ECE 661: Computer Vision

Homework 10, Fall 2018

Arindam Bhanja Chowdhury
abhanjac@purdue.edu

1 Overview

This homework consists of two parts. Face recognition with PCA and LDA for dimensionality reduction and the nearest-neighborhood rule for classification and Object detection with a cascaded AdaBoost classifier.

2 Principle Component Analysis (PCA)

PCA is a very widely used dimensionality reduction technique. There are 630 images of faces in the training set and 630 more in the testing set. These faces belong to 30 different people and so there are 30 different classes. There are 21 images for each class. For performing PCA, first the images are converted into gray scale and then from the size of 128x128, they are vectorized into 16384x1 size. First these vectorized images v_i are normalized as follows:

$$x_i = \frac{v_i}{||v_i||}$$

The global mean of these N (N = 630) vectorized images x_i is calculated as follows:

$$m = \frac{1}{N}\sum_{i=1}^{i=N} x_i$$

And the matrix X is formed as follows:

$$X = [\ x_1 - m \mid x_2 - m \mid x_3 - m \mid ... \mid x_{N-1} - m \mid x_N - m\ ]_{16384\times N}$$

Instead of directly calculating the eigen vectors t_i of the covariance matrix $C = \frac{1}{N}XX^T$, the eigen vectors u_i of the matrix X^TX is calculated. And then the eigen vectors t_i are obtained from u_i as follows:

$$t_i = Xu_i$$

Then the t_i's are normalized as follows:

$$w_i = \frac{t_i}{||t_i||}$$

These eigen vectors are now arranged in descending order of their corresponding eigen values. Now, the eigen vectors w_1 to w_p corresponding to the p largest eigen values of the XX^T matrix are considered to create the W matrix as follows:

$$W_p = [\ w_1 \mid w_2 \mid w_3 \mid ... \mid w_{p-1} \mid w_p\ ]_{16384\times p}$$

Then all the training and the testing images are projected onto this lower p dimensional subspace as follows to create the corresponding feature vector for that image y_i:

$$y_i = W_p^T(x_i - m)$$

Then the projected vector of the test image is classified using Nearest Neighbor classifier to predict which class it belongs to.

3 Linear Discriminant Analysis (LDA)

The objective of LDA is to find the eigen vectors w_j that maximizes the Fischer Discriminant Function

$$J(w_i) = \frac{w_j^T S_B w_j}{w_j^T S_W w_j}$$

Where S_B and S_W are the between-class scatter and the within-class scatter. However in most cases S_W is singular, and hence the procedure of Yu and Yang's algorithm is followed. First the images are converted into gray scale and then from the size of 128x128, they are vectorized into 16384x1 size. First these vectorized images v_i are normalized as follows:

$$x_i = \frac{v_i}{||v_i||}$$

The global mean of these N (N = 630) vectorized images x_i is calculated as follows:

$$m = \frac{1}{N}\sum_{i=1}^{i=N} x_i$$

Then the class means of the individual classes (here there are 30 classes each for the face of a different person) are calculated as follows:

$$m_k = \frac{1}{||C_k||}\sum_{i=1}^{i=||C_k||} x_i$$

Where $||C_k||$ is the number of training images in the kth class C_k. k = 1 to C. Then the mean matrix is formed as follows:

$$M = [\ m_1 - m \mid m_2 - m \mid m_3 - m \mid ... \mid m_{C-1} - m \mid m_C - m\ ]_{16384 \times C}$$

Instead of directly calculating the eigen vectors t_i of the matrix $S_B = \frac{1}{N}MM^T$, the eigen vectors u_i of the matrix M^TM is calculated. And then the eigen vectors t_i are obtained from u_i as follows:

$$t_i = Xu_i$$

Then the t_i's are normalized as follows:

$$V_i = \frac{t_i}{||t_i||}$$

Now form the matrix Y:

$$Y = [\ V_1 \mid V_2 \mid V_3 \mid ... \mid V_{C-1} \mid V_C\ ]$$

and calculate the D_B which are the eigen values of the matrix S_B. Now find the Z vectors as follows:

$$Z = Y D_B^{-\frac{1}{2}}$$

Now compute the eigen vectors of $Z^T S_W Z = (Z^T X)(Z^T X)^T$ where X is given by the following:

$$X = [\ x_1 1 - m_1 \mid x_1 2 - m_1 \mid x_1 3 - m1 \mid ... \mid x_1 k - m1 \mid ... \mid x_{C1} - m_C \mid ... \mid x_{Ck} - m_C\ ]$$

Organize the eigen vectors U of $Z^T S_W Z$ in ascending order and select the vectors corresponding to the smallest p eigen values from U. U have to be normalized as well. Now these new set of U_p will be used to create the projection vectors:

$$W_p = ZU_p$$

The W_p are also normalized and then the projections are done using the equation:

$$y_i = W_p^T (x_i - m)$$

Then the projected vector of the test image is classified using Nearest Neighbor classifier to predict which class it belongs to.

4 Procedure

- The training images are vectorized and the eigen vectors of the covariance matrix is computed.
- Then the PCA and LDA are used to create a lower dimensional representation for the faces in the training set. These dimensions are varied from p=1 to p=15 to see how the detection accuracy changes with the variation in p.
- All the training images are projected into the p dimensional subspace.
- The test images are also projected into the p dimensional subspace and then it is classified using the training images by nearest neighbor method. Only 1 nearest neighbor is used for this classification.
- Classification accuracy is calculated for the PCA and LDA and then the plots of the variation of the accuracy with the changes in the number of dimensions p is shown on a diagram.

5 Results: PCA and LDA

The results of PCA and LDA shows that the LDA achieves 100% accuracy earlier than PCA at a dimension of p = 5 whereas PCA takes p = 13 dimensions for reaching a 100%.

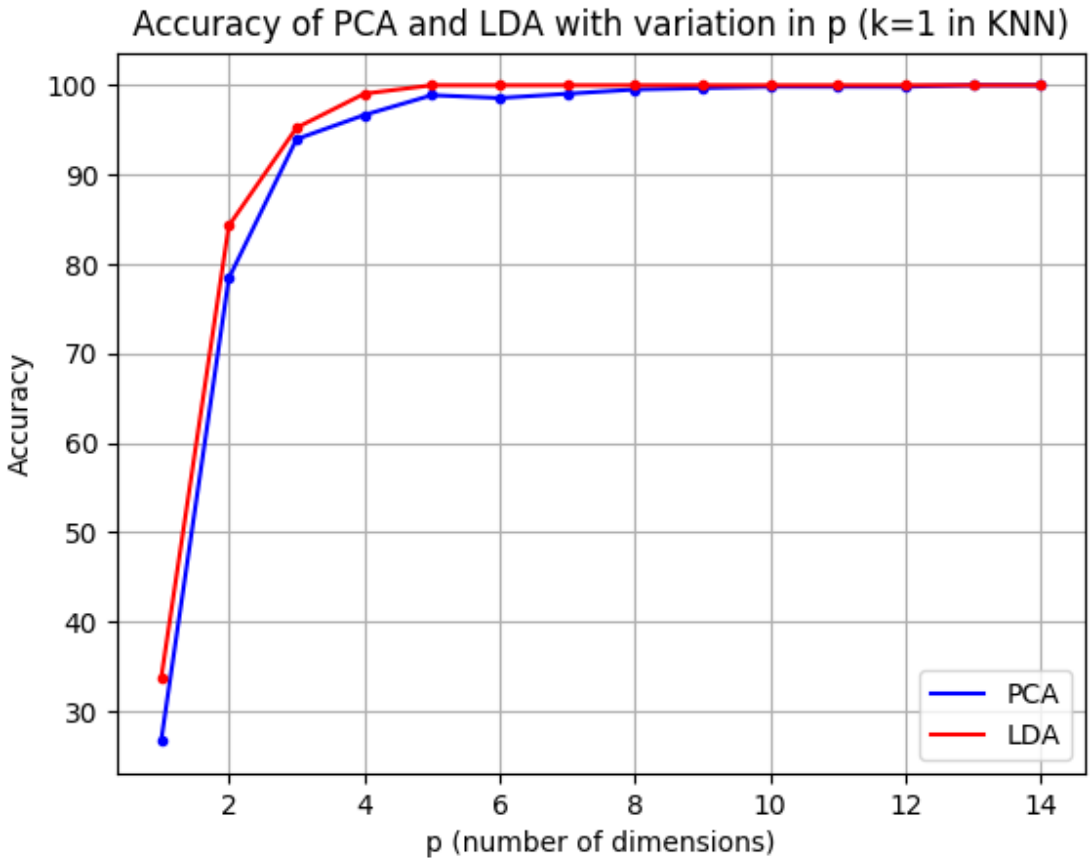

Figure 1: PCA and LDA accuracy plot.

6 Object Detection with Cascaded Classifier using AdaBoost Algorithm

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

The data given in this homework consists of images of cars and other non-car images. So there is only one class to detect. There are 2468 images for training. 1758 negative examples and 710 positive examples. There are 618 testing images. 440 negative examples and 178 positive examples. Each image is ofthe size of 20×40 (H×W)

6.1 HAAR Features for AdaBoost Classifier

In AdaBoost algorithm, the features are generated by using the weak classifiers. These weak classifiers are simply built by the thresholding of feature. In this homework, we generate the Haar-like edge detectors which detects edge features. They have the following form:

Figure 2: HAAR like feature detectors used to extract features from the image. The white part indicates the 0 region and the black part indicates the 1 region

In mathematical representation, the horizontal filter and vertical filter is denoted as $[0, 1]$ and $[1, 0].T$, respectively. To reduce computation burden, similar horizontal filters of size 1×2, 1×4, ..., 1×40 are used for sliding over the whole image to generate features. Analogously similar vertical filters of the size 2×2, 4×2, ..., 20×2 are slided over the whole image as well. The output of each of this filters at each location of an image will give rise to a feature for that image. So there is a total of 11,900 features that can be extracted from each 20×40 image. The feature calculation utilizes the integral image, which reduces computation efforts as well.

While computing the sum of the region inside the rectangle, the top left, top right, bottom left and bottom right vertices of the rectangle has to be provided. In the code that we have presented here, the function that calculates this sum needs the top left and top right vertices that are just outside the region of the rectangle and the bottom left and bottom right vertices should be inside the rectangle itself. Now for getting the area of the rectangles whose top left and top right vertices are outside the boundary of the image (which may happen for the 1st row and 1st column of the image), the area is considered to be 0.

