ECE661: Homework 10

Rahul Deshmukh
deshmuk5@purdue.edu
PUID: 0030004932

December 6, 2018

This homework has two separate parts:

1. Face Recognition using PCA and LDA
2. Object Detection using Cascaded Adaboost Classifier

1 Part 1:Face Recognition using PCA and LDA

For this Part we have to use PCA and LDA on face images and then classify them using nearest neighbor classification. The main idea for both of these methods is to reduce the dimension of the feature space so as to have accurate classification using NN.
For both of the methods we first convert our images into vectorized format and then reduce the dimension of this feature space using a projection matrix **W** which can be constructed in different ways based on the method we are using.

1.1 Principal Component Analysis (PCA)

The Goal of PCA is to find an orthogonal set of maximal-variance directions in the underlying vector space for all available training data.
We start with constructing the Co-Variance Matrix for the training data-set. We then find the eigen vectors of the Co-variance matrix which serves as our class-discriminating feature. It should be noted given the size of image is 128x128 the co-variance matrix will be a square matrix of dimension 16384. Also, as it is formed by summation of outer products it will be rank deficient. This helps us in using a computational trick to get the eigen vectors of the matrix which will be discussed shortly.

The procedure for PCA is as follows:

Step (1): Vectorize the training image (using reshape) to a to column vector $\vec{x_i}$. Normalize the vector with its magnitude to make it illumination invariant. (Make sure to use gray-scale images)

Step (2): Calculate the global mean of the training data set using:

$$\vec{m} = \frac{1}{N}\Sigma_i \vec{x_i}$$

Step (3): Subtract the global mean from all of the samples. It is best here to store the result as a matrix of the form

$$X = \begin{bmatrix} \vec{x_1} - \vec{m} & \vec{x_2} - \vec{m} & \vec{x_1} - \vec{m} ..... & \vec{x_N} - \vec{m} \end{bmatrix}$$

Step (4): The Co-variance matrix is can now be constructed using:

$$C = \frac{1}{N} X^T X$$

but, we need not form the co-variance matrix as we are only interested in its eigen vectors ($\vec{w}$). To find the eigen vectors of $X^T X$ we use a computational trick, We find the eigen vector($\vec{u}$) of XX^T and then find $\vec{w}$ from $\vec{u}$ using:

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

and then normalize $\vec{w}$ to get unit magnitude eigen vectors.

Step (5): We then select the P numbers of largest magnitude (of eigen values of $\vec{u}$ which will be the same for $\vec{w}$) eigen vectors to construct our projection matrix W as follows:

$$W = \begin{bmatrix} \vec{w_1} & \vec{w_2} & \vec{w_3} ... & \vec{w_P} \end{bmatrix}$$

Step (6): We can now use W to project all of our training data to get its projection in the reduced dimension space as follows:

$$\vec{y_i} = W(\vec{x_i} - \vec{m})$$

Step (7): Now, for the incoming query sample (testing sample) we first need to vectorize and normalize the sample, then project to the reduced dimension space (the feature space) and then classify the query sample as the same class as of its nearest neighbor from the set of projections of the training data (from previous step).

1.2 Linear Discriminant Analysis(LDA)

The Goal of LDA is to find the directions in the underlying vector space that are maximally discriminating between the classes which is done by finding the eigen vectors $\vec{w}$ that maximize the Fisher's Discriminant function:

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

where S_B is the between-class scatter and S_W is the within-class scatter. Given by:

$$S_B = \frac{1}{|C|} \Sigma_i^{|C|} (\vec{m_i} - \vec{m})(\vec{m_i} - \vec{m})^T$$

$$S_W = \frac{1}{|C|} \Sigma_i^{|C|} \frac{1}{|C_i|} \Sigma_k^{|C_i|} (\vec{x_k^i} - \vec{m_i})(\vec{x_k^i} - \vec{m_i})^T$$

When trying to maximize $J(\vec{w})$ we can reduce the problem to a generalized eigen value decomposition problem with the help of Lagrange multipliers and the problem now becomes:

$$S_B \vec{w} = \lambda S_W \vec{w}$$

From the above equation if S_W was full-rank then we can multiply both side by its inverse and reduce the above problem to a normal eigen decomposition problem. But as it turns out from definition of S_W, is not full rank(as its a sum of outer products). We use Yu and Yang's algorithm to find $\vec{w}$.

The steps for LDA is as follows:

Step (1): Vectorize and normalize the training sample

Step (2): Find global mean using:

$$\vec{m} = \frac{1}{N} \Sigma_i \vec{x_i}$$

Step (3): Find Class means:

$$\vec{m_i} = \frac{1}{|C_i|}\Sigma_i \vec{x_i}$$

Step (4): Form the matrix M :

$$M = \begin{bmatrix} \vec{m_1} - \vec{m} & \vec{m_2} - \vec{m} & \vec{m_3} - \vec{m} ... & \vec{m_C} - \vec{m} \end{bmatrix}$$

note that $S_B = MM^T/C$

Step (5): Find eigen vectors($\vec{V}$) of S_B using the computational trick by first finding eigen vectors($\vec{u}$) of $M^T M/C$ and then convert $\vec{u}$ to $\vec{V}$ using:

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

Step (6): We then sort the eigen-vectors $\vec{V}$ in descending order based on the eigen values(μ) of $\vec{u}$ (which will be the same for $\vec{V}$).

Step (7): Form the matrix Y and D_B as follows:

$$Y = \begin{bmatrix} \vec{V_1} & \vec{V_2} & \vec{V_3} ... & \vec{V_C} \end{bmatrix}$$

$$D_B = DiagonalMatrix(\mu)$$

Step (8): Compute $Z = YD_B^{-1/2}$

Step (9): Compute the eigen-vectors of $Z^T S_W Z$ using the computational trick. We can simplify it as follows:

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

where X_W is;

$$X_W = \begin{bmatrix} x_{11} - m_1 & x_{12} - m_1 ... & x_{21} - m_2 & x_{22} - m_2 ... & x_{Ck} - m_C \end{bmatrix}$$

Step (10): Sort the eigen vectors($\vec{U}$) of $Z^T S_W Z$ in ascending order based on the eigen values (γ). Select the smallest P eigen-vectors from the set $\vec{U}$ denoted by $\hat{U}$.

