

    return strongClassifiers        # Return all the information about ...
        all strong classifiers (stages) learned

```

```
#
# This function takes in a learned cascaded adaboost classifier and ...
    the test dataset and classifies the test images
# Returns the Final Accuracy, TP, FN, FP, TN and stage-wise TP, FN, ...
    FP, TN values.
def classifyTestImages(finalCascadeClassifier, folder, ...
    NUMBER_OF_POS_IMAGES, NUMBER_OF_NEG_IMAGES, OFFSET_POS, OFFSET_NEG):
    classifiedLabels=[]          # Initialize few variables
    correctClassifications=0
    TruePositives=0
    FalseNegatives=0
    FalsePositives=0
    TrueNegatives=0
    eachStageDecisions=[]

    posfolder=folder+'positive/'        # Folder for positive test images
    print '*********************Classifying Positive Test ...
        Images*************************'
    for loopVar1 in range(NUMBER_OF_POS_IMAGES):    # For each ...
        positive image
        img = ...
            cv2.imread(posfolder+str(loopVar1+1+OFFSET_POS).zfill(6)+'.png', ...
            0) # Read the image
        scalingFactor=norm(img.flatten())       # Normalize the image
        if scalingFactor!=0:
            img=img/scalingFactor
        integralImg=zeros((img.shape[0]+1, img.shape[1]+1)) # Find the ...
            integral image
        integralImg[1:,1:]=cumsum(cumsum(img, axis=0, dtype=float64), ...
            axis=1, dtype=float64)
        features=calcHaarFeatures(integralImg)  # Calculate HAAR features

        stageLevelDecisions=[]

        # These nested loops apply cascaded adaboost to the test image ...
            and classifies them
        for loopVar2 in range(len(finalCascadeClassifier)): # For each ...
            stage
            strongClassifier=finalCascadeClassifier[loopVar2]   # Get ...
                the strong classifier's info
            weightedDecision=0
            for loopVar3 in range(len(strongClassifier[0])):    # For ...
                each weak classifier
                if ...
                    (strongClassifier[1][loopVar3]*features[strongClassifier[0][loopVar3]]) ..
                    ≤ ...
                    (strongClassifier[1][loopVar3]*strongClassifier[2][loopVar3]): ...
                            # Find the decision
                    weightedDecision+=strongClassifier[3][loopVar3]*1   ...
                        # Find weighted decision

            if weightedDecision ≥ strongClassifier[5]:# If weighted ...
                decision is greater than strong classifier's threshold
```

```
                stageLevelDecisions.append(1)       # Declare the ...
                    image to be Positive at this stage
            else:                           # Else,
                stageLevelDecisions.append(0)       # Declare the ...
                    image to be negative at this stage

        eachStageDecisions.append(stageLevelDecisions)      # Save all ...
            the stagelevel decisions

        if all(stageLevelDecisions):        # If all stages said, ...
            "Positive", declare positive
            TruePositives+=1
            classifiedLabels.append(1)
            correctClassifications+=1
        else:                       # Otherwise, declare negative
            FalseNegatives+=1
            classifiedLabels.append(0)


    negfolder=folder+'negative/'        # Folder for negative test images
    print '*********************Classifying Negative Test ...
        Images*************************'
    for loopVar1 in range(NUMBER_OF_NEG_IMAGES):    # For each ...
        negative image
        img = ...
            cv2.imread(negfolder+str(loopVar1+1+OFFSET_NEG).zfill(6)+'.png', ...
            0) # Read the image
        scalingFactor=norm(img.flatten())       # Normalize the image
        if scalingFactor!=0:
            img=img/scalingFactor
        integralImg=zeros((img.shape[0]+1, img.shape[1]+1)) # Find the ...
            integral image
        integralImg[1:,1:]=cumsum(cumsum(img, axis=0, dtype=float64), ...
            axis=1, dtype=float64)
        features=calcHaarFeatures(integralImg)  # Calculate HAAR features