6.2 AdaBoost - Find Best Weak Classifier and Combine into a Strong Classifier

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

- First the Haar features are extracted from all the images in the training set and the test set. These are then stacked together and saved as separate files. The shape of the training set array is 11900×2468 (number of features x number of examples) and that of the testing set array is 11900×618. The labels for positive examples are 1 and that of negative examples are 0. They are also saved in arrays of size 2468 and 618.
- Now, for the training phase we only work with the training set feature array and label array. The objective is to create a cascade of strong classifiers each of which is a combination of several weak classifiers. We specify the maximum number of strong classifiers to be created by Adaboost as S and the maximum number of weak classifiers to be combined to create a cascade as T.
- Now, there are 11900 features available to us from each image. We can take any one of these feature and apply a simple threshold to its value. Say we selected the feature f and a threshold value as θ. So we can now use this feature to create a simple classifier h_1 such which classifies an image x as positive if its f value is $>= \theta$ or as negative if its f value is $< \theta$. This kind of classifier $h_1(x, f, \theta)$ is called a weak classifier.

- Now we want these weak classifiers to be at least good enough to classify 50% of the data correctly or have at least have the power to classify 50% of the data correctly. Should it happen that the decision threshold θ is giving an error rate of more than 50%, then we can always flip the logic of the classifier. That is modify h_1 into h_2 such that h_2 classifies an image x as negative if its f value is $>= \theta$ or as positive if its f value is $< \theta$.

- We will represent both h_1 and h_2 using the same name h, but they will have another parameter called polarity (p) such that if p = 1, then the logic of h_1 will be followed (image x as positive if its f value is $>= \theta$ and negative otherwise) and if p = -1 then the logic of h_2 will be followed (image x as negative if f value is $>= \theta$ and positive otherwise). So the final h will be represented as $h(x, f, \theta, p)$.

- Mathematically this can be represented as the follows: The t^{th} weak classifier is defined as:

$$\begin{aligned} h(x, f, p, \theta) &= 0 \quad (f(x) < \theta, \quad p = 1) \\ h(x, f, p, \theta) &= 1 \quad (f(x) >= \theta, \quad p = 1) \\ h(x, f, p, \theta) &= 1 \quad (f(x) < \theta, \quad p = -1) \\ h(x, f, p, \theta) &= 0 \quad (f(x) >= \theta, \quad p = -1) \end{aligned}$$

- Now if we take a feature f out of the 11900 features, then the possible value of this feature is obtained from what is observed in the training set. That is it can have altogether 2468 possible values (each value obtained from one of the training sample image). Now each of these values can be a potential threshold. But the best threshold will be the one for which the error is the lowest, i.e. the number of misclassifications in the lowest.

- Now instead of considering the number of misclassified samples as a metric to judge the quality of a threshold, all the values (all the 2468 values) of this feature are multiplied with some weights. And the sum of the weights of these misclassified samples is considered as the metric to judge the quality of a threshold.

- If the number of positive training example is M and the number of negative training example is L, then the initial weight for each positive training image is made $\frac{1}{2M}$ and that of each negative training image is made $\frac{1}{2L}$. This is so that the total sum of all the total sum of positive weights is 0.5 and total sum of negative weights is also 0.5. And thereby the total weights of all the examples taken together is 1.

- Now we have to select a bunch of weak classifiers h, to create a strong classifier, and the maximum number of weak classifier to create a cascade is already specified as T. So we start a loop with a loop variable t such that t will run from 1 to T. At each iteration we will do the following:

  - Normalize the current weights using the formula $w_{t,i} = \frac{w_{t,i}}{\sum_i w_{t,i}}$. The $w_{t,i}$ is basically the weight corresponding to the i^{th} training sample in the t^{th} iteration.
  - Take a feature f from the set of 11900 features and choose a value of f as a threshold θ. And for each θ do the following:
    * Use this θ and compare it with the f values of all the 2468 training examples (or x's). This is equivalent to applying a simple weak classifier $h(x, f, \theta, p)$.

  - This comparison will yeild a true and a false (or a positive and a negative) label for each of the training samples (x's). This will serve as the classification result of the weak classifier $h(x, f, \theta, p)$.
  - Now we have two sets of classification results. One corresponding to the $h(x, f, \theta, p)$ with p = 1 and another $h(x, f, \theta, p)$ with p = -1.
  - In the classification result, many of the examples will be misclassified as well. Now we have already assigned a weight corresponding to each of the training example. So we take the sum of all the weights of all the misclassified training samples. This sum will be called the weighted error ϵ. This error is calculated separately for both the polarities.
  - Now this is done by considering each of the values of f as a θ, one by one.

- Now each of the f's out of the 11900 features available, are subjected to the same kind of processing, one by one. And the combination of f, it's corresponding θ and polarity p, that yeilds the minimum value of ϵ is considered as the best weak classifier found in the current iteration t. So this corresponding $h_t(x) = h(x, f_t, \theta_t, p_t)$ is the best weak classifier found in the iteration t which yeilds a minimum error ϵ_t.
- This above calculation of the minimum error can also be done much easily in the following manner: For each feature, the feature values are sorted in ascending order first. And the error for selecting the feature value of the current example as the threshold is calculated as:

$$\epsilon = \min(S^+ + (T^- - S^-),\ S^- + (T^+ - S^+)) \tag{1}$$

  Where T^+ is the total sum of weights corresponding to positive examples, T^- is the total sum of weights corresponding to negative examples, S^+ is the sum of the weights corresponding to positive examples whose value of the current feature is below the current threshold value of the feature, and S^- is the sum of the weights corresponding to the negative examples whose value of the current feature is below the current threshold value of the feature. The feature which gives us the minimum value for this error (e) is selected as the best weak classifier.
- Now a parameter β_t is calculated for each iteration t, such that $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$. And another parameter α_t is calculated such that $\alpha_t = \log(\frac{1}{\beta_t})$. This α_t is the confidence factor for this best weak classifier $h_t(x) = h(x, f_t, \theta_t, p_t)$.
- This is because if the number of misclassification is high, then the sum of the weights of the misclassified examples will also be high i.e. the weighted error ϵ_t will be high. So β_t will also be high and α_t will be low (suppose $\epsilon_t = 0.1$, then $\beta_t = 9$, and α_t will be some small value). So this implies that when the amount of misclassification is more (which indicates that the classifier h(x) is not good), the α_t is also low. So a low α indicates a bad h(x) classifier and a high α indicates a good h(x) classifier. Hence the α is basically a confidence factor for the classifier h(x).
- Now, the weights are updated using the formula $w_{t+1,i} = w_{t,i}\beta_t^{-e_i}$. $w_{t+1,i}$ will be the weight of the training sample i for the next $(t+1)^{th}$ iteration. Where $e_i = 0$ if the i^{th} sample is correctly classified and $e_i = 1$ if the i^{th} sample is misclassified.
- This formula indicates that when the classification is correct for the sample i i.e. $e_i = 0$, $w_{t+1,i} = w_{t,i}$. That is, the weight is not updated for this training sample i. If $e_i = 1$, then weight for the sample i is updated. Now if $e_i = 1$ and β_t is low (i.e. the classification was wrong for sample i, but the ϵ_t was low for h(x), which means h(x) is a

good classifier) then it is a misclassification by a good classifier and so, the weights are increased by a large amount. This is intuitively correct, because if the good classifier is doing a misclassification then this is a serious situation (since our overall targe is to have the classifiers create low misclassification rates), hence we need to increase the weights a lot to put more emphasis on this current example while selecting the next classifer. Since a low value of β will be something between 1 and 0, so a negative power of such a β is a large number. [Suppose ϵ (ϵ will always stay between 1 and 0) is low, = 0.25 (suppose). So β is 0.25/0.75 = 0.3333. So $0.3333^{-1} = 3$, which can be called a big number (as compared to the next case)].

- However, if $e_i = 1$ and β_t is high (i.e. the classification was incorrect for sample i, and the ϵ_t was high for h(x) which means h(x) is not a good classifier), the weights are increased by small amount. This is intuitively correct as well, because if the classifier is not good (which means that it may be natural for it to make mistakes, so it is not a very serious issue), the weights need not increase a lot, so that while selecting the next weak classifier in the next iteration, this sample need not be emphasized too excessively. Since a high value of β may be something much larger than 1, so a negative power of such a β is a small number. [Suppose ϵ (ϵ will always stay between 1 and 0) is low, = 0.75 (suppose). So β is 0.75/0.25 = 3. So $3^{-1} = 0.3333$, which can be called a small number (as compared to the previous case)].
- Now each of the new best weak classifier found is then multiplied with its corresponding α_t value and then combined together to create the current version of the strong classifier using the following formula.

$$\begin{aligned} &C(x) = 1, \quad \sum_t \alpha_t h_t(x) \ >= \ threshold \\ &C(x) = 0, \quad otherwise \end{aligned} \tag{2}$$

- After this the next iteration is perfomed.
- The loop will end either when t reaches its maximum value T, or if the stopping criterion is reached. The stopping criterion is determined by the targeted false-positive rate and the true detection rate for each strong classifier which is specified before the loop starts. In this homework, the targeted true detection rate during training is 1, and the targeted false-positive rate during training is 0.5.

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

- Now in this manner a set of best weak classifier is selected to create one strong classifier.
- This process is continued for at most S times to create atmost S number of cascades. After the creation of one cascade, the overall true positive rate or the detection rate and the false positive rate is calculated for the overall cascaded classifier, containing the cascades created upto this current point. If it gets below the desired false positive rate and above the desired detection rate (or true positive rate) of the overall cascaded classifier, then the cascade creation process stops and no more cascade is created. So basically there are two sets of detection rates (or true positive rates) and false positive rates involved. One is for the individual cascades stages of strong classifier and the other is for the overall cascaded classifier.

- There is one more aspect to this training and creation of the cascades. After one strong classifier stage is created, the negative training samples which are correctly classified by the cascaded classifier built till now, are removed from the set of training samples. So only the set of positive training samples and the misclassified negative training samples are used to create the next cascade stage of strong classifier.
- So it may happen that the total number of negative training samples are all depleted after creation of certain number of stages. In that case also the cascade creation process stops.

6.3 AdaBoost - Performance Evaluation

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

$$\begin{aligned} FP &= \frac{Number\ of\ misclassified\ negative\ test\ images}{total\ number\ of\ negative\ test\ images} \\ FN &= \frac{Number\ of\ misclassified\ positive\ test\ images}{total\ number\ of\ positive\ test\ images} \end{aligned} \tag{3}$$

In case of testing as well, the test samples that are classified as positive by a stage of the cascaded classifier (which will include the false positives and the true positives) are sent into the next cascade stage to be classified. Those test samples which are already predicted as negative by a certain strong cascade stage are not tested in the later cascade stages.

