Rehana Mahfuz (rmahfuz@purdue.edu)
ECE 661 HW10

Part 1: Face Recognition

Dataset Preprocessing

Both training and testing images are converted to a vectorized format. Each image of dimension 128x128, which means each image in vectorized format will be a vector x_i of length 16384. Each set (training set and testing set) has 21 different images of 30 different people, which makes the total number of images in each set 630. A matrix X is formed with each column representing a vectorized image. For each set, the mean image m is calculated as:

m = (1/N)*$\sum_{i=1}^{N} x_i$

In the matrix X, from each column, the mean image is subtracted. So X becomes

X = [x_1-m x_2-m   … x_N-m]. Each column vector in X which represents each image is normalized by dividing that column by its norm.

Principal Components Analysis (PCA)

The covariance matrix is C = (1/N)* $\sum_{i=1}^{N} XX^T$. The eigenvectors corresponding to the largest k eigenvalues of C form the PCA feature set, represented as W_K, with each column being an eigenvector. k is the number of principal components we choose to retain. A vectorized image x_i's projection into this eigenspace can be computed as y_i = W_K^T (x_i – m), where m is the mean of all images in that set.

Since performing the eigendecomposition of the 16384 X 16384 covariance matrix is computationally heavy, a computation trick is used to find those eigenvectors instead. Eigendecomposition of the matrix X^TX is performed. Only the eigenvectors corresponding to the k largest eigenvalues are arranged into a matrix U, where each column is an eigenvector. Each of those eigenvectors is normalized. To get eigenvectors of the covariance matrix, we perform W=XU. Each of the columns of W is again normalized to obtain W_k, which is the PCA feature set.

Linear Discriminant Analysis (LDA)

The goal of LDA is to find eigenvectors w that maximize the Fisher Discriminant Function,

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

Where S_B is the between-class covariance matrix computed as S_B = (1/N_C)*$\sum_{j=1}^{N_C}(m_j - m)(m_j - m)^T$, where N_C is the number of classes in the training set, m_j is the mean of class j and m is the mean of all classes.

S_W is the within-class covariance matrix computed as S_W = (1/N_C)*$\sum_{j=1}^{N_C} 1/N_j \sum_{i=1}^{N_j}(x_{ij} - m_j)(x_{ij} - m_j)^T$,

Where N_j is the number of images in class j, and x_{ij} is the i^{th} image of class j.

Since it is likely that S_W is singular, Yu and Yang's algorithm is used to find the feature set. Here is the algorithm's description:

For each class k, a mean image m_k is calculated as m_k = 1/||C_K||*$\sum_{i \text{ belongs to } C_K} x_i$.

Rehana Mahfuz (rmahfuz@purdue.edu)
ECE 661 HW10

We find the eigendecomposition of S_B using the same computational trick used for PCA. We retain only the eigenvectors corresponding to the largest k eigenvalues, and arrange them as columns in a matrix Y. Next, we find D_B, the upper-left k x k submatrix of the diagonalized eigenvalues of S_B:

$D_B = (Y^TM)(Y^TM)^T$

Then we calculate $Z = YD_B^{-0.5}$

We use the same computational trick again to find the eigendecomposition of $Z^TS_WZ = (Z^TX_W)(Z^TX_W)^T$. We find the eigenvectors corresponding to the smallest k eigenvalues and arrange them as columns in a matrix U. The transpose of the projection matrix is $W_P^T = U^TZ^T$. The projection y_i of any vectorized image x_i can be calculated as $y_i = W_K^T(x_i - m)$, where m is the mean of all images in that set (training or testing).