Step (11): Form the projection matrix using:

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

Step (12): Normalize each eigen vector in W by its magnitude.

Step (13): We can now use W to project all of our training data to get its projection in the reduced dimension space as follows:

$$\vec{y_i} = W(\vec{x_i} - \vec{m})$$

Step (14): Now, for the incoming query sample (testing sample) we first need to vectorize and normalize the sample, then project to the reduced dimension space (the feature space) and then classify the query sample as the same class as of its nearest neighbor from the set of projections of the training data (from previous step).

1.3 Performance of PCA and LDA

To evaluate the performance of either of the methods we use the classification accuracy as our metric which is given by:

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

I have evaluated the performance for both of the methods by increasing the dimension of the subspace from 1 to 20.

1.4 Results and comparison

The plot for accuracy of classification of both methods is:

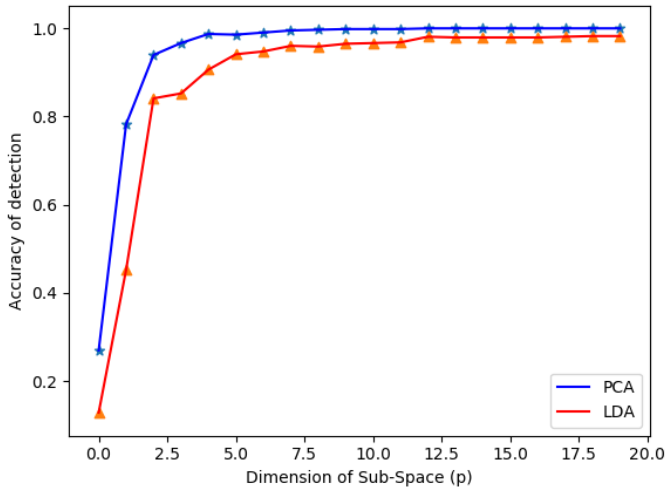

Figure 1: PCA vs LDA accuracy

Observations:

1. PCA is performing better than LDA for the current data-set.
2. As the dimension of the subspace increases, the accuracy for both the methods increases and then saturates to 1 after a certain minimum size of subspace.
3. PCA reaches accuracy of 100% for a subspace dimension of 12 whereas LDA reaches a maximum accuracy of 98% for a subspace dimension of 20.

2 Part 2: Object Detection using Cascaded Adaboost Classifier

For this part we are required to carry out cascaded stages of adaboost as discussed by the Viola and Jones. The Main idea behind cascaded adaboost classifier is to design several strong classifiers (such that each strong classifier consists of multiple weak classifier). With a goal of targeted false positive rate and true detection rate for each classifier stage, we can compound all stages to achieve a final low false-positive rate while keeping the true detection rate being acceptable.

Starting from stage-1 we allow all the positive images along with wrongly classified negative images to go for the next stage.

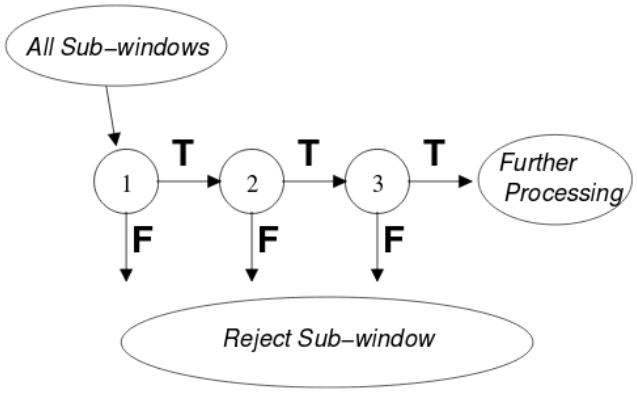

Figure 2: Cascades

The main tasks for the algorithm are:

1) Feature Generation

2) Adaboost Classifier

2.1 Feature Generation

In the Viola and Jones paper, the authors have generated a feature space using an over complete set of simple Haar-type classifiers. The classifiers are of three types:

1. Two Rectangle Type: this includes $[0, 1]$, $[0, 1]^T$
2. Three Rectangle Type: this includes $[0, 1, 0]$ and $[0, 1, 0]^T$
3. Four Rectangle Type: this includes $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$

Where the classifiers can have all possible heights and widths and all possible positions in the whole image. The values 1 and 0 in the classifier indicates the pixel values which need to be summed and subtracted respectively.
We make use of integral representation of the original image to carry out efficient evaluations of the classifiers.

NOTE: When using integral image for evaluation of Haar-filters, its easier to implement it with zero padding of the image on the left and top.

To reduce computation cost, I have selected only a subset of all possible classifiers with a total of 20461 classifiers. Also, for our case with a 20x40 image we can have a total of 312260 possible classifiers.

Generation and evaluation of classifiers can be checked at the source-code which is self-explanatory.

2.2 Adaboost Classifier

Within the Adaboost Classifier we carry out a maximum of T iterations and at each iteration we identify a weak classifier. We also keep track of the current True positive and False Positive rate. When the rates achieve a desired value we can terminate the loop.At the end of the loop we obtain a strong classifier.

The steps involved are as follows:

Step (1): Initialize the weights for the positive and negative samples as $w_{t,i} = 1/(2Npos), (1/2Nneg)$ where Npos and Nneg are the total number of positive and negative samples respectively.

Step (2): Iterate over t

Step (3): Normalize the weights

Step (4): For this step we need to identify a classifier $h(x, f, \theta, p)$ **out of all classifiers** which has minimum weighted error

As we need to find a threshold on the real-line we start by first sorting the the feature values. We then find evaluate the error pair for two possible polarities:

$$e = (S^+ + (T^- - S^-), S^- + (T^+ - S^+))$$

where, T^+ is the total sum of all positive weights, T^- is the total sum of negative weights, S^+ is the sum of positive weights below the current possible threshold and S^- is the sum of negative weights below the current possible threshold.

We evaluate the error pair for all possible values of threshold in the feature list and retain the one which gives us the minimum error.

The best weak classifier is defined as:

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

where x is the sample, f(x) is the feature value,θ is the threshold and p is the polarity determined by e such that if the minimum error is in the first type of error then the polarity is p=-1 and vice-versa.

Store the classifier which gives us the best-ever error(ϵ_t) along with its polarity and threshold.