Parameter	**Description**	**Value**
S	Maximum number of cascades	10
T	Maximum number of weak classifiers allowed in one cascade	100

Table 1: Parameter setting for the AdaBoost

7 Result: AdaBoost Training and Testing

7.1 Results of training phase

Cascade Stage	**No. of Weak Classifiers**	**No. of Positive samples before - after**	**Detection Rate**	**No. of Negative samples before - after**	**False Positive Rate**
1	10	710 - 710	100 %	1758 - 838	47.668 %
2	17	710 - 710	100 %	838 - 366	43.675 %
3	30	710 - 710	100 %	366 - 176	48.087 %
4	36	710 - 710	100 %	176 - 76	43.182 %
5	33	710 - 710	100 %	76 - 31	40.789 %
6	15	710 - 710	100 %	31 - 3	9.677 %
7	4	710 - 710	100 %	3 - 0	0.000 %

Table 2: Details of the stages of the cascaded classifier created by AdaBoost

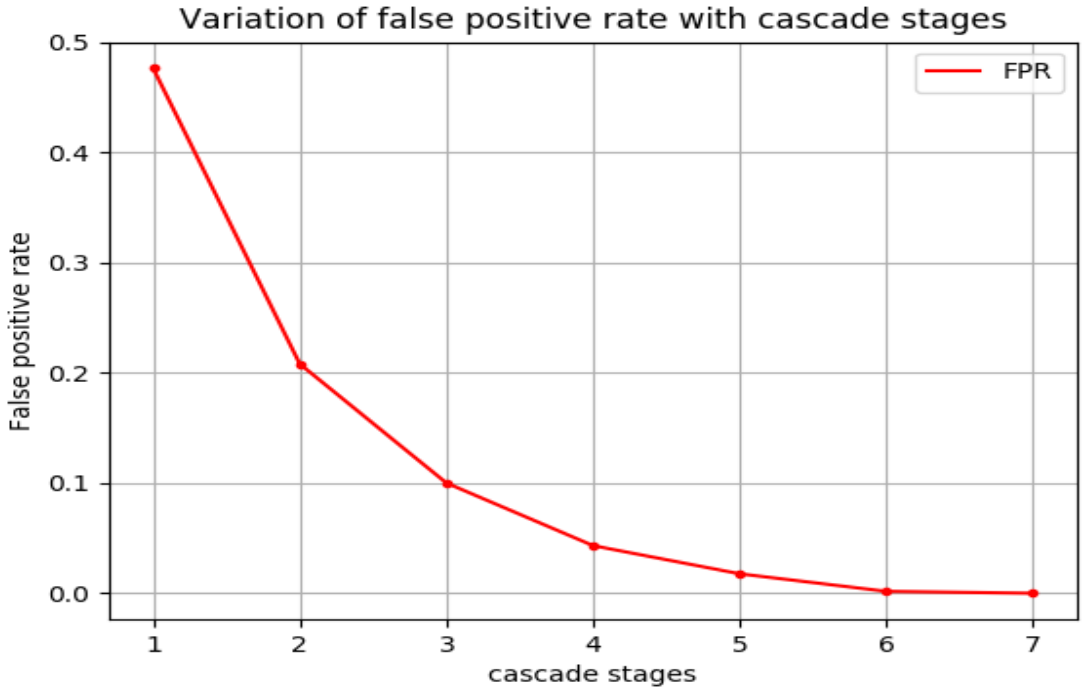

Figure 3: Plot of the False positive rate variation with the stages of the cascade during training.

7.2 Results of testing phase

Cascade Stage	**No. of Weak Classifiers**	**False Positive Rate (FP)**	**False Negative Rate (FN)**
1	10	0.089	0.152
2	17	0.018	0.202
3	30	0.014	0.236
4	36	0.005	0.281
5	33	0.002	0.320
6	15	0.002	0.365
7	4	0.002	0.365

Table 3: Details of the stages of the cascaded classifier created by AdaBoost

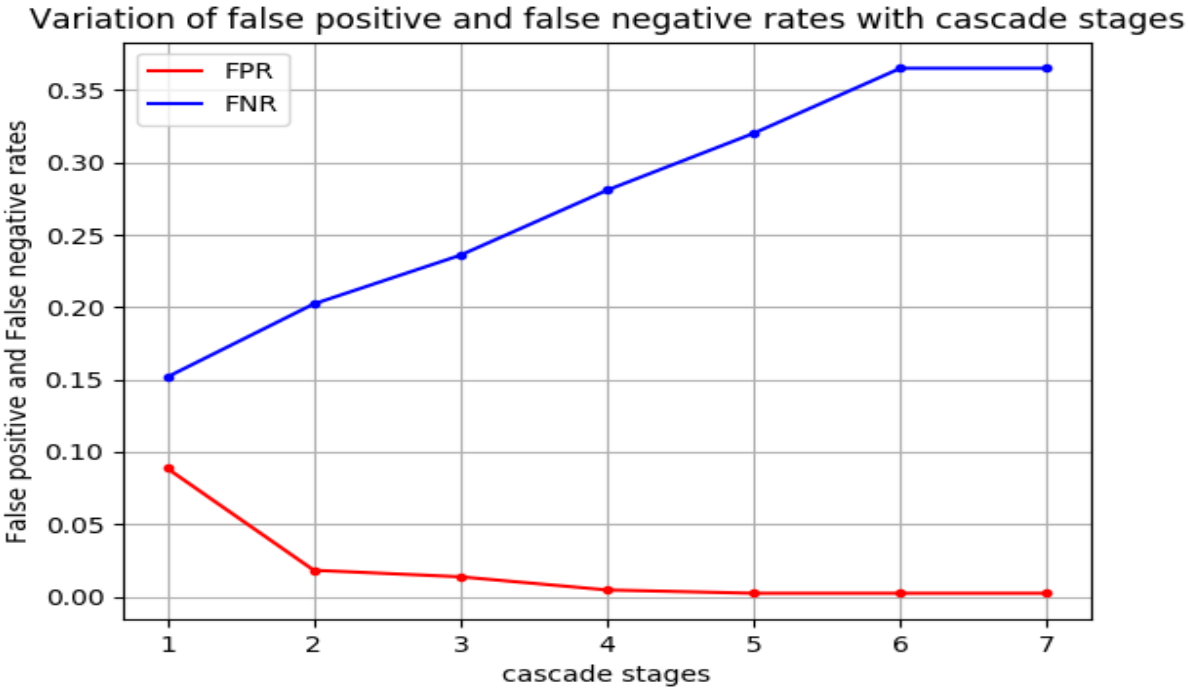

Figure 4: Plot of the False positive rate and False negative rate variations with the stages of the cascade during testing.

7.3 Observations

The classifier has been designed to achieve low false positive rates. So as the testing proceeds along the deeper cascades which are more stronger classifiers, the overall false positive rate decreases. There is an increase in the false negative rate as well, because in the attempt to reduce the number of false positives, the strong classifiers also classify some of the positive examples as negative.