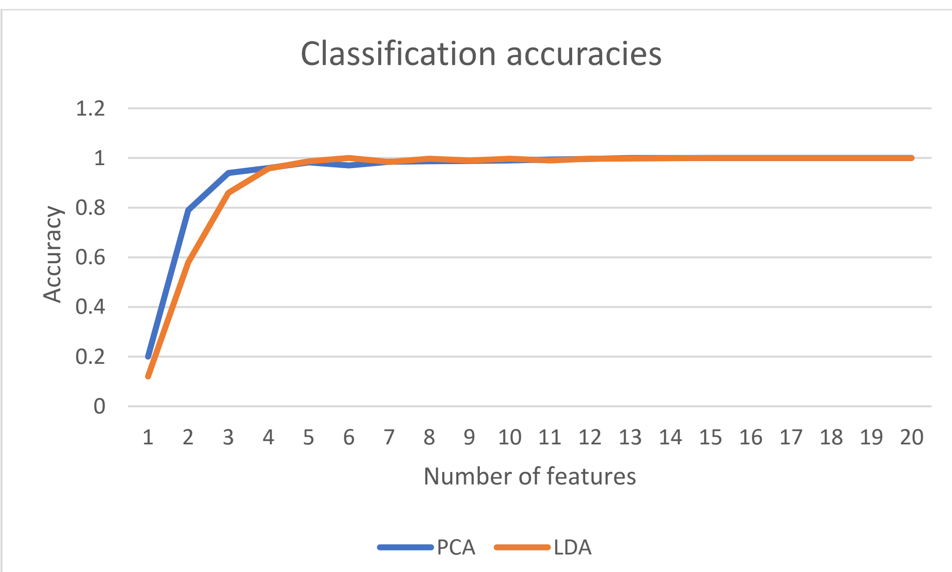
Nearest Neighbors Classification

First, the PCA feature set of the training set is computed, and each training image is projected into this space. Next, each test image is also projected into this space. To classify a test image into one of the 30 classes, the Euclidean distance is found between the projection of the test image and the projection of each of the training images. The training image which is closest to the test image reveals which class the test image should be classified into. The accuracy is computed as:

Accuracy = number of correct classifications/total number of classifications

Results

Both PCA and LDA were performed to obtain a different number of features each time to perform nearest neighbors classification, where the number of features ranged from 1 to 20. Here is the resulting plot showing the accuracies obtained in each case:



Comparison on Results for PCA and LDA

Rehana Mahfuz (rmahfuz@purdue.edu)
ECE 661 HW10

At low number of dimensions, PCA performs better than LDA. At 5 number of dimensions, LDA overtakes PCA. Also, LDA reaches 100% accuracy at k=6, which is much faster than PCA, which only achieves 100% accuracy at n=13. Generally, at higher dimensions, LDA is more reliable.

Source code

```
import numpy as np

import cv2, os

from scipy import spatial

from scipy.linalg import fractional_matrix_power


path = '../Users/rmahfuz/Desktop/661/HW10/'

num_classes=30; num_samples=21

#===PCA=================================================================================================================

def process_train_pca(k): #k is num of principal components. returns y_train(k x num_samples(630)), W_k

    img_arr = []

    for fn in os.listdir(path + 'ECE661_2018_hw10_DB1/train'):

        #print(fn)

        img=cv2.imread(path + 'ECE661_2018_hw10_DB1/train/' + fn)

        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)#; img=np.array(img, dtype=np.float32)

        img=img.flatten()

        #img=img/np.linalg.norm(img) #normalize to make it illumination-invariant

        img_arr.append(img)

        #print(img.shape)

        #print(img_arr.shape)

    img_arr = np.array(img_arr,dtype=np.float32)

    #print(img_arr.shape)

    mean_img = np.mean(img_arr,axis=0) #mean of each column over all rows

    #reshaped=mean_img.reshape((128,128));cv2.imwrite(path+'mean_img_train.png', reshaped)

    X=(img_arr-mean_img).T
```

```
    for i in range(X.shape[1]):
        X[:,i] = X[:,i]/np.linalg.norm(X[:,i])
    #print('X.shape = ',X.shape)#(16384,630)
    w,v = np.linalg.eig(np.matmul(X.T,X)) #eigenvalues, eigenvectors
    #print('w.shape = ', w.shape)#(630,)
    #print('v.shape = ', v.shape)#(630,630)
    v=v.T
    v=v[np.argsort(w)[::-1]]
    #w.sort(); w=w[::-1]
    v=v.T #each column is an eigenvector
    W=np.matmul(X,v)
    #Normalize each column of W:
    for i in range(W.shape[1]):
        W[:,i] = W[:,i]/np.linalg.norm(W[:,i])
    #Extract the largest/first k columns
    W_k = W[:,:k]
    #print('W_k.shape = ', W_k.shape)#(16384,k)
    #compute the projected y_train:
    y_train = np.matmul(W_k.T,X)
    #print('y_train.shape = ', y_train.shape)#(k,630). Each column represents a sample
    return (y_train, W_k)
#-------------------------------------------------------------------------------------------------------------------
def calc_pca_accuracy(k):
    #(y_train, W_k, mean_img) = process_train(k)
    (y_train, W_k) = process_train_pca(k) #(k,630)

    img_arr = []
    #print(os.listdir(path + 'ECE661_2018_hw10_DB1/test'))
    for fn in os.listdir(path + 'ECE661_2018_hw10_DB1/test'):
```

```
        #print(fn)
        img=cv2.imread(path + 'ECE661_2018_hw10_DB1/test/' + fn)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)#; img=np.array(img, dtype=np.float32)
        img=img.flatten()
        #img=img/np.linalg.norm(img) #normalize to make it illumination-invariant
        img_arr.append(img)
        #print(img.shape) #(128,128)
    img_arr = np.array(img_arr,dtype=np.float32)
    '''#print('img_arr_train.shape = {}, img_arr.shape = {}'.format(img_arr_train.shape, img_arr.shape))
    #img_arr = np.vstack((img_arr_train, img_arr))
    #print(img_arr.shape)
    mean_img = np.mean(img_arr,axis=0) #mean of each column over all rows
    #reshaped=mean_img.reshape((128,128));cv2.imwrite(path+'mean_img_test.png', reshaped)
    X=(img_arr-mean_img).T
    print('X.shape = ',X.shape)
    w,v = np.linalg.eig(np.matmul(X.T,X)) #eigenvalues, eigenvectors
    print('w.shape = ', w.shape)#(630,)
    print('v.shape = ', v.shape)#(630,630)
    v=v.T
    v=v[np.argsort(w)[::-1]]
    w.sort(); w=w[::-1]
    v=v.T #each column is an eigenvector
    W=np.matmul(X,v)
    #Normalize each column of W:
    for i in range(W.shape[1]):
        W[:,i] = W[:,i]/np.linalg.norm(W[:,i])
    #Extract the largest/first k columns
    W_k = W[:,:k]
    print('W_k.shape = ', W_k.shape)#(16384,k)
```