Step (5): compute

$$\beta = \epsilon_t/(1 - \epsilon_t)$$
$$\alpha_t = log(1/\beta_t)$$

Step (6): Update the weights using:

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

where $e_i = 0$ is sample is correctly classifier vice-versa

Step (7): Measure current True positive and False Positive rate, terminate the loop over t if they meet desired value else repeat steps 2-7.

Step (8): The final Strong Classifier is:

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

For this homework I have chosen the targeted True-Positive rate as 1 and targeted False Positive rate as 0.5 for training.

The threshold for the Strong classifier is chosen such that it allows for all positive sample to pass for the next stage for the case of training data and for testing data we set the threshold as $0.5 * \Sigma_t \alpha_t$

2.3 Results and discussion

The Performance of adaboost is conducted using two metrics: the False-Positive(FP) rate and the False-Negative(FN) rate evaluated as:

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

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

Training results

Stage #	1	2	3	4	5	6	7	8
No. of classifiers	10	18	17	18	14	9	7	3
FP (for stage)	0.24	0.42	0.39	0.45	0.45	0.4	0.16	0
TP (for stage)	1	1	1	1	1	1	1	1

Testing Results

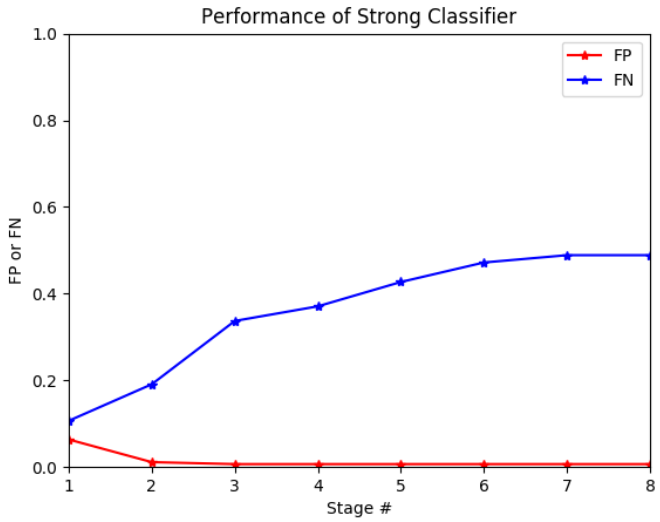

Figure 3: Performance of Strong Classifier on Testing Data-set

Observations From the above plot for the testing data we can observe that the False-Positive rate decreases as expected because that is how we have designed our cascades. However, The False-Negative rate increases, this is because the True Positive rate has decreased. As False-Negative rate is given by 1-TP, therefore as TP decreases FN increases.

3 Source Code:

3.1 PCA and LDA

3.1.1 Main File:

```
"""
ECE661: hw10 Part-1: PCA-LDA
@author: Rahul Deshmukh
email: deshmuk5@purdue.edu
PUID: 0030004932
"""
#%% ---------------------import libraries ---------------------------------------#
import cv2
import numpy as np
import sys
sys.path.append('../../')
import MyCVModule as MyCV
import os
#-------------------------------------------------------------------------------#
#define read path
readpath = '../files/faces/'
savepath = '../data/PCA/'
#-------------------------Training----------------------------------------------#
trainpath = readpath +'train/'
# Read images
img_list=os.listdir(trainpath)
"""
Given info about dataset: 30 classes with 21 images for each class
naming is such that xx_yy.png where xx is class number and yy the image number
"""
Num_class = 30
Num_img = 21
# Loop over training images
x= []
train_labels = []
for i in range(len(img_list)):
    # read image
    img= cv2.imread(trainpath+img_list[i],0)
    train_labels.append(int(img_list[i].split('_')[0]))
    # convert to vectorize format
    vec_img = np.reshape(img,(img.shape[0]*img.shape[1]))
    # normalize image
    vec_img = vec_img/np.linalg.norm(vec_img)
    #store
    x.append(list(vec_img))

x= np.array(x).T
# find mean of images
m = np.mean(x,axis=1)
# make X matrix
X = x-np.kron(m,np.ones((np.shape(x)[1],1))).T
# save data
np.save(savepath+'x.npy',x)
np.save(savepath+'m.npy',m)
np.save(savepath+'capX.npy',X)
np.save(savepath+'train_labels.npy',train_labels)
##-------------------------Testing-----------------------------------------------#
testpath = readpath+'test/'
test_img_list = os.listdir(testpath)
# read test images
test_labels=[]
test_x=[]
for i in range(len(test_img_list)):
```

```
    # read image
    img= cv2.imread(testpath+test_img_list[i],0)
    test_labels.append(int(test_img_list[i].split('_')[0]))
    # convert to vectorize format
    vec_img = np.reshape(img,(img.shape[0]*img.shape[1]))
    # normalize image
    vec_img = vec_img/np.linalg.norm(vec_img)
    #store
    test_x.append(list(vec_img))

test_x= np.array(test_x).T
# substract mean
test_X = test_x-np.kron(m,np.ones((np.shape(test_x)[1],1))).T