The classifier can be made stronger if more number of features were used. But that will also increase the computation time and the overall training time.

```python
#!/usr/bin/env python

import numpy as np, cv2, os, time, math, copy, matplotlib.pyplot as plt
from scipy import signal, optimize
from sklearn.neighbors import KNeighborsClassifier


#================================================================================
# ARINDAM BHANJA CHOWDHURY
# abhanjac@purdue.edu
# ECE 661 FALL 2018, HW 10 Part 1.
#================================================================================

if __name__ == '__main__':

    # TASK 1.1 Face Recognition using PCA.

    # Loading the images.

    trainFilepath = './ECE661_2018_hw10_DB1/train'
    testFilepath = './ECE661_2018_hw10_DB1/test'

    listOfTrainImgs = os.listdir( trainFilepath )
    listOfTestImgs = os.listdir( testFilepath )

    img1 = cv2.imread( os.path.join( trainFilepath, listOfTrainImgs[0] ) )
    imgH, imgW, _ = img1.shape      # Shape is 128x128x3.

#--------------------------------------------------------------------------------

    # Creating the W matrix for mapping images.

    listOfImgs, filepath, dataName = listOfTrainImgs, trainFilepath, 'train_face'

    nImgs = len( listOfImgs )
    arrOfLabels = []

    # All the vectorized version of the image will be stored in this array.
    for idx, i in enumerate( listOfImgs ):
        # Convert to single channel gray image and then vectorize.
        img = cv2.imread( os.path.join( filepath, i ) )
        img = cv2.cvtColor( img, cv2.COLOR_BGR2GRAY )
        imgVec = np.expand_dims( img.flatten(), axis=1 )     # Vectorized image (16384x1).
        imgVec = imgVec / np.linalg.norm( imgVec )      # Normalizing the vector.

        arrOfVecImgs = imgVec if idx == 0 else np.hstack( (arrOfVecImgs, imgVec) )

        label = int( i[:2] )
        arrOfLabels.append( label )

        print(f'Read img {idx+1}: {i}')

    arrOfLabels = np.array( arrOfLabels )   # Converting the list into array.

    meanVec = np.mean( arrOfVecImgs, axis=1 )
    meanVec = np.expand_dims( meanVec, axis=1 )     # Mean vector is now 16384x1.
    X = arrOfVecImgs - meanVec

    xTxMat = np.matmul( X.T, X )
    L, V = np.linalg.eigh( xTxMat )      # Eigen values and vectors for X.T*X.

    # These eigen values (and corresponding vectors) are not sorted from highest
    # to lowest values. So sorting them before taking the largest eigen values.

    # The arrays have to be converted to lists before sorting.
```

```
    # The V will now be a list and each of its elements should be a sublist
    # that is the eigen vector.
    # But if we dont do the transpose, then the V[0] sublist will be formed of
    # the row elemets of V array, which we dont want. We want them to be formed
    # of the column elements of V array and hence we do the transpose before
    # converting into a list.

    L, V = L.tolist(), V.T.tolist()
    L, V = zip( *sorted( zip( L, V ), key=lambda x: x[0], reverse=True ) )
    L, V = np.array(L), np.array(V).T

    W = np.matmul( X, V )

    normW = np.linalg.norm( W, axis=0 )
    for n in range(nImgs):      W[:,n] /= normW[n]

    #print(np.matmul(W.T,W))        # Checking for orthonormality.

    filename = f'W_L_meanVec.npz'
    np.savez( filename, W, L, meanVec )       # Saving the W matrix and L.
    print( f'File {filename} saved.' )

    # Saving the X vectors for the training set.
    filename = f'X_&_labels_{dataName}.npz'
    np.savez( filename, X, arrOfLabels )      # Saving X and labels.
    print( f'File {filename} saved.' )

#-------------------------------------------------------------------------------

    # Creating the feature set for the test images.

    listOfImgs, filepath, dataName = listOfTestImgs, testFilepath, 'test_face'

    nImgs = len( listOfImgs )
    arrOfLabels = []

    for idx, i in enumerate( listOfImgs ):
        # Convert to single channel gray image and then vectorize.
        img = cv2.imread( os.path.join( filepath, i ) )
        img = cv2.cvtColor( img, cv2.COLOR_BGR2GRAY )
        imgVec = np.expand_dims( img.flatten(), axis=1 )     # Vectorized image (16384x1).
        imgVec = imgVec / np.linalg.norm( imgVec )      # Normalizing the vector.

        arrOfVecImgs = imgVec if idx == 0 else np.hstack( (arrOfVecImgs, imgVec) )

        label = int( i[:2] )
        arrOfLabels.append( label )

        print(f'Read img {idx+1}: {i}')

    arrOfLabels = np.array( arrOfLabels )   # Converting the list into array.

    X = arrOfVecImgs - meanVec

    # Saving the X vectors for the training set.
    filename = f'X_&_labels_{dataName}.npz'
    np.savez( filename, X, arrOfLabels )      # Saving X and labels.
    print( f'File {filename} saved.' )

#-------------------------------------------------------------------------------

    # Nearest Neighbor Classification for PCA method.

    # Loading train features and labels.
    dataName = 'train_face'
```

```
    npzFile = np.load( f'X_&_labels_{dataName}.npz' )
    Xtrain, labelsTrain = npzFile['arr_0'], npzFile['arr_1']

    # Loading test features and labels.
    dataName = 'test_face'
    npzFile = np.load( f'X_&_labels_{dataName}.npz' )
    Xtest, labelsTest = npzFile['arr_0'], npzFile['arr_1']

    # Loading W, L and meanVec values.
    npzFile = np.load( f'W_L_meanVec.npz' )
    W, L, meanVec = npzFile['arr_0'], npzFile['arr_1'], npzFile['arr_2']

    # Calculate the feature vectors for different p values.
    # p is the number of eigen vectors to be considered (size of the dimension).
    accuracyList, pList = [], []

    # The labelsTrain have to be a list for using it for the KNN.
    labelsTrain = labelsTrain.tolist()
    nNeighbors = 1

    for p in range( 1, 15 ):
        # Since p is the number of dimensions, so it should start from 1 and not 0.
        Wp = W[:, :p]

        # Projecting the images into the p-dimension space.
        featuresTrain = np.matmul( Wp.T, Xtrain )   # These are train features.
        featuresTest = np.matmul( Wp.T, Xtest )   # These are test features.

        # The arrays have to be converted to lists before applying kNN.
        # The Xtrain will now be a list and each of its elements should be a sublist
        # that is the image vector projected on the p-dimension subspace or in other
        # words a p-dimension feature vector corresponding to an image.

        # But if we dont do the transpose, then the Xtrain[0] sublist will be formed
        # of row elemets of Xtrain array, which we dont want. We want them to be formed
        # of the column elements of Xtrain array and hence we do the transpose before
        # converting into a list.
        featuresTrain = featuresTrain.T.tolist()

        KNN = KNeighborsClassifier( n_neighbors=nNeighbors )

        KNN.fit( featuresTrain, labelsTrain )

        featuresTest = featuresTest.T.tolist()

        predLabel = KNN.predict( featuresTest )
        #print( KNN.predict_proba( Xtest ) )

        # Now matching the predLabelArr with the labelsTest to find which of the
        # predictions match.
        predLabelArr = np.array( predLabel )
        match = labelsTest == predLabelArr      # Array of true and false.
        accuracy = np.mean( np.asarray( match, dtype=int ) ) * 100
        accuracyList.append( accuracy )
        pList.append( p )
        print( f'Accuracy with p = {p} and {nNeighbors} neighbors in KNN: {accuracy} %' )

#============================================================================

    # TASK 1.2 Face Recognition using LDA.

    # Loading the images.

    trainFilepath = './ECE661_2018_hw10_DB1/train'
    testFilepath = './ECE661_2018_hw10_DB1/test'
```

```
    listOfTrainImgs = os.listdir( trainFilepath )
    listOfTestImgs = os.listdir( testFilepath )

    img1 = cv2.imread( os.path.join( trainFilepath, listOfTrainImgs[0] ) )
    imgH, imgW, _ = img1.shape      # Shape is 128x128x3.

#-------------------------------------------------------------------------------

    # Creating the Sw matrix.

    listOfImgs, filepath, dataName = listOfTrainImgs, trainFilepath, 'train_face'

    nImgs = len( listOfImgs )
    nClasses = 30
    nImgsPerClass = 21

    dictOfClassImgs = {}

    for idx, i in enumerate( listOfImgs ):
        # Convert to single channel gray image and then vectorize.
        img = cv2.imread( os.path.join( filepath, i ) )
        img = cv2.cvtColor( img, cv2.COLOR_BGR2GRAY )
        imgVec = np.expand_dims( img.flatten(), axis=1 )     # Vectorized image (16384x1).
        imgVec = imgVec / np.linalg.norm( imgVec )      # Normalizing the vector.

        label = int( i[:2] )

        # If there is already images of this class saved, then stack this current
        # image with those. Else create a new entry for this class.
        if label in dictOfClassImgs:
            dictOfClassImgs[ label ] = np.hstack( ( dictOfClassImgs[ label ], imgVec ) )
        else:
            dictOfClassImgs[ label ] = imgVec

    # Calculating the mean images.
    dictOfMeanImgs = { k: np.expand_dims( np.mean( v, axis=1 ), axis=1 ) \
                                        for k, v in dictOfClassImgs.items() }

#-------------------------------------------------------------------------------

    # Now taking out all the elements from the dictionary and stacking them together.
    for idx in range( nClasses ):
        label = idx + 1     # Since label is from 1 to 30 and idx is from 0 to 29.

        # Also subtracting the meanImgs from the stack of images of each class.
        dictOfClassImgs[ label ] -= dictOfMeanImgs[ label ]

        arrOfVecImgs = dictOfClassImgs[ label ] if idx == 0 \
                else np.hstack( ( arrOfVecImgs, dictOfClassImgs[ label ] ) )

        arrOfMeanImgs = dictOfMeanImgs[ label ] if idx == 0 \
                else np.hstack( ( arrOfMeanImgs, dictOfMeanImgs[ label ] ) )

    globalMeanImg = np.mean( arrOfMeanImgs, axis=1 )
    globalMeanImg = np.expand_dims( globalMeanImg, axis=1 )

    #print( arrOfMeanImgs.shape, globalMeanImg.shape, arrOfVecImgs.shape )

#-------------------------------------------------------------------------------

    M = arrOfMeanImgs - globalMeanImg

    mTmMat = np.matmul( M.T, M )
    L, V = np.linalg.eigh( mTmMat )      # Eigen values and vectors for M.T*M.
```

```
    # These eigen values (and corresponding vectors) are not sorted from highest
    # to lowest values. So sorting them before taking the largest eigen values.

    # The arrays have to be converted to lists before sorting.
    # The V will now be a list and each of its elements should be a sublist
    # that is the eigen vector.
    # But if we dont do the transpose, then the V[0] sublist will be formed of
    # the row elemets of V array, which we dont want. We want them to be formed
    # of the column elements of V array and hence we do the transpose before
    # converting into a list.

    L, V = L.tolist(), V.T.tolist()
    L, V = zip( *sorted( zip( L, V ), key=lambda x: x[0], reverse=True ) )
    L, V = np.array(L), np.array(V).T

    Vb = np.matmul( M, V )

    normVb = np.linalg.norm( Vb, axis=0 )
    for n in range(nClasses):      Vb[:,n] /= normVb[n]

    # There will be nClasses no. of eigen values. But the total no. of independent
    # eigen vectors is only nClasses - 1 theoritically. Hence there will be one
    # eigen value that will be very close to 0. So when arranged in descending