Rehana Mahfuz (rmahfuz@purdue.edu)
ECE 661 HW10

```
    #compute the projected y_test:

    y_test = np.matmul(W_k.T,X)

    print('y_test.shape = ', y_test.shape) #(k,630)'''

    mean_img = np.mean(img_arr,axis=0) #mean of each column over all rows

    X=(img_arr-mean_img).T#(16384,630)

    for i in range(X.shape[1]):

        X[:,i] = X[:,i]/np.linalg.norm(X[:,i])

    y_test = np.matmul(W_k.T,X)

    #print('y_test.shape = ', y_test.shape) #(k,630)

    #-------------------Nearest neighbors classification---------------------------

    #Now I have both y_test and y_train. For each column in y_test, I will try to match it with the nearest
column in y_train.

    acc = [0]*num_classes

    for i in range(y_test.shape[1]):

        dist = []

        for j in range(y_train.shape[1]):

            dist.append(spatial.distance.euclidean(y_test[:,i],y_train[:,j]))

        min_idx=np.argmin(dist)

        #print('dist = ', dist); print('min_idx = ',min_idx, ', dist[min_idx] = ', dist[min_idx])

        #print(int(min_idx/21))

        if int(min_idx/21) == i:

            acc[i] += 1

    #print('accuracy per class = ', acc)

    print('PCA accuracy for k = ', k, ' = ', np.sum(acc)/630)

    return acc

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

def test_pca():

    for k in range(20):

        calc_pca_accuracy(k)
```