# save data
np.save(savepath+'test_x.npy',test_x)
np.save(savepath+'captest_X.npy',test_X)
np.save(savepath+'test_labels.npy',test_labels)
#-------------------------------------------------------------------------------#
#---------------------------------- load data ----------------------------------#
x = np.load(savepath+'x.npy')
m = np.load(savepath+'m.npy')
X = np.load(savepath+'capX.npy')
test_x = np.load(savepath+'test_x.npy')
test_X= np.load(savepath+'captest_X.npy')
train_labels = np.load(savepath+'train_labels.npy')
test_labels = np.load(savepath+'test_labels.npy')
#-------------------------------------PCA---------------------------------------#
# find eigen vectors using the trick
w_pca,lam,sorted_lam = MyCV.eig_rankdef(X,0)
# define number of eigen vectors to be taken into consideration
p = np.arange(20)
# make list of eigen vectors w for every p
W_pca=[]
for i in range(len(p)):
    temp=[]
    for j in range(p[i]+1):
        temp.append(w_pca[:,sorted_lam[j]])
    temp=np.array(temp).T
    W_pca.append(temp)
#-------------------------------------------------------------------------------#

#-------------------------------------LDA---------------------------------------#
# find class means m_i
m_i = []
for i in range(Num_class):
    temp_x = x[:,i*Num_img:(i+1)*Num_img]
    m_i.append(np.mean(temp_x,axis=1))
# Find Sb and Sw
M = np.array(m_i).T
M = M-np.kron(m,np.ones((np.shape(M)[1],1))).T

## find eigen vectors of Sb using the computational trick
V,mu,sorted_mu = MyCV.eig_rankdef(M,0)
mu = (1/Num_class)*mu
Y = np.zeros_like(V)
Db = np.zeros_like(mu)
for i in range(np.shape(V)[1]):
    Y[:,i]=V[:,sorted_mu[i]]
    Db[i]=mu[sorted_mu[i]]
# find Z
Z = Y@np.linalg.inv(np.diag(np.sqrt(Db)))
# now need to diagonalize Z.T Sw Z using the same trick
Xw = x - np.kron(M,np.ones((1,Num_img)))
ZtXw = Z.T@Xw
U,gamma,sorted_gamma=MyCV.eig_rankdef(ZtXw) # in ascending order
```

```
# make list of eigen vectors w for every p
W_Lda=[]
for i in range(len(p)):
    U_hat=[]
    for j in range(p[i]+1):
        U_hat.append(U[:,sorted_gamma[j]])
    U_hat = np.array(U_hat).T
    Wp = Z@U_hat
    #normalize Wp
    Wp= Wp/np.linalg.norm(Wp, axis=0)
    W_Lda.append(Wp)

#-----------------------------------------LDA-----------------------------------------#

#-------------------Plot-------------------------#
# loop over p: chosen sub-space dimnesion
PCA_accuracy = []
LDA_accuracy = []
for i in range(len(p)):
    pca_detected_correctly = 0
    lda_detected_correctly = 0
    iW_pca = W_pca[p[i]]
    iW_lda = W_Lda[p[i]]
    # project training images using iW
    y_pca = iW_pca.T@X
    y_lda = iW_lda.T@X
    # project test images using iW
    test_y_pca = iW_pca.T@test_X
    test_y_lda = iW_lda.T@test_X
    # find Euclidean distance of test_y col vector from all y col vector
    # and use 1-NN classifier
    for j in range(len(test_img_list)):
        dist_pca = (y_pca.T - test_y_pca[:,j]).T
        dist_lda = (y_lda.T - test_y_lda[:,j]).T
        dist_pca = np.linalg.norm(dist_pca,axis=0)
        dist_lda = np.linalg.norm(dist_lda,axis=0)
        j_min_dis_pca = np.argmin(dist_pca)
        j_min_dis_lda = np.argmin(dist_lda)
        if train_labels[j_min_dis_pca]==test_labels[j]:
            pca_detected_correctly+=1
        if train_labels[j_min_dis_lda]==test_labels[j]:
            lda_detected_correctly+=1
    i_accuracy_pca=pca_detected_correctly/len(test_img_list)
    i_accuracy_lda=lda_detected_correctly/len(test_img_list)
    PCA_accuracy.append(i_accuracy_pca)
    LDA_accuracy.append(i_accuracy_lda)

import matplotlib.pyplot as plt
plt.plot(p,PCA_accuracy,c='b',label='PCA')
plt.scatter(p,PCA_accuracy,marker='*')
plt.plot(p,LDA_accuracy,c='r',label='LDA')
plt.scatter(p,LDA_accuracy,marker='^')
plt.xlabel('Dimension of Sub-Space (p)')
plt.ylabel('Accuracy of detection')
plt.legend()
plt.show()
```