    # order, the last eigen value is the one that is almost 0. So only the first
    # non-negligible ones are retained. The corresponding eigen vector is also
    # ignored.
    Db, Y = np.diag( L[:-1] ), Vb[:, :-1]

    Db1 = np.linalg.inv( np.sqrt( Db ) )    # This inverse will exist as the near
    # zero eigen values of Db are already ignored.

    Z = np.matmul( Y, Db1 )

#-------------------------------------------------------------------------------

    # Z.T*Sw*Z = Z.T*(X.T*X)*Z = (Z.T*X)*(Z.T*X).T. Where X is the zero mean
    # array of all image vectors.
    X = arrOfVecImgs

    ZTX = np.matmul( Z.T, X )       # This is 29x630 in shape. Small enough. So
    # finding its eigen vectors and vectors of ZTX*ZTX.T directly, as that will
    # be of shape 29x29 (not using the same trick as done earlier in finding the
    # eigen values of X*X.T using the X.T*X instead).

    ztxztxTmat = np.matmul( ZTX, ZTX.T )

    # Eigen values of a real symmetric matrix should be real. But due to some
    # internal calcluations np.linalg.eig was giving complex eigen values for
    # ztxTztxMat even though it is a symmetric matrix. Hence the function
    # np.linalg.eigh is used which is specifically designed to find the eigen
    # values of symmetric matrix, and it is giving proper real eigen values.
    L, U = np.linalg.eigh( ztxztxTmat )

    # These eigen values (and corresponding vectors) are not sorted from LOWEST
    # to HIGHEST values. So sorting them before taking the SMALLEST eigen values.

    # The arrays have to be converted to lists before sorting.
    # The U will now be a list and each of its elements should be a sublist
    # that is the eigen vector.
    # But if we dont do the transpose, then the U[0] sublist will be formed of
    # the row elemets of U array, which we dont want. We want them to be formed
    # of the column elements of U array and hence we do the transpose before
    # converting into a list.
```

```
    L, U = L.tolist(), U.T.tolist()
    L, U = zip( *sorted( zip( L, U ), key=lambda x: x[0] ) )
    L, U = np.array(L), np.array(U).T

    normU = np.linalg.norm( U, axis=0 )
    for n in range(U.shape[1]):      U[:,n] /= normU[n]

    #print(Z.shape, U.shape)

    W = np.matmul( Z, U )

    normW = np.linalg.norm( W, axis=0 )
    for n in range(W.shape[1]):      W[:,n] /= normW[n]

    ##print(np.matmul(W.T,W))        # Checking for orthonormality.

    filename = f'W_LDA.npz'
    np.savez( filename, W )       # Saving the W matrix and L.
    print( f'File {filename} saved.' )

#-------------------------------------------------------------------------------

    # The SAME train and test image vectors will be used for LDA as used for PCA.

#-------------------------------------------------------------------------------

    # Nearest Neighbor Classification for LDA method.

    # Loading train features and labels.
    dataName = 'train_face'
    npzFile = np.load( f'X_&_labels_{dataName}.npz' )
    Xtrain, labelsTrain = npzFile['arr_0'], npzFile['arr_1']

    # Loading test features and labels.
    dataName = 'test_face'
    npzFile = np.load( f'X_&_labels_{dataName}.npz' )
    Xtest, labelsTest = npzFile['arr_0'], npzFile['arr_1']

    # Loading W values.
    npzFile = np.load( f'W_LDA.npz' )
    W = npzFile['arr_0']

    # Calculate the feature vectors for different p values.
    # p is the number of eigen vectors to be considered (size of the dimension).
    accuracyListLDA, pListLDA = [], []

    # The labelsTrain have to be a list for using it for the KNN.
    labelsTrain = labelsTrain.tolist()
    nNeighbors = 1

    for p in range( 1, 15 ):
        # Since p is the number of dimensions, so it should start from 1 and not 0.
        Wp = W[:, :p]

        # Projecting the images into the p-dimension space.
        featuresTrain = np.matmul( Wp.T, Xtrain )   # These are train features.
        featuresTest = np.matmul( Wp.T, Xtest )   # These are test features.

        # The arrays have to be converted to lists before applying kNN.
        # The Xtrain will now be a list and each of its elements should be a sublist
        # that is the image vector projected on the p-dimension subspace or in other
        # words a p-dimension feature vector corresponding to an image.

        # But if we dont do the transpose, then the Xtrain[0] sublist will be formed
        # of row elemets of Xtrain array, which we dont want. We want them to be formed
```

```
        # of the column elements of Xtrain array and hence we do the transpose before
        # converting into a list.
        featuresTrain = featuresTrain.T.tolist()

        KNN = KNeighborsClassifier( n_neighbors=nNeighbors )

        KNN.fit( featuresTrain, labelsTrain )

        featuresTest = featuresTest.T.tolist()

        predLabel = KNN.predict( featuresTest )
        #print( KNN.predict_proba( Xtest ) )

        # Now matching the predLabelArr with the labelsTest to find which of the
        # predictions match.
        predLabelArr = np.array( predLabel )
        match = labelsTest == predLabelArr      # Array of true and false.
        accuracy = np.mean( np.asarray( match, dtype=int ) ) * 100
        accuracyListLDA.append( accuracy )
        pListLDA.append( p )
        print( f'Accuracy with p = {p} and {nNeighbors} neighbors in KNN: {accuracy} %' )

#-------------------------------------------------------------------------------

    # Plotting the accuracy vs p values.
    fig1 = plt.figure(1)
    fig1.gca().cla()
    plt.plot( pList, accuracyList, 'b', label='PCA' )
    plt.plot( pList, accuracyList, '.b' )
    plt.plot( pListLDA, accuracyListLDA, 'r', label='LDA' )
    plt.plot( pListLDA, accuracyListLDA, '.r' )
    plt.xlabel( 'p (number of dimensions)' )
    plt.ylabel( 'Accuracy' )
    plt.grid()
    plt.legend( loc=4 )
    plt.title( 'Accuracy of PCA and LDA with variation in p (k=1 in KNN)' )
    fig1.savefig( 'plot_of_accuracy_vs_p_for_face_classification.png' )
    plt.show()

#===============================================================================
```

```
#!/usr/bin/env python

import numpy as np, cv2, os, time, math, copy, matplotlib.pyplot as plt, json
from scipy import signal, optimize
from sklearn.neighbors import KNeighborsClassifier


#================================================================================
# ARINDAM BHANJA CHOWDHURY
# abhanjac@purdue.edu
# ECE 661 FALL 2018, HW 10 Part 2.
#================================================================================

#================================================================================
# FUNCTIONS CREATED IN HW10b.
#================================================================================

def createIntegralImg( img ):
    '''
    This function takes in a gray image and creates an integral representation of
    the same image and returns it. But the input image has to be grayscale.
    '''
    imgH, imgW = img.shape

    # The values in the integral images can go beyond 255. Hence this image should
    # not be of np.uint8. Otherwise there will be value overflow.
    intImg = np.zeros( ( imgH, imgW ) )

    for y in range( 1, imgH+1 ):
        for x in range( 1, imgW+1 ):
            intImg[ y-1, x-1 ] = np.sum( img[ :y, :x ] )

    return intImg

#================================================================================

def sumOfRect( intImg, tlc, brc ):
    '''
    This function takes in an integral image and also the top left corner (tlc)
    and bottom right corner (brc) coordinates of a rectangle and returns the
    sum of the pixels inside that rectangle. The tlc and brc should be in the
    form of tuple (x,y) or list [x,y].

    A------B
    |      |
    C------D

    '''
    tlx, tly, brx, bry = tlc[0] - 1, tlc[1] - 1, brc[0], brc[1]

    # If the index is outside the array boundaries then the value of
    # that region is 0.
    D = intImg[ bry, brx ] if bry > -1 and brx > -1 else 0
    B = intImg[ tly, brx ] if tly > -1 and brx > -1 else 0
    C = intImg[ bry, tlx ] if bry > -1 and tlx > -1 else 0
    A = intImg[ tly, tlx ] if tly > -1 and tlx > -1 else 0

    s = D - C - B + A

    return s

#================================================================================

def type1HAARkernels( img ):
    '''
```

```
    This function takes in the most elementary horizontal HAAR kernel of type
    [0,1], [0,0,1,1], [0,0,0,1,1,1] etc. The image is given as input to
    find how much should be the maximum width of the kernels. It returns all
    the possible kernels in a listOfKernels.
    '''
    imgH, imgW = img.shape

    listOfKernels = []
    for w in range( 1, int(imgW/2)+1 ):
        zero, one = np.zeros( (1,w) ), np.ones( (1,w) )
        kernel = np.hstack( (zero, one) )
        listOfKernels.append( kernel )

    return listOfKernels

#================================================================================

def type2HAARkernels( img ):
    '''
    This function takes in the most elementary horizontal HAAR kernel of type
    [[1,1],[0,0]] (2x2), [[1,1],[1,1],[0,0],[0,0]] (4x2),
    [[1,1],[1,1],[1,1],[0,0],[0,0],[0,0]] (6x2) etc. The image is given as input
    to find how much should be the maximum height of the kernels. It returns all
    the possible kernels in a listOfKernels.
    '''
    imgH, imgW = img.shape

    listOfKernels = []
    for h in range( 1, int(imgH/2)+1 ):
        one, zero = np.ones( (h,2) ), np.zeros( (h,2) )
        kernel = np.vstack( (one, zero) )
        listOfKernels.append( kernel )

    return listOfKernels

#================================================================================

def computeFeatureType1( intImg, listOfType1Kernels ):
    '''
    This function takes in the integral image and listOfKernels of type 1 and
    calculates the haar features of type 1 using the integral image.
    The features are returned as an array.

    A---B---E       A----B----E     A-----B-----E
    | 0 | 1 |       | 00 | 11 |     | 000 | 111 |
    C---D---F       C----D----F     C-----D-----F

       1x2              1x4              1x6

    '''
    imgH, imgW = intImg.shape

    listOfS = []
    for k in listOfType1Kernels:
        kh, kw = k.shape

        for r in range( -1, imgH-kh ):
            for c in range( -1, imgW-kw ):
                # Corners of the 1 region and the 0 region in the kernel.
                tlx0, tly0, brx0, bry0 = c, r, c + int(kw/2), r + kh
                tlx1, tly1, brx1, bry1 = c + int(kw/2), r, c + kw, r + kh

                # If the index is outside the array boundaries then the value of
                # that region is 0.
                A = intImg[ tly0, tlx0 ] if tly0 > -1 and tlx0 > -1 else 0
```

```
                B = intImg[ tly1, tlx1 ] if tly1 > -1 and tlx1 > -1 else 0
                C = intImg[ bry0, tlx0 ] if bry0 > -1 and tlx0 > -1 else 0
                D = intImg[ bry0, brx0 ] if bry0 > -1 and brx0 > -1 else 0
                E = intImg[ tly1, brx1 ] if tly1 > -1 and brx1 > -1 else 0
                F = intImg[ bry1, brx1 ] if bry1 > -1 and brx1 > -1 else 0

                s = F - 2*D + 2*B - E + C - A

                listOfS.append( s )

    return listOfS

#================================================================================

def computeFeatureType2( intImg, listOfType2Kernels ):
    '''
    This function takes in the integral image and listOfKernels of type 2 and
    calculates the haar features of type 2 using the integral image.