Rehana Mahfuz (rmahfuz@purdue.edu)
ECE 661 HW10

```
#===LDA=================================================================================
===========================

def process_train_lda(k): #k is num of components. returns y_train(k x num_samples(630)), W_k
  img_arr = []
  for fn in os.listdir(path + 'ECE661_2018_hw10_DB1/train'):
    img=cv2.imread(path + 'ECE661_2018_hw10_DB1/train/' + fn)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)#; img=np.array(img, dtype=np.float32)
    img=img.flatten()
    img=img/np.linalg.norm(img) #normalize to make it illumination-invariant
    img_arr.append(img)

  img_arr = np.array(img_arr,dtype=np.float32) #each row represents an image (630x16384)
  mean_img = np.mean(img_arr,axis=0) #mean of each column over all rows (1x16384)
  X=(img_arr-mean_img).T#(
  #Finding mean of each class
  class_means = []
  for i in range(num_classes):
    class_means.append(np.mean(img_arr[i*num_samples:i*num_samples+num_samples-1], axis=0))
  class_means = np.array(class_means, dtype = np.float32) #(30x16384)
  M=[]
  for i in range(num_classes):
    M.append(class_means[i,:] - mean_img)
  M=np.array(M, dtype=np.float32); M=M.T #(16384,30)
  w,u=np.linalg.eig(np.matmul(M.T,M))
  u=u.T

  u=u[np.argsort(w)[::-1]] #sorting in descending order
  #w.sort(); w=w[::-1]
  u=u.T #each column is an eigenvector
```

```
V=np.matmul(M,u)

#Normalize each column of V:

for i in range(V.shape[1]):

    V[:,i] = V[:,i]/np.linalg.norm(V[:,i])

#Extract the largest/first k columns into Y

Y = V[:,:k]

#print('Y.shape = ', Y.shape) #(16384,k)

#Finding D_B:

fac=np.matmul(Y.T,M); D_B = np.matmul(fac, fac.T)

#Finding Z:

#tmp=np.linalg.matrix_power(D_B,-1);

tmp = fractional_matrix_power(D_B, -0.5); Z=np.matmul(Y, tmp)

#Finding eigenvectors of Z.T*S_w*Z:

X_w = []

for i in range(630):

    X_w.append(img_arr[i] - class_means[int(i/21)])

X_w = np.array(X_w, dtype=np.float32).T

#print('X_w.shape = ', X_w.shape)#(16384,630)

fac = np.matmul(Z.T, X_w); S_BW = np.matmul(fac.T, fac)

w,u = np.linalg.eig(S_BW)

u=u.T

u=u[np.argsort(w)] #sorting in ascending order

#w.sort();

u=u.T #each column is an eigenvector

U=np.matmul(S_BW,u)

#Normalize each column of U:

for i in range(U.shape[1]):

    U[:,i] = U[:,i]/np.linalg.norm(U[:,i])

#Extract the smallest/first k columns into U_hat
```

```
    U_hat = U[:,:k]

    #print('U_hat.shape = ', U_hat.shape)#(630,k)

    #Finally finding W_p:

    W_k = np.matmul(U_hat.T, Z.T).T

    y_train = np.matmul(W_k.T,X)#(k,630)

    return (y_train, W_k)

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

def calc_lda_accuracy(k):

    (y_train, W_k) = process_train_lda(k) #(k,630)


    img_arr = []

    for fn in os.listdir(path + 'ECE661_2018_hw10_DB1/test'):

        img=cv2.imread(path + 'ECE661_2018_hw10_DB1/test/' + fn)

        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)#; img=np.array(img, dtype=np.float32)

        img=img.flatten()

        img=img/np.linalg.norm(img) #normalize to make it illumination-invariant

        img_arr.append(img)

    img_arr = np.array(img_arr,dtype=np.float32)

    mean_img = np.mean(img_arr,axis=0) #mean of each column over all rows

    X=(img_arr-mean_img).T#(16384,630)

    for i in range(X.shape[1]):

        X[:,i] = X[:,i]/np.linalg.norm(X[:,i])

    y_test = np.matmul(W_k.T,X)

    #print('y_test.shape = ', y_test.shape) #(k,630)

    #-------------------Nearest neighbors classification---------------------------

    #Now I have both y_test and y_train. For each column in y_test, I will try to match it with the nearest
column in y_train.

    acc = [0]*num_classes

    for i in range(y_test.shape[1]):
```

```
        dist = []
        for j in range(y_train.shape[1]):
            dist.append(spatial.distance.euclidean(y_test[:,i],y_train[:,j]))
        min_idx=np.argmin(dist)
        #print('dist = ', dist); print('min_idx = ',min_idx, ', dist[min_idx] = ', dist[min_idx])
        #print(int(min_idx/21))
        if int(min_idx/21) == i:
            acc[i] += 1
    #print('accuracy per class = ', acc)
    print('overall accuracy for k = ', k, ' = ', np.sum(acc)/630)
    return acc
#--------------------------------------------------------------------------------------------------------------
def test_lda():
    #process_train_lda(15)
    #calc_lda_accuracy(15)
    for k in range(20):
        calc_lda_accuracy(k)
#================================================================================================================
def main():
    test_pca()
    #test_lda()
#--------------------------------------------------------------------------------------------------------------
if __name__ == '__main__':
    main()
```

Part 2: Object Detection with Cascaded AdaBoost Classification

Haar Feature Extraction

First, an integral representation of the image is obtained:

I(x,y) = $\sum_{x_i \le x, y_i \le y} i(x_i, y_i)$

Rehana Mahfuz (rmahfuz@purdue.edu)
ECE 661 HW10

To generate features, we use Haar-like filters. For example, [0,1] is a filter which can be slid over the entire image, a feature being generated at every position where it is applied to the image. [0,1] means that the element corresponding to 0 will be subtracted from the element corresponding to 1. Since the image size is 20 x 40 pixels, the number of features that can be generated using just this one filter of this size 1 x 2 is 20x39. Similarly, using a filter [0,0,1,1], 20 x 37 filters can be generated. The horizontal features used were of sizes 1 x 2, 1 x 4, …, 1 x 40. The vertical features used were of sizes 2 x 2, 4 x 2, …, 20 x 2. The total number of features generated was 11,900.

