**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 xi 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)\*

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

X = [x1-m x2-m … xN-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)\* . The eigenvectors corresponding to the largest k eigenvalues of C form the PCA feature set, represented as WK, with each column being an eigenvector. k is the number of principal components we choose to retain. A vectorized image xi’s projection into this eigenspace can be computed as yi = WKT (xi – 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 XTX 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 Wk, 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) =

Where SB is the between-class covariance matrix computed as SB = (1/NC)\*, where NC is the number of classes in the training set, mj­ is the mean of class j and m is the mean of all classes.

SW is the within-class covariance matrix computed as SW = (1/NC)\*,

Where Nj is the number of images in class j, and xij is the ith image of class j.

Since it is likely that SW 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 mk is calculated as mk = 1/||CK||\*.

We find the eigendecomposition of SB 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 DB, the upper-left k x k submatrix of the diagonalized eigenvalues of SB:

DB = (YTM)(YTM)T

Then we calculate Z = YDB-0.5

We use the same computational trick again to find the eigendecomposition of ZTSWZ = (ZTXW)(ZTXW)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 WPT = UTZT. The projection yi­ of any vectorized image xi can be calculated as yi = WKT (xi – 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

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)

#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)

#===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) =

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) =

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 ht, where 1<=t<=T to form a strong classifier

h(x) =

, where αt 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).

Wi  = , , for yi = 0, 1 respectively, where m is the number of negative training samples, and l is the number of positive training samples. yi is the label corresponding to the ith training sample, which is either 0 or 1.

For t = 1,…,T:

* Normalize the weights

wi =

* For each feature j, train a classifier hj. The error is epsilonj = . Find the classifier ht with the least error.
* Update the weights

Wi = wi\*βt1-e\_i, where ei = 1 if example xi is classified incorrectly, and ei = 0 otherwise.

βt = , αt = log(1/ βt)

As mentioned earlier, the strong classifier is

h(x) =

The threshold which is set to in the above equation is actually set to the minimum value of while training, since we want the classifier to pass positive examples during training.

Cascading Strong Classifiers

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.

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])

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()