    The features are returned as an array.

    A----B       A----B     A----B
    | 11 |       | 11 |     | 11 |
    C----D       | 11 |     | 11 |
    | 00 |       C----D     | 11 |
    E----F       | 00 |     C----D
                 | 00 |     | 00 |
     2x2         E----F     | 00 |
                            | 00 |
                   4x2      E----F

                             6x2

    '''
    imgH, imgW = intImg.shape

    listOfS = []
    for k in listOfType2Kernels:
        kh, kw = k.shape

        for r in range( -1, imgH-kh ):
            for c in range( -1, imgW-kw ):
                # Corners of the 1 region and the 0 region in the kernel.
                tlx1, tly1, brx1, bry1 = c, r, c + kw, r + int(kh/2)
                tlx0, tly0, brx0, bry0 = c, r + int(kh/2), c + kw, r + kh

                # If the index is outside the array boundaries then the value of
                # that region is 0.
                A = intImg[ tly1, tlx1 ] if tly1 > -1 and tlx1 > -1 else 0
                B = intImg[ tly1, brx1 ] if tly1 > -1 and brx1 > -1 else 0
                C = intImg[ tly0, tlx0 ] if tly0 > -1 and tlx0 > -1 else 0
                D = intImg[ bry1, brx1 ] if bry1 > -1 and brx1 > -1 else 0
                E = intImg[ bry0, tlx0 ] if bry0 > -1 and tlx0 > -1 else 0
                F = intImg[ bry0, brx0 ] if bry0 > -1 and brx0 > -1 else 0

                s = 2*D - 2*C - B + A - F + E

                listOfS.append( s )

    return listOfS

#================================================================================

def findBestWeakClassifier( arrOfTrainFeatures=None, arrOfTrainLabels=None, \
                            arrOfNormWeights=None, nPosEx=None ):
    '''
```

```
    This function takes in the positive and negative training features and labels
    and normalized weights and then returns the best possible weak feature among
    all the available features, that gives the lowest misclassification rate.
    It also returns the number of mismatches and the classification result for
    this best weak classifier.
    '''
    if arrOfTrainFeatures is None or arrOfTrainLabels is None or arrOfNormWeights \
       is None or nPosEx is None:
        print( '\nERROR: arrOfTrainFeatures or arrOfTrainLabels or arrOfNormWeights '\
               'or nPosEx not provided. Aborting.\n')

#-------------------------------------------------------------------------------

    nFeatures, nSamples = arrOfTrainFeatures.shape
    nNegEx = nSamples - nPosEx

    # Converting to int because afterwards predicted labels will be compared with
    # these using '==' or '!=' operations (which are not good for using on floats).
    arrOfTrainLabels = np.asarray( arrOfTrainLabels, dtype=int )

    # Initializing some variables which will be updated in the loop.
    bestWeakClassifier = []
    bestClassifResult = np.zeros( nSamples )
    bestArrOfMisMatch = np.zeros( nSamples, dtype=int )     # Mismatch array.
    bestErr = math.inf

#-------------------------------------------------------------------------------

    # Total sum of positive example weights (located at the beginning of the array).
    Tp = np.sum( arrOfNormWeights[ : nPosEx ] )
    # Total sum of negative example weights (located after positive example weights).
    Tn = np.sum( arrOfNormWeights[ nPosEx : ] )

#-------------------------------------------------------------------------------

    for i in range( nFeatures ):
    #for i in range( 1 ):
        # Scanning each feature of the set of 11900 features.
        featuresOfAll = arrOfTrainFeatures[ i ].tolist()
        sampleIdx = list( range( nSamples ) )
        trueLabels = arrOfTrainLabels.tolist()
        normWeightArr = arrOfNormWeights.tolist()

        # Number of possible values of this current feature.
        # This will be same as the number of training samples, as each of the
        # training sample provides one possible value of this feature.

        # Sorting the values of the current featuresOfAll.
        featuresOfAll, sampleIdx, trueLabels, normWeightArr = zip( *sorted( zip( \
                            featuresOfAll, sampleIdx, trueLabels, normWeightArr ), \
                                                    key=lambda x: x[0] ) )

        # Converting back to arrays.
        featuresOfAll = np.array( featuresOfAll )
        sampleIdx = np.array( sampleIdx )
        trueLabels = np.array( trueLabels )
        normWeightArr = np.array( normWeightArr )

#-------------------------------------------------------------------------------

        # Sum of positive example weights, whose value for the current feature
        # (the feature i) is below the threshold value of this feature (i.e.
        # jth element of the sorted featuresOfAll array where j = 0 to nSamples-1).
        Sp = np.cumsum( normWeightArr * trueLabels )
        Sn = np.cumsum( normWeightArr ) - Sp
```

```
        err1 = Sp + ( Tn - Sn )
        err2 = Sn + ( Tp - Sp )

        # Array containing the min value of err1 and err2 arrays.
        minErr = np.minimum( err1, err2 )

        # The minimum value of this minErr will give the best possible error rate.
        # The index of this minimum error is extracted.
        minErrIdx = np.argmin( minErr )

        if err1[ minErrIdx ] <= err2[ minErrIdx ]:
            polarity = 1

            # Classification results using current threshold.
            # For polarity 1, all the samples which have feature value below
            # the threshold are classified as 0. Rest are 1.
            classifResult = arrOfTrainFeatures[ i ] >= featuresOfAll[ minErrIdx ]
            classifResult = np.asarray( classifResult, dtype=int )

            arrOfMisMatch = np.asarray( classifResult != arrOfTrainLabels, dtype=int )
            nMisMatch = int( np.sum( arrOfMisMatch ) )

            triple = [ i, featuresOfAll[ minErrIdx ], polarity, err1[ minErrIdx ], nMisMatch ]

        else:
            polarity = -1

            # Classification results using current threshold.
            # For polarity -1, all the samples which have feature value below
            # the threshold are classified as 1. Rest are 0.
            classifResult = arrOfTrainFeatures[ i ] < featuresOfAll[ minErrIdx ]
            classifResult = np.asarray( classifResult, dtype=int )

            arrOfMisMatch = np.asarray( classifResult != arrOfTrainLabels, dtype=int )
            nMisMatch = int( np.sum( arrOfMisMatch ) )

            triple = [ i, featuresOfAll[ minErrIdx ], polarity, err2[ minErrIdx ], nMisMatch ]

#-------------------------------------------------------------------------------

        # Chosing the feature as the current best feature if its
        # misclassification rate is less than the current minimum.
        # And also updating the bestErr.
        if minErr[ minErrIdx ] < bestErr:
            bestWeakClassifier = triple
            bestClassifResult = classifResult
            bestArrOfMisMatch = arrOfMisMatch

            bestErr = minErr[ minErrIdx ]    # Updating bestErr for next iteration.

        #print( i+1, nMisMatch )

#-------------------------------------------------------------------------------

    return bestWeakClassifier, bestClassifResult, bestArrOfMisMatch

#===============================================================================

def createCascade( arrOfTrainFeatures=None, arrOfTrainLabels=None, nPosEx=None, \
                   s=None, acceptableFPRforOneCascade=None, \
                   acceptableTPRforOneCascade=None, T=None, cascadeDict=None ):
    '''
    This function takes in the training features and labels and also the values
    for the acceptable thresholds for detection rate (true positive rate) and
```

```
    false positive rate and also T (which is the maximum number of best weak
    classifiers to be used for creating one cascade) and the cascade id (s), and
    creates a cascade.
    It returns the new set of training samples to be used for the creating the
    next cascade, along with the dictionary that has the details of all the
    cascades created till now.
    '''
    if arrOfTrainFeatures is None or arrOfTrainLabels is None or nPosEx is None or \
       s is None or acceptableFPRforOneCascade is None or \
       acceptableTPRforOneCascade is None or T is None:
        print( '\nERROR: arrOfTrainFeatures or arrOfTrainLabels or nPosEx=None or ' \
               's or acceptableFPRforOneCascade or acceptableTPRforOneCascade or T '\
               'or cascadeDict not provided. Aborting.\n' )

#-------------------------------------------------------------------------------

    nFeatures, nSamples = arrOfTrainFeatures.shape
    nNegEx = nSamples - nPosEx

    # Initialize the positive and negative example weights.
    arrOfWeightsPos = np.ones( nPosEx ) / ( 2 * nPosEx )
    arrOfWeightsNeg = np.ones( nNegEx ) / ( 2 * nNegEx )

    # Creating a combined array of weights.
    arrOfNormWeights = np.hstack( ( arrOfWeightsPos, arrOfWeightsNeg ) ) / 1.0
    # Sum of all weights is 1.0. Dividing by this sum to normalize the weight array.
    # The 1/2 here is to make the total sum of all weights to be equal to 1. Sum of
    # positive examples weights will be 0.5 and negative example weights is also 0.5.

    listOfAlphas = []
    hxList = []      # List of classification results of best weak classifiers.
    bestWeakClassifList = []    # List of best weak classifiers that will form the cascade.
    listOfTpr = []
    listOfFpr = []

#-------------------------------------------------------------------------------

    for t in range( T ):

        startTime = time.time()

        # Creating normalized weights.
        arrOfNormWeights = arrOfNormWeights / np.sum( arrOfNormWeights )

        # Finding the best weak classifier.
        bestWeakClassifier, bestClassifResult, bestArrOfMisMatch = \
                findBestWeakClassifier( arrOfTrainFeatures, arrOfTrainLabels, \
                                        arrOfNormWeights, nPosEx )

        print( f'Selected best weak classifier {t+1}: Triple: {bestWeakClassifier}, ' \
               f'Time taken: {time.time() - startTime : 0.3f} sec.' )

#-------------------------------------------------------------------------------

        # Storing the bestWeakClassifier in the list.
        bestWeakClassifList.append( bestWeakClassifier )

        # Calculating the parameters.
        epsilon = bestWeakClassifier[3]

        beta = epsilon / ( 1 - epsilon + 0.000000001 )
        #print( f'beta: {beta}' )

        alpha = math.log( 1 / ( beta + 0.000000001 ) )
        # The 0.000000001 is to prevent division by 0.
```

```
        # Updating the weights for the next iteration.

        #-------------------------------------------------------------------------
        # DIFFERENCE or CORRECTION.
        #
        # This should be ( beta_t )^(- e_i ) and NOT ( beta_t )^( 1 - e_i ) as given
        # in the paper.
        # The 1 in the power of beta_t will not be there.
        # This is explained better in the report itself.
        #
        #-------------------------------------------------------------------------
        #arrOfNormWeights = arrOfNormWeights * np.power( beta, 1 - bestArrOfMisMatch )

        arrOfNormWeights = arrOfNormWeights * np.power( beta, -bestArrOfMisMatch )
        #-------------------------------------------------------------------------