In summary, a feature is characterized by the filter and the position of the image where it is applied.

Building a Weak Classifier

A weak classifier is characterized by a feature, a threshold, and a polarity. As mentioned earlier, a feature is a filter applied to a specific location of the image. The output of a feature is a value. After applying the feature to each sample in a dataset, if we apply a threshold to the output, we can divide all the data samples into two parts. If all the samples which have feature values above the threshold are considered to be classified as positive, we set the polarity to +1. If all the samples which have feature values above the threshold are considered to be classified as negative, we set the polarity to -1. Training a weak classifier means that given a particular feature, we strategically set a threshold value to obtain a good classification of the training images, and according set the polarity.

The feature is denoted as f, the polarity as p, and the threshold as theta. A classifier is represented by h(x, f, p, theta), where x is the data sample.

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

The error of a classifier is

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

where T^+ is the total sum of positive example weights, T^- is the total sum of negative example weights, S^+ is the sum of positive weights below this threshold, and S^- is the sum of negative weights below this threshold. We experiment by placing the threshold between different data samples to determine the best position of the threshold. We sort the data samples by output value of the feature before doing this placement.

Building a strong Adaptive Boosted "AdaBoost" strong classifier

Out of the many classifiers we have, we combine the best T weak classifiers h_t, where 1<=t<=T to form a strong classifier

h(x) = $\begin{cases} 1, if\ \sum_{t=1}^{T} alpha_t * h_t(x) \geq\ 0.5 * \sum_{t=1}^{T} \alpha_t \\ 0, \qquad\qquad\qquad otherwise \end{cases}$

, where α_t will be described below. We choose T = 100. To find this strong classifier, we do the following:

First, a weight vector is initialized such that every training example has a weight associated with it. The weight vector is initialized such that the weight corresponding to a positive training sample is

1/(2*number of positive examples), and the weight corresponding to a negative training sample is 1/(2*number of negative examples).

$W_i = \frac{1}{2m}, \frac{1}{2l}$, for y_i = 0, 1 respectively, where m is the number of negative training samples, and l is the number of positive training samples. y_i is the label corresponding to the i^{th} training sample, which is either 0 or 1.

For t = 1,…,T:

- Normalize the weights
  $w_i = \frac{w_i}{\sum_{j=1}^{n} w_j}$
- For each feature j, train a classifier h_j. The error is $epsilon_j = \sum_i \; w_i |h_j(x_i) - y_i|$. Find the classifier h_t with the least error.
- Update the weights
  $W_i = w_i * \beta_t^{1-e_i}$, where e_i = 1 if example x_i is classified incorrectly, and e_i = 0 otherwise.
  $\beta_t = \frac{epsilon_t}{1-epsilon_t}$, $\alpha_t = \log(1/\beta_t)$

As mentioned earlier, the strong classifier is

$$h(x) = \begin{cases} 1, if \;\; \sum_{t=1}^{T} alpha_t * h_t(x) \geq \; 0.5 * \sum_{t=1}^{T} \alpha_t \\ 0, \qquad\qquad\qquad\qquad\qquad otherwise \end{cases}$$

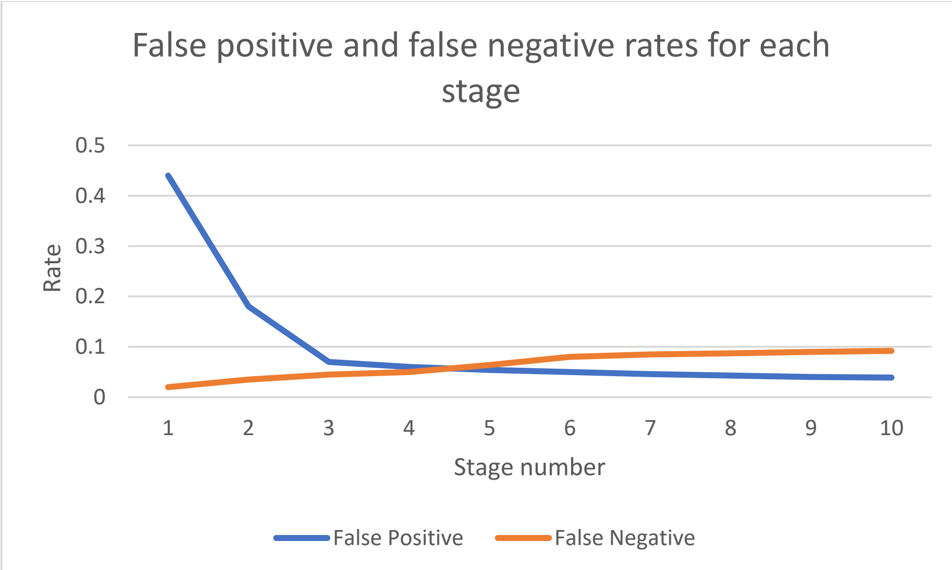
The threshold which is set to $0.5 * \sum_{t=1}^{T} \alpha_t$ in the above equation is actually set to the minimum value of $\sum_{t=1}^{T} \alpha_t \, h_t(x)$ while training, since we want the classifier to pass positive examples during training.