3.1.2 Functions:

```
#%%-------------------------------------PCA & LDA-------------------------------------#
# function for finding the eigen vectors for the reduced dimensional space
def eig_rankdef(X, ascending=1):
    """
    Input: X a matrix of
           ascending: option 1 or 0(default=1) if 0 then output of
           indices is in descending order
    Output: w:all eigen vectors
            sorted_lam: list of eigen values indices in descending order
    using the computational trick by first finding the eigne vectors u of
    X.TX and then converting to w
    """
    # find eigen vectors of XTX
    lam,u = np.linalg.eig(X.T@X)
    sorted_lam = np.argsort(lam)
    # reverse the list of sorted _lam to make it in descending order
    if ascending == 0:
        sorted_lam= sorted_lam[::-1]
    # convert u to w
    w = X@u
    # normalize w
    w = w/np.linalg.norm(w, axis=0)
    return(w,lam,sorted_lam)
#-------------------------------------PCA & LDA-------------------------------------#
```

3.2 AdaBoost

3.2.1 Main File:

```
"""
ECE661: hw10 Part-2: ADABOOST
@author: Rahul Deshmukh
email: deshmuk5@purdue.edu
PUID: 0030004932
"""
#%% ---------------------- Import libraries ----------------------------------#
import numpy as np
import sys
sys.path.append('../../')
import MyCVModule as MyCV
import pickle
import matplotlib.pyplot as plt
#%%--------------------------------------------------------------------------#
# define read path
readpath = '../files/cars/'
# define tasks to be carried out
generate_training_data = 0
generate_testing_data = 0

do_training=0
do_testing=0

#%%------------------- Generate Data-------------------------------------------#
#------------------------------ Training Data-------------------------------#
train_path= readpath+'train/'
#-------------read images->Integrate image-> features ---------------#
if generate_training_data:
    S,label,N_pos,N_neg = MyCV.Read_img4AdaBoost(train_path)
    np.save(train_path+'train_data/'+'S.npy',S)
    np.save(train_path+'train_data/'+'label.npy',label)
    np.save(train_path+'train_data/'+'N_pos.npy',N_pos)
    np.save(train_path+'train_data/'+'N_neg.npy',N_neg)
    train_features=[]
    for i in range(len(S)):
        ifeature = MyCV.get_feature(S[i])
        train_features.append(ifeature)
    np.save(train_path+'train_data/'+'f.npy',train_features)

    Num_classifier = len(train_features[1])
    np.save(train_path+'train_data/'+'Num_classifier.npy',Num_classifier)
else:
    S_train = np.load(train_path+'train_data/'+'S.npy')
    train_label = np.load(train_path+'train_data/'+'label.npy')
    train_features = np.load(train_path+'train_data/'+'f.npy')
    Num_classifier=np.load(train_path+'train_data/'+'Num_classifier.npy')

#------------------------- Testing Data-------------------------------------#
test_path =readpath+'test/'
#-------------read images->Integrate image-> features ---------------#
if generate_testing_data:
    S,label,N_pos,N_neg = MyCV.Read_img4AdaBoost(test_path)
    np.save(test_path+'test_data/'+'S.npy',S)
    np.save(test_path+'test_data/'+'label.npy',label)
    np.save(test_path+'test_data/'+'N_pos.npy',N_pos)
    np.save(test_path+'test_data/'+'N_neg.npy',N_neg)
    test_features=[]
    for i in range(len(S)):
        ifeature = MyCV.get_feature(S[i])
        test_features.append(ifeature)
    np.save(test_path+'test_data/'+'f.npy',test_features)
```

```
else:
    S_test = np.load(test_path+'test_data/'+'S.npy')
    test_label = np.load(test_path+'test_data/'+'label.npy')
    test_features = np.load(test_path+'test_data/'+'f.npy')

#%%------------------------------ Do Training ---------------------------------------#

# set parameters for cascading and adaboost
Smax =10
#adaboost params
Tmax=100
TP_crit=1
FP_crit=0.5
if do_training:
    Strong_classifiers,TP_s,FP_s = MyCV.do_cascades(train_features,train_label,
                                                    Num_classifier,Smax,
                                                    Tmax,TP_crit,FP_crit)
    fid = open(train_path+'train_data/'+'Strong_classifiers.pkl','wb')
    pickle.dump(Strong_classifiers,fid)
    fid.close()
    np.save(train_path+'train_data/'+'TP_s.npy',TP_s)
    np.save(train_path+'train_data/'+'FP_s.npy',FP_s)
else:
    fid= open(train_path+'train_data/'+'Strong_classifiers.pkl','rb')
    Strong_classifiers= pickle.load(fid)
    fid.close()
    TP_s=np.load(train_path+'train_data/'+'TP_s.npy')
    FP_s=np.load(train_path+'train_data/'+'FP_s.npy')

print('Trained Strong Classifier with Final TP: '+str(np.prod(TP_s[:-1]))+
      ' FP: '+str(np.prod(FP_s[:-1])))
#%%------------------------------ Do Testing----------------------------------------#

if do_testing:
    print('-----------------------Testing-----------------------------------------')
    FP_measure,FN_measure=MyCV.Cascade_Testing(test_features,test_label,Strong_classifiers)
    np.save(test_path+'test_data/'+'FP_measure.npy',FP_measure)
    np.save(test_path+'test_data/'+'FN_measure.npy',FN_measure)
else:
    FP_measure = np.load(test_path+'test_data/'+'FP_measure.npy')
    FN_measure = np.load(test_path+'test_data/'+'FN_measure.npy')

# make plot for FP and FN
s=np.arange(1,len(Strong_classifiers)+1,dtype=np.float16)
plt.plot(s,FP_measure,color='r',marker='*',label='FP')
plt.plot(s,FN_measure,color='b',marker='*',label='FN')
plt.xlabel('Stage #')
plt.ylabel('FP or FN')
plt.title('Performance of Strong Classifier')
plt.ylim([0,1])
plt.xlim([1,s[-1]])
plt.legend()
plt.show()
```