#-----------------------------------------------------------------------------

        # Calculating the strong cascade classifier output.
        listOfAlphas.append( alpha )
        hxList.append( bestClassifResult )

        # We have to implement the product alpha * bestClassifResult for each of the
        # best classifier that was found out. Then all those have to be added together
        # to create a new vector of size (2468, same as the number of samples).
        # This new vector should be compared with the threshold of the sum of alphas.
        # So we create an array of alpha and another matrix of the classification
        # results of all the best weak classifiers found out till now.
        arrOfAlphas = np.array( [ listOfAlphas ] ).T    # Converting to array (T x 1).
        hxArr = np.array( hxList ).T      # Converting to array (2468 x T).

        Cxtemp = np.matmul( hxArr, arrOfAlphas )

        # Since we want the true positive rate or the detection rate to be 1, i.e.
        # all the positive examples should be correctly detected, so the threshold
        # for comparing the output (of this current version of the cascade classifier)
        # is made such that all the positive examples are classified as 1
        # which is result in a true positive rate or detection rate to be 1.
        # Now the alphas corresponding to the positive examples are present in the
        # beginning nPosEx no. of elements of the arrOfAlphas. Hence the lowest
        # value among these first nPosEx is used as a threshold.
        thresholdAlpha = np.min( Cxtemp[ : nPosEx ] )
        #thresholdAlpha = np.sum( arrOfAlphas ) * 0.5

        # Output of current version of cascade classifier with t no. of weak classifiers.
        Cx = Cxtemp >= thresholdAlpha
        Cx = np.asarray( Cx, dtype=int )

        #print(Cx.shape)
        #print(thresholdAlpha, np.argmin( Cxtemp[ : nPosEx ] ))

#-----------------------------------------------------------------------------

        # Calculate the False Positive and False Negative rates for the current
        # cascade classifier with t no. of weak classifiers.

        # No. of misclassified -ve images divided by total no. of -ve images.
        fpr = np.sum( Cx[ nPosEx : ] ) / nNegEx

        # No. of correctly classified +ve images divided by total no. of +ve images.
        tpr = np.sum( Cx[ : nPosEx ] ) / nPosEx
        # This is the same as the detection rate.

        # No. of misclassified +ve images divided by total no. of +ve images.
```

```
        fnr = 1 - tpr

        # No. of correctly classified -ve images divided by total no. of -ve images.
        tnr = 1 - fpr

        listOfTpr.append( tpr )
        listOfFpr.append( fpr )

        print( f'tpr: {tpr}, fpr: {fpr}' )

        # Break if tpr and fpr have reached their acceptable thresholds.
        if tpr >= acceptableTPRforOneCascade and fpr <= acceptableFPRforOneCascade:
            break

#-------------------------------------------------------------------------------

    # Now once the set of best weak classifiers in the cascade has made the
    # detection rate (tpr) and the fpr reach the respective acceptable threshold,
    # it can be said that the cascade is formed.
    # Now the negative examples which are rightly classified as negative by this
    # cascade will be removed from the list of samples and the rest of the samples
    # will be used to create the next cascade.

    newArrOfTrainFeatures = arrOfTrainFeatures[ :, : nPosEx ]

    for i in range( nNegEx ):
        negIdx = i + nPosEx     # Pointing to negative example index.

        if Cx[ negIdx ] > 0:
            # Only appending the misclassified negative examples to the new arrays.
            misclassifiedNegEx = arrOfTrainFeatures[ :, negIdx ]
            misclassifiedNegEx = np.expand_dims( misclassifiedNegEx, axis=1 )
            newArrOfTrainFeatures = np.hstack( ( newArrOfTrainFeatures, \
                                                 misclassifiedNegEx ) )

    # Number of negative examples correctly classified.
    # This is the same as the number by which the negative example set is reduced
    # by this current cascade.
    nRemainingNegEx = newArrOfTrainFeatures.shape[1] - nPosEx
    nNegExReduced = nNegEx - nRemainingNegEx

    # Creating the new array of training labels.
    newArrOfTrainLabels = np.ones( nPosEx + nRemainingNegEx )
    newArrOfTrainLabels[ nPosEx : ] = 0

#-------------------------------------------------------------------------------

    # Recording the details in the cascade dictionary.
    newCascadeDict = copy.deepcopy( cascadeDict )

    newCascadeDict[ s ] = { 'nWeakClassifiers': t+1, 'tpr': tpr, 'fpr': fpr, \
                            'nNegExReduced': nNegExReduced, \
                            'nRemainingNegEx': nRemainingNegEx, \
                            'listOfAlphas': listOfAlphas, \
                            'bestWeakClassifList': bestWeakClassifList
                          }

#-------------------------------------------------------------------------------

    return newCascadeDict, newArrOfTrainFeatures, newArrOfTrainLabels

#===============================================================================

def testWithCascade( arrOfTestFeatures=None, cascade=None ):
    '''
```

```
    This function takes in a set of test features and a cascade
    and with which the features will be classified and sends out the result.
    '''

    # Accessing the parameters.
    T = cascade[ 'nWeakClassifiers' ]
    tpr, fpr = cascade[ 'tpr' ], cascade[ 'fpr' ]
    nNegExReduced = cascade[ 'nNegExReduced' ]
    nRemainingNegEx = cascade[ 'nRemainingNegEx' ]
    listOfAlphas = cascade[ 'listOfAlphas' ]
    bestWeakClassifList = cascade[ 'bestWeakClassifList' ]

    #print( len(listOfAlphas) )

    nFeatures, nSamples = arrOfTestFeatures.shape

    hxList = []

#-------------------------------------------------------------------------------

    for t in range( T ):
        # Accessing the parameters of the weak classifiers.
        featureId = bestWeakClassifList[t][0]
        thresh = bestWeakClassifList[t][1]
        polarity = bestWeakClassifList[t][2]

        # Evaluating the output of current weak classifier.
        featuresOfAll = arrOfTestFeatures[ featureId ]
        if polarity == 1:
            classifResult = np.asarray( featuresOfAll >= thresh, dtype=int )
        else:
            classifResult = np.asarray( featuresOfAll < thresh, dtype=int )

        # Storing the result in a list. This combined list of all results will
        # be used to get the final output of this overall cascade.
        hxList.append( classifResult )

#-------------------------------------------------------------------------------

    # Now combining the results from all the weak classifiers to get the final
    # result of this cascade classifier.
    arrOfAlphas = np.array( [ listOfAlphas ] ).T    # Converting to array (T x 1).
    hxArr = np.array( hxList ).T      # Converting to array (618 x T).

    Cxtemp = np.matmul( hxArr, arrOfAlphas )

    thresholdAlpha = np.sum( arrOfAlphas ) * 0.5

    # Output of current version of cascade classifier with T no. of weak classifiers.
    Cx = Cxtemp >= thresholdAlpha
    Cx = np.asarray( Cx, dtype=int )

    #print( arrOfAlphas.shape, hxArr.shape, Cx.shape, thresholdAlpha )

    return Cx

#===============================================================================

if __name__ == '__main__':

    # TASK 1.1 Object detection using AdaBoost based cascaded classifier.

    # Loading the images.

    trainFilepathPos = './ECE661_2018_hw10_DB2/train/positive'
```

```
    trainFilepathNeg = './ECE661_2018_hw10_DB2/train/negative'
    testFilepathPos = './ECE661_2018_hw10_DB2/test/positive'
    testFilepathNeg = './ECE661_2018_hw10_DB2/test/negative'

    listOfTrainImgsPos = os.listdir( trainFilepathPos )
    listOfTrainImgsNeg = os.listdir( trainFilepathNeg )
    listOfTestImgsPos = os.listdir( testFilepathPos )
    listOfTestImgsNeg = os.listdir( testFilepathNeg )

##-------------------------------------------------------------------------------

    ## Creating the features for the train positive samples
    ## and then saving them to a file.

    #nImgs = len( listOfTrainImgsPos )
    #for idx, i in enumerate( listOfTrainImgsPos ):
        #img = cv2.imread( os.path.join( trainFilepathPos, i ) )
        #imgH, imgW, _ = img.shape      # Shape is 20x40x3.
        #img = cv2.cvtColor( img, cv2.COLOR_BGR2GRAY )

        #intImg = createIntegralImg( img )

        #listOfType1Kernels = type1HAARkernels( img )
        #listOfType2Kernels = type2HAARkernels( img )

        #listOfS1 = computeFeatureType1( intImg, listOfType1Kernels )
        #listOfS2 = computeFeatureType2( intImg, listOfType2Kernels )

        #arrOfS = np.array( listOfS1 + listOfS2 )
        #arrOfS = np.expand_dims( arrOfS, axis=1 )   # Size now is 11900x1.

        #arrOfTrainFeaturesPos = arrOfS if idx == 0 else \
                                    #np.hstack( ( arrOfTrainFeaturesPos, arrOfS ) )

        #print(f'Read img {idx+1}/{nImgs}: {i}')

    #arrOfTrainLabelsPos = np.ones( ( len( listOfTrainImgsPos ) ) )

    ## Saving the array in a file.
    #filename = 'train_features_pos.npz'
    #np.savez( filename, arrOfTrainFeaturesPos, arrOfTrainLabelsPos )
    #print( f'File {filename} saved.' )

##-------------------------------------------------------------------------------

    ## Creating the features for the train negative samples
    ## and then saving them to a file.
    #nImgs = len( listOfTrainImgsNeg )
    #for idx, i in enumerate( listOfTrainImgsNeg ):
        #img = cv2.imread( os.path.join( trainFilepathNeg, i ) )
        #imgH, imgW, _ = img.shape      # Shape is 20x40x3.
        #img = cv2.cvtColor( img, cv2.COLOR_BGR2GRAY )

        #intImg = createIntegralImg( img )

        #listOfType1Kernels = type1HAARkernels( img )
        #listOfType2Kernels = type2HAARkernels( img )

        #listOfS1 = computeFeatureType1( intImg, listOfType1Kernels )
        #listOfS2 = computeFeatureType2( intImg, listOfType2Kernels )

        #arrOfS = np.array( listOfS1 + listOfS2 )
        #arrOfS = np.expand_dims( arrOfS, axis=1 )   # Size now is 11900x1.

        #arrOfTrainFeaturesNeg = arrOfS if idx == 0 else \
```

```
                                #np.hstack( ( arrOfTrainFeaturesNeg, arrOfS ) )

        #print(f'Read img {idx+1}/{nImgs}: {i}')

    #arrOfTrainLabelsNeg = np.zeros( ( len( listOfTrainImgsNeg ) ) )
    ##arrOfTrainLabelsNeg = np.ones( ( len( listOfTrainImgsNeg ) ) ) * -1

    ## Saving the array in a file.
    #filename = 'train_features_neg.npz'
    #np.savez( filename, arrOfTrainFeaturesNeg, arrOfTrainLabelsNeg )
    #print( f'File {filename} saved.' )

##-------------------------------------------------------------------------

    ## Creating the features for test positive samples
    ## and then saving them to a file.
    #nImgs = len( listOfTestImgsPos )
    #for idx, i in enumerate( listOfTestImgsPos ):
        #img = cv2.imread( os.path.join( testFilepathPos, i ) )
        #imgH, imgW, _ = img.shape      # Shape is 20x40x3.
        #img = cv2.cvtColor( img, cv2.COLOR_BGR2GRAY )

        #intImg = createIntegralImg( img )

        #listOfType1Kernels = type1HAARkernels( img )
        #listOfType2Kernels = type2HAARkernels( img )

        #listOfS1 = computeFeatureType1( intImg, listOfType1Kernels )