<u>Cascading Strong Classifiers</u>

We cascade S or lesser strong classifiers for our object detection purposes. We choose S = 10. Only examples classified as positive by one stage are passed on to the next stage for classification. To generate each strong classifier, we stop adding more weak classifiers if the false positive rate reaches 0.5. We stop generating more strong classifiers if almost all negative samples are correctly classified.

Rehana Mahfuz (rmahfuz@purdue.edu)
ECE 661 HW10

Results



Source Code

```
import numpy as np

import cv2, os

from scipy import spatial


#path = '../Users/rmahfuz/Desktop/661/HW10/'

path = ''

num_max_strong=10 #maximum number of strong classifiers permitted

num_max_weak=100 #maximum number of weak classifiers permitted

thresh_positive = 1 #acceptable positive detection rate

thresh_falsePositive = 0.5 #acceptable false positive rate

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

def find_features(filePath, saveName):

    def findRec(intImg, corner):

        one = intImg[int(corner[0,0]), int(corner[0,1])]

        two = intImg[int(corner[1,0]),int(corner[1,1])]

        three = intImg[int(corner[2,0]), int(corner[2,1])]
```

```
        four = intImg[int(corner[3,0]), int(corner[3,1])]

        return four+one-two-three

    feature=[]

    h_size=np.linspace(2,20,10, dtype=np.int32)

    v_size=np.linspace(2,40,20, dtype=np.int32)

    for fn in os.listdir(path+filePath):

        #print(fn)

        img=cv2.imread(path+filePath+fn)

        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        #Finding integral representation of image:

        intImg = np.zeros((img.shape[0]+1, img.shape[1]+1))

        intImg[1:,1:] = np.cumsum(np.cumsum(img,axis=0),axis=1)

        feature_temp = []

        #Find horizontal feature:

        for j in range(len(h_size)):

            width=h_size[j]

            for k in range(img.shape[0]):

                for m in range(img.shape[1]-int(width)+1):

                    corner0 = np.array([[k,m],[k,m+width/2],

                                        [k+1,m],[k+1,m+width/2]])

                    corner1 = np.array([[k,m+width/2],[k,m+width],

                                        [k+1,m+width/2],[k+1,m+width]])

                    rec0=findRec(intImg, corner0)

                    rec1=findRec(intImg, corner1)

                    feature_temp.append(rec1-rec0)

        #Find vertical feature:

        for j in range(len(v_size)):

            height=v_size[j]

            for k in range(img.shape[0]-int(height)+1):
```

```
            for m in range(img.shape[1]-1):
                corner1 = np.array([[k,m],[k,m+2],
                              [k+height/2,m],[k+height/2,m+2]])
                corner0 = np.array([[k+height/2,m],[k+height/2,m+2],
                              [k+height,m],[k+height,m+2]])
                rec1=findRec(intImg, corner1)
                rec0=findRec(intImg, corner0)
                feature_temp.append(rec1-rec0)
        feature.append(feature_temp)
    feature = np.array(feature).T
    np.save(path + saveName, feature)#write feature into file
    print('feature.shape = ', feature.shape)
    return feature
#===============================================================================
def adaBoost(feature_all, num_pos, current_idx, stage, num_max_weak):
    def find_bestWeak(feature, weights, labels, num_pos):
        bestWeak = dict()
        (num_features, num_img) = feature.shape
        T_plus = np.repeat(np.sum(weights[:num_pos]), num_img)
        T_minus = np.repeat(np.sum(weights[num_pos:]), num_img)
        bestWeak['min_err'] = np.inf
        for i in range(num_features):
            current_feature = feature[i,:]
            sorted_features = np.sort(current_feature)
            sorted_feature_idx = np.argsort(current_feature)
            sorted_weights = weights[sorted_feature_idx]
            sorted_labels = labels[sorted_feature_idx]
            S_plus = np.cumsum(sorted_weights*sorted_labels)
            S_minus = np.cumsum(sorted_weights) - S_plus
```

```
        error1 = S_plus + (T_minus - S_minus)

        error2 = S_minus + (T_plus - S_plus)

        #print('len(error1) = {}, len(error2) = {}'.format(len(error1), len(error2)))

        e = []

        for j in range(len(error1)):

            e.append(min(error1[j], error2[j]))

        #e = np.min(error1, error2) #finding the error

        min_error = np.min(e) #finding best threshold

        thresh = np.argmin(e)

        polarity = -1 if error1[thresh] <= error2[thresh] else 1

        #obtain classification result

        classification_result = np.zeros((num_img, 1))

        if polarity == -1:

            classification_result[thresh:] = 1

        else:

            classification_result[:thresh] = 1

        classification_result[sorted_feature_idx] = classification_result


        if min_error < bestWeak['min_err']:

            bestWeak['min_err'] = min_error

            bestWeak['polarity'] = polarity

            bestWeak['feature'] = i

            bestWeak['result'] = classification_result

            if thresh == 0: #a little smaller than the smallest

                bestWeak['thresh'] = sorted_features[thresh] - 0.001

            elif thresh == len(sorted_features): #a little larger than the largest

                bestWeak['thresh'] = sorted_features[thresh] + 0.001

            else: #between that feature value and the previous feature value

                bestWeak['thresh'] = 0.5*(sorted_features[thresh] + sorted_features[thresh-1])
```