3.2.2 Functions:

```
#%%-------------------------------------ADABoost-------------------------------------------#
#%%-------------------------------- Testing ------------------------------------------------#
# function for running cascades of adaboost for testing
def Cascade_Testing(all_features,all_labels,Strong_classifiers):
    """
    Input: all_features:  matrix of all test features with single row as a feature
                          vector for one image
           known_labels: lables of all test images, a array of 1,0
           Strong_classifiers: Learned Classifier in the format
           a list of list with one element in the list as list of 5 elements:-
           1)h_t: index of the classifier to be used
           2)theta_t: threhold value for the classifer
           3)p_t: polarity of the classifier
           4)alpha_t: list of trust factors
           5)Cx_threshold: threshold for the strong classifiers
    Output: FP_measure,FN_measure: list of stage-wise values of FP and FN
    for the test data
    """
    # find number of stages, N_pos and N_neg for test images
    Num_stages= len(Strong_classifiers)
    N_pos = np.sum(all_labels)
    N_neg = len(all_labels)-N_pos
    original_idx = np.arange(N_pos+N_neg)

    # initialize counts for FP and FN
    FP_measure = []
    FN_measure = []
    FP_count = 0
    TN_count = 0

    known_label=all_labels
    this_stage_idx = original_idx
    print('FP\tFN')
    for s in range(Num_stages):
        # get the Strong classifier parameters for this stage
        h_t,theta_t,p_t,alpha_t,Cx_threshold = Strong_classifiers[s]
        # find number of weak classifiers
        T = len(alpha_t)
        # predict the class of the testing data for this stage
        # make features and known_labels
        known_label = all_labels[this_stage_idx]
        features = all_features[this_stage_idx,:]
        sN_pos = np.sum(known_label)
        sN_neg = len(known_label)-sN_pos
        predicted_label = AdaBoost_Testing(features,known_label,T,sN_pos,sN_neg,
                          h_t,theta_t,p_t,alpha_t)
        # calculate FP and FN
        num_correctly_classified_positives = np.sum(np.multiply(known_label,predicted_label)
            )
        num_misclassified_positives = sN_pos - num_correctly_classified_positives
        FP_count+=num_misclassified_positives

        num_correctly_classified_negatives = np.sum(np.multiply(1-predicted_label,1-
            known_label))
        TN_count += num_correctly_classified_negatives

        # store FP and TN for this stage
        sFP = (N_neg-TN_count)/N_neg
        FP_measure.append(sFP)

        sFN = FP_count/N_pos
        FN_measure.append(sFN)
        print('----------------------Stage '+str(s+1)+'----------------------')
```

```
        print(sFP,sFN)
        # update data: send only those positive images which were correctly classified
        # and those negative images which where wrongly classified for the next stage
        this_stage_idx = np.where(predicted_label>0)[0]
    return(FP_measure,FN_measure)
# function for predicting class for one stage of adaboost testing
def AdaBoost_Testing(features,known_label,T,N_pos,N_neg,
                     h_t,theta_t,p_t,alpha_t):
    """
    Input: features: matrix of test features with single row as a feature
                          vector for one image
           known_label: labels of test samples
           T: number of weak classifiers in this stage
           h_t: index of the classifier to be used
           theta_t: threhold value for the classifer
           p_t: polarity of the classifier
           alpha_t: list of trust factors
    Output: predicted_label: array of 1,0 1:predicted as pos 0:predicted as neg
    """
    # initialize
    predicted_label = np.zeros(N_pos+N_neg)

    h_q = np.zeros((N_pos+N_neg,T)) # array for storing value of weak classifier
    # for the query data

    # loop over weak classifier
    for t in range(T):
        fi = features[:,h_t[t]]
        for i in range(N_pos+N_neg):
            if p_t[t]*fi[i]<p_t[t]*theta_t[t]:
                h_q[i,t]= 1

    # find sum h_t*a_t for all query data
    Cx_q = h_q@np.array(alpha_t)
    Cx_threshold= 0.5*np.sum(alpha_t)

    # find predicted labels using the threshold
    predicted_label = np.where(Cx_q>=Cx_threshold,1,0)

    return(predicted_label)
#%%-----------------------------------Training-------------------------------------------#
#function for performing one stage of cascade
def do_cascades(all_features,all_labels,
                Num_classifier,Smax,
                Tmax,TP_crit,FP_crit):
    """
    Input:Smax: maximum number of cascades allowed
          all_features: feature vector matrix with feature vec for one image
          all_labels: list of all labels
          stacked as a row vector
          ------ criterias for Adaboost-------
          Tmax: max num of adaboost iters
          TP_crit,FP_crit: true positive and false positive criterias (0-1)
          ---------------------------------------
    Output:Strong_classifiers: list of strong classifiers
           a list of list with one element in the list as list of 5 elements:-
           1)h_t: index of the classifier to be used
           2)theta_t: threhold value for the classifer
           3)p_t: polarity of the classifier
           4)alpha_t: list of trust factors
           5)Cx_threshold: threshold for the strong classifiers
           TP_s: list of TP for each stage
           FP_s: list of FP for each stage
    """
    # initialize storing structures
```

```
    Strong_classifiers = []
    TP_s=[]
    FP_s=[]

    N_pos = np.sum(all_labels)
    N_neg = len(all_labels)-N_pos

    original_idx = np.arange(N_pos+N_neg)
    this_stage_idx = original_idx

    for s in range(Smax):
        # make feature matrix, label, N_pos,N_neg for adbaboost call
        known_label = all_labels[this_stage_idx]
        features = all_features[this_stage_idx,:]
        N_pos = np.sum(known_label)
        N_neg = len(known_label)-N_pos
        # check if all negative samples are exhausted
        if N_neg==0:
            print('Converged at Stage '+str(s))
            break;
        print('------------ Stage:  '+str(s+1)+'-------------')
        print('Total images: '+str(N_pos+N_neg)+' Pos: '+str(N_pos)+' Neg: '+str(N_neg))
        #call adaboost
        h_t,theta_t,p_t,alpha_t,Cx_threshold,samples_classified_pos_idx,TP_final,FP_final=\
        Do_AdaBoost(features,Tmax,TP_crit,FP_crit,N_pos,N_neg,Num_classifier,known_label)
        #store this strong classifier
        Strong_classifiers.append([h_t,theta_t,p_t,alpha_t,Cx_threshold])
        TP_s.append(TP_final)
        FP_s.append(FP_final)
        # update indices for next stage of adaboost
        this_stage_idx = this_stage_idx[samples_classified_pos_idx]
    return(Strong_classifiers,TP_s,FP_s)

# function for carrying out AdaBoost Learning from Training data set
def Do_AdaBoost(features,Tmax,TP_crit,FP_crit,
                N_pos,N_neg,Num_classifier,known_label):
    """
    Input: features: feature vectors stacked as rows and one row for on image
           TP_crit,FP_crit: true positive and false positive criterias(0-1)
           Tmax: max time iterations to be carried by AdaBoost
           N_pos, N_neg: Number of positive and negative samples
           Num_classifier: number of classfiers in the library
           known_label: true labels of the images