        # These nested loops apply cascaded adaboost to the test image ...
            and classifies them
        stageLevelDecisions=[]
        for loopVar2 in range(len(finalCascadeClassifier)): # For each ...
            stage
            strongClassifier=finalCascadeClassifier[loopVar2]   # Get ...
                the strong classifier's info
            weightedDecision=0
            for loopVar3 in range(len(strongClassifier[0])):    # For ...
                each weak classifier
                if ...
                    (strongClassifier[1][loopVar3]*features[strongClassifier[0][loopVar3]]) ...
                    ≤ ...
                    (strongClassifier[1][loopVar3]*strongClassifier[2][loopVar3]): ...
                                # Find the decision
                    weightedDecision+=strongClassifier[3][loopVar3]*1   ...
                        # Find weighted decision

```

```
            if weightedDecision ≥ strongClassifier[5]:# If weighted ...
                decision is greater than strong classifier's threshold
                stageLevelDecisions.append(1)   # Declare the image to ...
                    be Positive at this stage
            else:                       # Else,
                stageLevelDecisions.append(0)   # Declare the image to ...
                    be negative at this stage

        eachStageDecisions.append(stageLevelDecisions)  # Save all the ...
            stagelevel decisions

        if all(stageLevelDecisions):            # If all stages said, ...
            "Positive", declare positive
            FalsePositives+=1
            classifiedLabels.append(1)
        else:                           # Otherwise, declare negative
            TrueNegatives+=1
            classifiedLabels.append(0)
            correctClassifications+=1


    #------------------------------------------------------------------------
    # This section of the code computes all performance metrics ...
        (global and stage-wise)
    eachStageDecisions=array(eachStageDecisions)
    validity=ones((eachStageDecisions.shape[0],1))
    eachStageDecisions=hstack((eachStageDecisions, validity))

    #print eachStageDecisions
    TP=zeros((1, eachStageDecisions.shape[1]-1))[0].tolist()
    FN=zeros((1, eachStageDecisions.shape[1]-1))[0].tolist()
    FP=zeros((1, eachStageDecisions.shape[1]-1))[0].tolist()
    TN=zeros((1, eachStageDecisions.shape[1]-1))[0].tolist()

    for loopVar2 in range(eachStageDecisions.shape[1]-1): # For every ...
        Stage
        subNumberofPos=((eachStageDecisions[0:NUMBER_OF_POS_IMAGES, ...
            eachStageDecisions.shape[1]-1]).tolist()).count(1)
        subNumberofNeg=((eachStageDecisions[NUMBER_OF_POS_IMAGES:eachStageDecisions.shape[0],
            eachStageDecisions.shape[1]-1]).tolist()).count(1)
        print 'Images Passed to Next Stage:', subNumberofPos, ...
            subNumberofNeg
        for loopVar1 in range(eachStageDecisions.shape[0]): # For ...
            every Image
            if eachStageDecisions[loopVar1, ...
                eachStageDecisions.shape[1]-1]==1:
                if eachStageDecisions[loopVar1, loopVar2]==1:
                    if loopVar1<NUMBER_OF_POS_IMAGES:
                        TP[loopVar2]+=1
                    else:
                        FP[loopVar2]+=1
                else:
                    if loopVar1<NUMBER_OF_POS_IMAGES:
                        FN[loopVar2]+=1
```