Rehana Mahfuz (rmahfuz@purdue.edu)


```
        return bestWeak

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

    feature = feature_all[:, current_idx]

    num_neg = len(current_idx) - num_pos

    #Initializing weights and labels:

    weights = [0.5*(1.0/num_pos)]*num_pos

    weights.extend([0.5*(1.0/num_neg)]*num_neg)

    labels = [1]*num_pos; labels.extend([0]*num_neg); labels=np.array(labels)


    alpha = np.zeros((num_max_weak, 1))

    weak = np.zeros((4, num_max_weak))

    weak_result = np.zeros((len(current_idx), num_max_weak)) #(feature, thresh, polarity, alpha)

    strong_result = np.zeros((len(current_idx),1))

    positive_accuracy = [] #of strong classifier at the end of each stage

    negative_FP = [] #of strong classifier at the end of each stage


    for t in range(num_max_weak):

            #print('weights = {}, np.sum(weights) = {}'.format(weights, np.sum(weights)))

        weights = weights/np.sum(weights) #normalizing the weights

        best_weak = find_bestWeak(feature, weights, labels, num_pos)

        err = best_weak['min_err']

        weak[:3,t] = [best_weak['feature'], best_weak['thresh'], best_weak['polarity']]

        weak_result[:,t] = best_weak['result'].flatten()

        #compute beta

        beta = err/(1-err)

        alpha[t,0] = np.log(1/beta)

        weak[3,t] = alpha[t,0]

        #update weights

        e = []
```

```
        for j in range(len(weak_result)):
            e.append(int(weak_result[j,t] == labels[j]))
        e = np.array(e)
        #e = np.array([weak_result==labels], dtype = np.int32)
        print('e = {}'.format(e))
        for i in range(len(weights)):
            weights[i] = weights[i]*pow(beta,1-e[i])
        #compute strong classifier result
        strong_tmp = np.matmul(weak_result[:,:t], alpha[:t,0])
        thresh = np.min(strong_tmp[:num_pos])
        for i in range(len(current_idx)):
            strong_result[i] = 1 if strong_tmp[i] >=thresh else 0
        positive_accuracy.append(np.sum(strong_result[:num_pos])/num_pos)
        negative_FP.append(np.sum(strong_result[num_pos:])/num_neg)
        if positive_accuracy[t] >= thresh_positive and negative_FP[t] <= thresh_falsePositive:
            break
    strong = dict()
    strong['updated_idx'] = np.arange(num_pos, dtype = np.int32).tolist()
    remaining = np.nonzero(strong_result[num_pos:])[0]+num_pos
    strong['updated_idx'].extend(remaining.tolist()) #the images classified as positive
    strong['num_weak'] = t #number of weak classifiers
    strong['parameters'] = weak #collection of weak classifiers
    return strong
#======================================================================
def train():
    feature_pos = np.load(path + 'train_pos.npy')
    feature_neg = np.load(path + 'train_neg.npy')
    feature_all = np.hstack((feature_pos, feature_neg))
    num_pos = feature_pos.shape[1]
```

```
    num_neg = feature_neg.shape[1]
    train_result = []
    current_idx = np.arange(num_pos+num_neg)
    for i in range(num_max_strong):
        print('starting stage {}'.format(i))
        strong = adaBoost(feature_all, num_pos, current_idx, i, num_max_weak)
        current_idx = strong['updated_idx']
        neg_idx = []
        for j in range(len(current_idx)):
            if current_idx[j] > num_pos:
                neg_idx.append(j)
        train_result.append(strong)
        if len(neg_idx) == 0:
            break
        num_pos = len(current_idx) - len(neg_idx)
    np.save(path + 'training_result.npy', np.array(train_result))