    Output: Strong classifer data
            h_t: index of the classifier to be used
            theta_t: threhold value for the classifer
            p_t: polarity of the classifier
            alpha_t: list of trust factors
            samples_classified_pos_idx : indices of samples
                which will go for the next stage
            TP_final: last value of the measured TP
            FP_final: last value of the measured FP
    """

    #initialize storing stuctures for storing the classifiers ht
    h_t=[]
    theta_t=[]
    p_t=[]
    h_t_predicted_labels=[]
    alpha_t =[] #alpha: trust factors
    f_t = [] #list for storing the feature values

    # initialize weights for samples
    wt= np.hstack((1/(2*N_pos)*np.ones(N_pos),1/(2*N_neg)*np.ones(N_neg)))
```

```
    # tracking varibles
    TP_measure=[]
    FP_measure=[]

    for t in range(Tmax):
        # normalize weights
        wt= wt/np.sum(wt)
        # find best weak classifier
        h_star,theta_star,p_star,error_star,predicted_label,f_star=\
        find_best_weak_classifier(Num_classifier,known_label,
                                  wt,features,N_pos,N_neg)
        # compute alpha
        beta = error_star/(1-error_star)
        a_t = np.log(1/beta)
        #store t'th best weak classifer
        alpha_t.append(a_t)
        h_t.append(h_star);
        theta_t.append(theta_star);
        p_t.append(p_star);
        h_t_predicted_labels.append(predicted_label)
        f_t.append(f_star)
        #update wts
        wt = np.multiply(wt,np.power(beta,1-np.logical_xor(predicted_label,
                                                           known_label).astype(int)))
        # build current Strong classifier and find its acuracy
        Cx = np.array(h_t_predicted_labels).T@np.array(alpha_t) # this is just the value of
            the classifier
        # we need to compare it to a threshold value
        # instead of choosing the threshol as (1/2)*np.sum(a_t)
        # we choose the threshold such that all the positive images get classified as 1
        # such a threshold will be the minimum of Cx slice corresponding to positive images
        Cx_threshold = np.min(Cx[:N_pos])
        # find predicted labels for the strong classifer
        Cx_predicted_label = np.where(Cx>=Cx_threshold,1,0)
        # find accuracy of current strong classifier
        # find true positive accuracy
        TP_count = np.sum(Cx_predicted_label[:N_pos])/N_pos
        TP_measure.append(TP_count)
        # find false positive accuracy
        FP_count = np.sum(Cx_predicted_label[N_pos:])/N_neg
        FP_measure.append(FP_count)
        print(TP_count,FP_count)
        # check for convergence
        if TP_count>=TP_crit and FP_count<=FP_crit:
            print('AdaBoost Early Termination: reached convergence in '+
                  str(t+1)+' iterations')
            break
    if t==Tmax-1:
        print('!! Adaboost did not meet TP and FP criterias !!')
    print('TP: '+str(TP_measure[-1]))
    print('FP: '+str(FP_measure[-1]))
    TP_final = TP_measure[-1]
    FP_final = FP_measure[-1]
    # find number the negative example wrongly predicted as positive and all
    # positive images
    samples_classified_pos_idx = np.where(Cx_predicted_label>0)[0]
    return(h_t,theta_t,p_t,alpha_t,Cx_threshold,samples_classified_pos_idx,
           TP_final,FP_final)

# function for finding the best weak classifier
def find_best_weak_classifier(Num_classifier,label,
                              wt,features,N_pos,N_neg):
    """
    Input: Num_classifier: number of classifiers
           label: true labels for all training images np.array
```

```
            wt: t'th iteration weights for s'th iter samples np.array
        Output: h_star: index number of the best classifier
                theta_star: threshold for the best classifier
                p_star: polarity of the best classifier
                f_star: array of feature vlaues for the best classifier
                error_star: weighted error for the best classifier
                predicted_label: binary list, classification of the images using the
                best classifier
        """
        # calculate T+ and T-
        T_plus = np.sum(wt[:N_pos])
        T_plus = T_plus*np.ones(N_pos+N_neg)
        T_minus= np.sum(wt[N_pos:])
        T_minus = T_minus*np.ones(N_pos+N_neg)
        # loop over all calssifiers to find the best classifier with min error
        error_star = np.inf # initialization
        for i in range(Num_classifier):
            fi = features[:,i]
            # sort the feature vector in increasing order
            sorted_idx = np.argsort(fi)
            sorted_label = label[sorted_idx]
            sorted_wt = wt[sorted_idx]
            # Calculate S+ and S-: for all values of threshold
            S_plus= np.multiply(sorted_wt,sorted_label)
            S_plus = np.cumsum(S_plus)
            S_minus = np.cumsum(sorted_wt)-S_plus
            #find the two types of error
            error1 = S_plus+(T_minus-S_minus)
            error2 = S_minus+(T_plus-S_plus)
            # find min error
            element_wise_min = np.minimum(error1,error2)
            min_id = np.argmin(element_wise_min)
            min_error = element_wise_min[min_id]
            # update the best ever if current_error is less than error_star
            if min_error<error_star:
                error_star=min_error
                # find polarity and prediction by the classifier
                predicted_label = np.zeros(N_pos+N_neg)
                if error1[min_id]<=error2[min_id]:
                    p=-1
                    predicted_label[min_id:] = 1
                else:
                    p=1
                    predicted_label[:min_id] = 1
                predicted_label[sorted_idx]=np.copy(predicted_label) # binary classification 1
                    or 0
                # store best polarity, classifier index, threshold value
                p_star= p
                h_star=i
                theta_star= fi[sorted_idx[min_id]]
                f_star = fi
        return(h_star,theta_star,p_star,error_star,predicted_label,f_star)

    #%% helper function for adaboost to read images
    def Read_img4AdaBoost(train_path):
        """
        Input: train_path: path for training dataset
        Output: S: is a list of integral images
                label: list of 1 or 0 1: positive 0:negative
        """
        pos_path = train_path+'positive/'
        neg_path = train_path +'negative/'
        S=[];
        # read postive images
        pos_img_list = os.listdir(pos_path)
```

```
    pos_img_list.sort(key = lambda x: int(x.split('.')[0]))
    N_pos =len(pos_img_list)
    for i in range(N_pos):
        iI = cv2.imread(pos_path+pos_img_list[i],0) # gray image
        iI = iI.astype(float)
        iS = Integrate_Img(iI)
        S.append(iS)
    # read negative images
    neg_img_list = os.listdir(neg_path)
    neg_img_list.sort(key = lambda x: int(x.split('.')[0]))
    N_neg=len(neg_img_list)
    for i in range(N_neg):
        iI = cv2.imread(neg_path+neg_img_list[i],0) # gray image
        iI = iI.astype(float)
        iS = Integrate_Img(iI)
        S.append(iS)
    label = np.hstack((np.ones(N_pos,dtype=int),np.zeros(N_neg,dtype=int)))
    return(S,label,N_pos,N_neg)
#%%function for finding the Haar like operators
def get_feature(S):
    """
    Input:S: integral Image with left and top padding