        #listOfS2 = computeFeatureType2( intImg, listOfType2Kernels )

        #arrOfS = np.array( listOfS1 + listOfS2 )
        #arrOfS = np.expand_dims( arrOfS, axis=1 )   # Size now is 11900x1.

        #arrOfTestFeaturesPos = arrOfS if idx == 0 else \
                                #np.hstack( ( arrOfTestFeaturesPos, arrOfS ) )

        #print(f'Read img {idx+1}/{nImgs}: {i}')

    #arrOfTestLabelsPos = np.ones( ( len( listOfTestImgsPos ) ) )

    ## Saving the array in a file.
    #filename = 'test_features_pos.npz'
    #np.savez( filename, arrOfTestFeaturesPos, arrOfTestLabelsPos )
    #print( f'File {filename} saved.' )

##-------------------------------------------------------------------------

    ## Creating the features for the test negative samples
    ## and then saving them to a file.
    #nImgs = len( listOfTestImgsNeg )
    #for idx, i in enumerate( listOfTestImgsNeg ):
        #img = cv2.imread( os.path.join( testFilepathNeg, i ) )
        #imgH, imgW, _ = img.shape      # Shape is 20x40x3.
        #img = cv2.cvtColor( img, cv2.COLOR_BGR2GRAY )

        #intImg = createIntegralImg( img )

        #listOfType1Kernels = type1HAARkernels( img )
        #listOfType2Kernels = type2HAARkernels( img )

        #listOfS1 = computeFeatureType1( intImg, listOfType1Kernels )
        #listOfS2 = computeFeatureType2( intImg, listOfType2Kernels )

        #arrOfS = np.array( listOfS1 + listOfS2 )
        #arrOfS = np.expand_dims( arrOfS, axis=1 )   # Size now is 11900x1.
```

```
        #arrOfTestFeaturesNeg = arrOfS if idx == 0 else \
                                    #np.hstack( ( arrOfTestFeaturesNeg, arrOfS ) )

        #print(f'Read img {idx+1}/{nImgs}: {i}')

    #arrOfTestLabelsNeg = np.zeros( ( len( listOfTestImgsNeg ) ) )
    ##arrOfTestLabelsNeg = np.ones( ( len( listOfTestImgsNeg ) ) ) * -1

    ## Saving the array in a file.
    #filename = 'test_features_neg.npz'
    #np.savez( filename, arrOfTestFeaturesNeg, arrOfTestLabelsNeg )
    #print( f'File {filename} saved.' )

##=============================================================================

    # TRAINING THE ADABOOST CLASSIFIER.

    # Loading the train and test positive and negative features and labels.
    filename = 'train_features_pos.npz'
    npzFile = np.load( filename )
    arrOfTrainFeaturesPos, arrOfTrainLabelsPos = npzFile['arr_0'], npzFile['arr_1']
    #print( arrOfTrainFeaturesPos.shape, arrOfTrainLabelsPos.shape )

    filename = 'train_features_neg.npz'
    npzFile = np.load( filename )
    arrOfTrainFeaturesNeg, arrOfTrainLabelsNeg = npzFile['arr_0'], npzFile['arr_1']
    #print( arrOfTrainFeaturesNeg.shape, arrOfTrainLabelsNeg.shape )

    nPosEx, nNegEx = arrOfTrainLabelsPos.shape[0], arrOfTrainLabelsNeg.shape[0]
    nFeatures, nSamples = arrOfTrainFeaturesPos.shape[0], nPosEx + nNegEx

#------------------------------------------------------------------------------

    arrOfTrainFeatures = np.hstack( ( arrOfTrainFeaturesPos, arrOfTrainFeaturesNeg ) )
    arrOfTrainLabels = np.hstack( ( arrOfTrainLabelsPos, arrOfTrainLabelsNeg ) )

    T = 100   # Max number of weak classifiers to be used for creating 1 strong cascade.
    S = 10   # Max number of strong cascades to be built.
    targetFPR = 0.000001    # Target false positive rate.
    targetTPR = 1   # Target true positive rate (detection rate).
    FPR = 1
    TPR = 0
    FPRlist, TPRlist, cascadeStageIdx = [], [], []

    # This is the threshold for accepting the false +ve rate for a cascade.
    acceptableFPRforOneCascade = 0.5
    # This is the threshold for accepting the true +ve rate or the detection rate
    # for a cascade.
    acceptableTPRforOneCascade = 1

    # Dictionary to hold the details of the cascade.
    cascadeDict = {}

    loopStartTime = time.time()

    for s in range( 1, S+1 ):   # Cascades are named as 1,2,3... (not as 0,1,2..).
        # Creating one cascade.
        newCascadeDict, newArrOfTrainFeatures, newArrOfTrainLabels = \
                createCascade( arrOfTrainFeatures, arrOfTrainLabels, nPosEx, s, \
                   acceptableFPRforOneCascade, acceptableTPRforOneCascade, T, \
                       cascadeDict )

#------------------------------------------------------------------------------
```

```
        cascadeDict = copy.deepcopy( newCascadeDict )   # Updating the cascadeDict.

        # Updating the feature and label array for the next iteration.
        # The features of negative examples which are rightly classified as negative
        # by this cascade are removed from the feature array and the rest of the
        # samples are used to create the new array of features. This will be used
        # to create and train the next cascade.

        arrOfTrainFeatures = copy.deepcopy( newArrOfTrainFeatures )
        arrOfTrainLabels = copy.deepcopy( newArrOfTrainLabels )

        #print( arrOfTrainFeatures.shape, arrOfTrainLabels.shape )

        nRemainingNegEx = cascadeDict[s]['nRemainingNegEx']
        nNegExReduced = cascadeDict[s]['nNegExReduced']
        tpr, fpr = cascadeDict[s]['tpr'], cascadeDict[s]['fpr']

        print( f'\nCascade {s} created: Number of negative examples reduced from ' \
               f'{nRemainingNegEx + nNegExReduced} to {nRemainingNegEx}. Reduction by ' \
               f'{nRemainingNegEx * 100 / (nRemainingNegEx + nNegExReduced) : 0.3f} %.\n' )

        TPR *= tpr      # Updating the true positive (detection) rate.
        FPR *= fpr      # Updating the false positive rate.

        FPRlist.append( FPR )
        TPRlist.append( TPR )
        cascadeStageIdx.append( s )

        if ( TPR >= targetTPR and FPR <= targetFPR ) or nRemainingNegEx == 0:   break
        # break if the target false positive rate and true positive (detection)
        # rates are achieved or there are no more misclassified negative examples.

#-------------------------------------------------------------------------------

    # Now save the cascadeDict in a json file.
    with open( 'cascadeDict.json', 'w' ) as infoFile:
        json.dump( cascadeDict, infoFile, indent=4, separators=(',', ': ') )

    print( f'\nTotal training time: {time.time() - loopStartTime : 0.3f} sec.\n' )

#-------------------------------------------------------------------------------

    # Plotting the variation of the False positive rate with the cascade stages.
    fig1 = plt.figure(1)
    fig1.gca().cla()
    plt.plot( cascadeStageIdx, FPRlist, 'r', label='FPR' )
    plt.plot( cascadeStageIdx, FPRlist, '.r' )
    plt.grid()
    plt.legend( loc=1 )
    plt.xlabel( 'cascade stages' )
    plt.ylabel( 'False positive rate' )
    plt.title( 'Variation of false positive rate with cascade stages' )
    fig1.savefig( 'plot_of_FPR_vs_nStages_training.png' )
    plt.show()

#===============================================================================

    # TESTING THE ADABOOST CLASSIFIER.

    filename = 'test_features_pos.npz'
    npzFile = np.load( filename )
    arrOfTestFeaturesPos, arrOfTestLabelsPos = npzFile['arr_0'], npzFile['arr_1']
    #print( arrOfTestFeaturesPos.shape, arrOfTestLabelsPos.shape )

    filename = 'test_features_neg.npz'
```

```
    npzFile = np.load( filename )
    arrOfTestFeaturesNeg, arrOfTestLabelsNeg = npzFile['arr_0'], npzFile['arr_1']
    #print( arrOfTestFeaturesNeg.shape, arrOfTestLabelsNeg.shape )

    nTestPosEx, nTestNegEx = arrOfTestLabelsPos.shape[0], arrOfTestLabelsNeg.shape[0]

#-------------------------------------------------------------------------------

    # Load the classifier from the saved dictionary.
    with open( 'cascadeDict.json', 'r' ) as infoFile:
        cascadeDict = json.load( infoFile )

    nStages = len( cascadeDict )

    # Lists to store the false positive and false negative rates on the test set.
    testFPRlist, testFNRlist = [], []

    # Initializing the number of false positives and the number of false negative
    # examples nFP and nFN. These will be updated at every stage.
    nFP, nFN = 0, 0

    arrOfTestFeatures = np.hstack( ( arrOfTestFeaturesPos, arrOfTestFeaturesNeg ) )

    nPosEx, nNegEx = nTestPosEx, nTestNegEx     # Initialize nPosEx and nNegEx.

#-------------------------------------------------------------------------------

    for idx, (k, cascade) in enumerate( cascadeDict.items() ):
        # Testing and extracting the prediction result of the current cascade.
        Cx = testWithCascade( arrOfTestFeatures, cascade )

        # Remove the samples which are classified as negative by current cascade
        # to create the new array which will be tested using the next cascade.
        # Only samples classified as positive are considered for testing
        # on the next stage. Combining these samples to create a new array.
        # This will include the true positive examples and false positive examples.

        nTPex = 0
        for i in range( nPosEx ):
            if Cx[ i, 0 ] == 1:   # Correctly classified positive example.
                example = arrOfTestFeatures[ :, i ]
                example = np.expand_dims( example, axis=1 )
                newArrOfTestFeatures = np.hstack( ( newArrOfTestFeatures, example ) ) \
                                                    if i > 0 else example
                nTPex += 1

        # No. of false negatives is same as the difference of the current no. of
        # true positive samples from the initial no. of positive samples.
        nFN += ( nPosEx - nTPex )

        nFPex = 0
        for i in range( nNegEx ):
            label = nPosEx + i      # Pointing to negative example index.
            if Cx[ label, 0 ] == 1:   # Negative example classified falsely as positive.
                example = arrOfTestFeatures[ :, label ]
                example = np.expand_dims( example, axis=1 )
                newArrOfTestFeatures = np.hstack( ( newArrOfTestFeatures, example ) )

                nFPex += 1

        nFP = nFPex

#-------------------------------------------------------------------------------

        testFPRlist.append( nFP / nTestNegEx )
```

```
        testFNRlist.append( nFN / nTestPosEx )

        print( f'FPR for cascade {k} during testing: {testFPRlist[-1] : 0.3f}' )
        print( f'FNR for cascade {k} during testing: {testFNRlist[-1] : 0.3f}' )

        # Updating the counts of positive and negative examples and also the array
        # of features for the next cascade.
        nPosEx, nNegEx = nTPex, nFPex
        arrOfTestFeatures = copy.deepcopy( newArrOfTestFeatures )

        if nPosEx == 0:     break   # break if there are no more positive examples.
        # This is just for fail safe for the case when all examples are classified
        # as negetive.

#-------------------------------------------------------------------------------

    cascadeStageIdx = list( cascadeDict )

    # Plotting the variation of the False positive rate with the cascade stages.
    fig2 = plt.figure(2)
    fig2.gca().cla()
    plt.plot( cascadeStageIdx, testFPRlist, 'r', label='FPR' )
    plt.plot( cascadeStageIdx, testFPRlist, '.r' )
    plt.plot( cascadeStageIdx, testFNRlist, 'b', label='FNR' )
    plt.plot( cascadeStageIdx, testFNRlist, '.b' )
    plt.grid()
    plt.legend( loc=2 )
    plt.xlabel( 'cascade stages' )
    plt.ylabel( 'False positive and False negative rates' )
    plt.title( 'Variation of false positive and false negative rates with cascade stages' )
    fig2.savefig( 'plot_of_FPR_and_FNR_vs_nStages_testing.png' )
    plt.show()
```