#==============================================================================
def test():
    def strong_predict(feature_pos, feature_neg, feature_idx, thresh, polarity, alpha, num_weak):
            feature_all = np.hstack((feature_pos, feature_neg))
            num_pos = feature_pos.shape[1]
            num_neg = feature_neg.shape[1]
            num_img = num_pos + num_neg
            #calculating weak classifier result
            weak_result = np.zeros((num_img, num_weak))
            for i in range(num_weak):
                current_feature = feature_all[int(feature_idx[i]),:]
                for j in range(num_img):
                        if polarity[i]*current_feature[j] <= polarity[i]*thresh[i]:
```

```
                weak_result[j,i] = 1 #otherwise zero by default
        #calculating strong classifier result
        strong_result = np.zeros((num_img, 1))
        strong_tmp = np.matmul(weak_result, alpha.T)
        strong_thresh = 0.5*np.sum(alpha)
        for i in range(num_img):
            if strong_tmp[i] >= strong_thresh:
                strong_result[i] = 1
        return strong_result
#----------------------------------------------------------------------
feature_pos = np.load(path + 'test_pos.npy')
feature_neg = np.load(path + 'test_neg.npy')
train_result = np.load(path + 'training_result.npy')
num_test_pos = feature_pos.shape[1]
num_test_neg = feature_neg.shape[1]
num_stages = len(train_result)
false_positive = 0; true_negative = 0
fp = np.zeros((num_stages, 1)); fn = np.zeros((num_stages, 1))
for i in range(num_stages):
    current_stage = train_result[i]
    num_weak = current_stage['num_weak']
    weak = current_stage['parameters'] #collection of weak classifiers
    feature_idx = weak[0,:num_weak]
    weak_thresh = weak[1,:num_weak]
    polarity = weak[2,:num_weak]
    alpha = weak[3,:num_weak]
    predicted_labels = strong_predict(feature_pos, feature_neg, feature_idx,
                            weak_thresh, polarity, alpha, num_weak)
    num_pos = feature_pos.shape[1]
```

```
        num_neg = feature_neg.shape[1]
        #calculating false positive and false negative for this stage
            #print(predicted_labels)
            if len(np.nonzero(predicted_labels[:num_pos])[0]) == 0:
              false_positive += 0
            else:
              false_positive += num_pos-len(np.nonzero(predicted_labels[:num_pos])[0])
            if len(np.nonzero(predicted_labels[num_pos:])[0]) == 0:
              true_negative += 0
            else:
              true_negative += num_neg - len(np.nonzero(predicted_labels[num_pos:])[0])
        fp[i] = (num_test_neg-true_negative)/num_test_neg #misclassified negative
        fn[i] = false_positive/num_test_pos #misclassified positive
        #update features
        feature_pos = feature_pos[:,np.nonzero(predicted_labels[:num_pos])[0]]
        feature_neg = feature_neg[:,np.nonzero(predicted_labels[num_pos:])[0]]
    print('fp = ', fp)
    print('fn = ', fn)
#=====================================================================
def main():
    '''find_features('ECE661_2018_hw10_DB2/train/positive/', 'train_pos.npy')
    find_features('ECE661_2018_hw10_DB2/train/negative/', 'train_neg.npy')
    find_features('ECE661_2018_hw10_DB2/test/positive/', 'test_pos.npy')
    find_features('ECE661_2018_hw10_DB2/test/negative/', 'test_neg.npy')'''
    #train()
    test()


if __name__ == '__main__':
    main()
```

Rehana Mahfuz (rmahfuz@purdue.edu)
ECE 661 HW10