```
                    else:
                        TN[loopVar2]+=1
                    eachStageDecisions[loopVar1, ...
                        eachStageDecisions.shape[1]-1]=0
        if subNumberofPos!=0:
            if loopVar2==0:
                TP[loopVar2]=TP[loopVar2]/float(subNumberofPos)
                FN[loopVar2]=FN[loopVar2]/float(NUMBER_OF_POS_IMAGES)
            else:
                TP[loopVar2]=TP[loopVar2-1]*(TP[loopVar2]/float(subNumberofPos))
                FN[loopVar2]=((FN[loopVar2-1]*NUMBER_OF_POS_IMAGES)+FN[loopVar2])/float(NUMBER
        else:
            TP[loopVar2]=TP[loopVar2-1]
            FN[loopVar2]=FN[loopVar2-1]

        if subNumberofNeg!=0:
            if loopVar2==0:
                FP[loopVar2]=FP[loopVar2]/float(subNumberofNeg)
                TN[loopVar2]=TN[loopVar2]/float(NUMBER_OF_NEG_IMAGES)
            else:
                FP[loopVar2]=FP[loopVar2-1]*(FP[loopVar2]/float(subNumberofNeg))
                TN[loopVar2]=((TN[loopVar2-1]*NUMBER_OF_NEG_IMAGES)+TN[loopVar2])/float(NUMBER
        else:
            FP[loopVar2]=FP[loopVar2-1]
            TN[loopVar2]=TN[loopVar2-1]

    subNumberofPos=((eachStageDecisions[0:NUMBER_OF_POS_IMAGES, ...
        eachStageDecisions.shape[1]-1]).tolist()).count(1)
    subNumberofNeg=((eachStageDecisions[NUMBER_OF_POS_IMAGES:eachStageDecisions.shape[0], ...
        eachStageDecisions.shape[1]-1]).tolist()).count(1)
    print 'Images Passed to Next Stage:', subNumberofPos, subNumberofNeg
    #------------------------------------------------------------------------------------

    # Compute overall accuracy and TP, FN, FP, TN rates
    accuracy=correctClassifications/float(NUMBER_OF_POS_IMAGES+NUMBER_OF_NEG_IMAGES)
    FinalTP=TruePositives/float(NUMBER_OF_POS_IMAGES)
    FinalFN=FalseNegatives/float(NUMBER_OF_POS_IMAGES)
    FinalFP=FalsePositives/float(NUMBER_OF_NEG_IMAGES)
    FinalTN=TrueNegatives/float(NUMBER_OF_NEG_IMAGES)

    return classifiedLabels, accuracy, FinalTP, FinalFN, FinalFP, ...
        FinalTN, [TP, FN, FP, TN]  # Return the classification results

#----------------------------------------------------------------------------------------
# Main Code starts

global haarFeatures, sampleWeights      # Declare global HAAR feature ...
    matrix and weights array
haarFeatures=zeros((NUMBER_OF_TRAIN_POS_IMAGES+NUMBER_OF_TRAIN_NEG_IMAGES, ...
    166001), dtype=float32)
sampleWeights=zeros((NUMBER_OF_TRAIN_POS_IMAGES+NUMBER_OF_TRAIN_NEG_IMAGES, ...
    1))#, dtype=float64)
print shape(haarFeatures), shape(sampleWeights), ...
    shape(sampleWeights[:, :])
```

```
readImageHaar(folder+'train/', NUMBER_OF_TRAIN_POS_IMAGES, ...
    NUMBER_OF_TRAIN_NEG_IMAGES, 0) # Read the training images and ...
    compute HAAR features
finalCascadeClassifier=learnCascadeClassifier()#'''     # Learnt the ...
    cascade classifier
print '****************** Done Learning Cascade Classifier ...
    **************************'

# Classify the test data
classifiedLabels, accuracy, TP, FN, FP, TN, stageLevelScores = ...
    classifyTestImages(finalCascadeClassifier, folder+'test/', ...
    NUMBER_OF_TEST_POS_IMAGES, NUMBER_OF_TEST_NEG_IMAGES, ...
    TEST_POS_IMG_NUMBERING_OFFSET, TEST_NEG_IMG_NUMBERING_OFFSET)

# Print the test results
print '********************** Test Results ...
    ******************************'
print 'Accuracy=', accuracy
print 'TP=', TP
print 'FN=', FN
print 'FP=', FP
print 'TN=', TN
print 'stageLevelScores=', array(stageLevelScores)
print '********************** All Done *******************************'
```