    Output: features
    """
    m,n=np.shape(S);
    m=m-1; n=n-1;
    feature=[]
#-------------------All Classifiers-------------------------------------------#
# for 20x40 image we will have a total of 312260 classifiers which is huge!!
#    # type 1: horizontal operator [0|1]
#    for h in range(1,m+1):
#        for w in range(1,n//2+1):
#            #operator is of size h,2*w
#            # shift x & y position of upper left corner of operator
#            for x in range(n-2*w+1):
#                for y in range(m-h+1):
#                    feature.append(type1_haar(S,x,y,h,w))

#    # type 2: vertical operator [0|1]^T
#    for h in range(1,m//2+1):
#        for w in range(1,n+1):
#            #operator is of size 2*h,w
#            # shift x & y position of upper left corner of operator
#            for x in range(n-w+1):
#                for y in range(m-2*h+1):
#                    feature.append(type2_haar(S,x,y,h,w))

#    # type 3: x derivative type operator [0|1|0]
#    for h in range(1,m+1):
#        for w in range(1,n//3+1):
#            #operator is of size h,3*w
#            # shift x & y position of upper left corner of operator
#            for x in range(n-3*w+1):
#                for y in range(m-h+1):
#                    feature.append(type3X_haar(S,x,y,h,w))

#    # type 3_2: Y derivative type operator [0|1|0]
#    for h in range(1,m//3+1):
#        for w in range(1,n+1):
#            #operator is of size h,3*w
#            # shift x & y position of upper left corner of operator
#            for x in range(n-w+1):
#                for y in range(m-3*h+1):
#                    feature.append(type3Y_haar(S,x,y,h,w))
```

```
#    # type 4: diagonal operator [[0,1][1,0]]
#    for h in range(1,m//2+1):
#        for w in range(1,n//2+1):
#            #operator is of size 2*h,2*w
#            # shift x & y position of upper left corner of operator
#            for x in range(n-2*w+1):
#                for y in range(m-2*h+1):
#                    feature.append(type4_haar(S,x,y,h,w))
#-------------------------------------------------------------------------#
# to reduce computational cost i am picking a subset of the all classifiers
# such that i still have a mix of all types
    k=1
    # type 1: horizontal operator [0|1]
    for h in [k]:
        for w in range(1,n//2+1):
            #operator is of size h,2*w
            # shift x & y position of upper left corner of operator
            for x in range(n-2*w+1):
                for y in range(m-h+1):
                    feature.append(type1_haar(S,x,y,h,w))
    # type 2: vertical operator [0|1]^T
    for h in range(1,m//2+1):
        for w in [k]:
            #operator is of size 2*h,w
            # shift x & y position of upper left corner of operator
            for x in range(n-w+1):
                for y in range(m-2*h+1):
                    feature.append(type2_haar(S,x,y,h,w))
    # type 3: x derivative type operator [0|1|0]
    for h in [k]:
        for w in range(1,n//3+1):
            #operator is of size h,3*w
            # shift x & y position of upper left corner of operator
            for x in range(n-3*w+1):
                for y in range(m-h+1):
                    feature.append(type3X_haar(S,x,y,h,w))
    # type 3_2: Y derivative type operator [0|1|0]^T
    for h in range(1,m//3+1):
        for w in [k]:
            #operator is of size h,3*w
            # shift x & y position of upper left corner of operator
            for x in range(n-w+1):
                for y in range(m-3*h+1):
                    feature.append(type3Y_haar(S,x,y,h,w))
    # type 4: diagonal operator [[0,1][1,0]]
    for h in [k]:
        for w in [k]:
            #operator is of size 2*h,2*w
            # shift x & y position of upper left corner of operator
            for x in range(n-2*w+1):
                for y in range(m-2*h+1):
                    feature.append(type4_haar(S,x,y,h,w))
    return(feature)
#function for finding feature value of type4 haar
def type4_haar(S,x,y,h,w):
    """
    Input:  S: integral image with left and top padding
            x,y: coordinates of A
            h,w: half-height and half-width of the operator
    Output: f:feature value
     A-------B-------C
     |   0   |   1   |
     D-------E-------F
     |   1   |   0   |
     G-------H-------I
```

```
    """
    A = S[y,x]
    B = S[y,x+w]
    C = S[y,x+2*w]
    D = S[y+h,x]
    E = S[y+h,x+w]
    F = S[y+h,x+2*w]
    G = S[y+2*h,x]
    H = S[y+2*h,x+w]
    I = S[y+2*h,x+2*w]
    f = (F-C-E+B)+(H-E-G+D)-(I-F-H+E)-(E-B-D+A)
    return(f)
#function for finding feature value of type3 haar
def type3X_haar(S,x,y,h,w):
    """
    X double derivative
    Input:  S: integral image with left and top padding
            x,y: coordinates of A
            h,w:height and 1/3rd-width of the operator
    Output: f:feature value
     A-------B-------C-------D
     |   0   |   1   |   0   |
     E-------F-------G-------H
    """
    A = S[y,x]
    B = S[y,x+w]
    C = S[y,x+2*w]
    D = S[y,x+3*w]
    E = S[y+h,x]
    F = S[y+h,x+w]
    G = S[y+h,x+2*w]
    H = S[y+h,x+3*w]
    f = (G-C-F+B)-(F-B-E+A)-(H-D-G+C)
    return(f)
#function for finding feature value of type3_2 haar
def type3Y_haar(S,x,y,h,w):
    """
    Y double derivative
    Input:  S: integral image with left and top padding
            x,y: coordinates of A
            h,w: 1/3rd-height and width of the operator
    Output: f:feature value
     A-------E
     |   0   |
     B-------F
     |   1   |
     C-------G
     |   0   |
     D-------H
    """
    A = S[y,x]
    B = S[y+h,x]
    C = S[y+2*h,x]
    D = S[y+3*h,x]
    E = S[y,x+w]
    F = S[y+h,x+w]
    G = S[y+2*h,x+w]
    H = S[y+3*h,x+w]
    f = (G-C-F+B)-(F-B-E+A)-(H-D-G+C)
    return(f)
#function for finding feature value of type2 haar
def type2_haar(S,x,y,h,w):
    """
    Input:  S: integral image with left and top padding
            x,y: coordinates of A
```

```
            h,w: half-height and width of the operator
        Output: f:feature value
         A-------D
         |   0   |
         B-------E
         |   1   |
         C-------F
        """
        A = S[y,x]
        B = S[y+h,x]
        C = S[y+2*h,x]
        D = S[y,x+w]
        E = S[y+h,x+w]
        F = S[y+2*h,x+w]
        f = (F-E-C+B)-(E-B-D+A)
        return(f)
#function for finding feature value of type1 haar
def type1_haar(S,x,y,h,w):
        """
        Input:  S: integral image with left and top padding
                x,y: coordinates of A
                h,w:height and half-width of the operator
        Output: f:feature value
         A-------B-------C
         |   0   |   1   |
         D-------E-------F
        """
        A = S[y,x]
        B = S[y,x+w]
        C = S[y,x+2*w]
        D = S[y+h,x]
        E = S[y+h,x+w]
        F = S[y+h,x+2*w]
        f = (F-C-E+B)-(E-B-D+A)
        return(f)
#%% function for finding intergral representation of image
def Integrate_Img(I):
        """
        using summed table approach
        Input: I: grayscale image mxn
        Output: S: integral image, m+1xn+1 array with left and top padding
        """
        S= np.zeros((np.shape(I)[0]+1,np.shape(I)[1]+1)) # padded with zeros
        for y in range(I.shape[0]):
            for x in range(I.shape[1]):
                S[y+1,x+1]=I[y,x]+S[y+1,x]+S[y,x+1]-S[y,x]
        return(S)
#-------------------------------------ADABoost-------------------------------------#
```