ECE661: Homework 10

Wan-Eih Huang

November 30, 2018

1 Face Recognition

1.1 Principal Components Analysis (PCA)

PCA tries to find a set of orthogonal directions with maximal variance for all training data.

1.1.1 Estimate Covariance Matrix

If we are given an image set which consists of N images, then we can estimate the covariance matrix C of the image set by

$$C = \mathbf{X}\mathbf{X}^T$$
$$\text{where } \mathbf{X} = [\vec{\mathbf{x}}_0 - \bar{\mathbf{m}} | \vec{\mathbf{x}}_1 - \bar{\mathbf{m}} | \cdots | \vec{\mathbf{x}}_{N-1} - \bar{\mathbf{m}}]$$
$$\bar{\mathbf{m}} = \frac{1}{N} \sum_{i=0}^{N-1} \vec{\mathbf{x}}_i$$

$\vec{\mathbf{x}}_i$ are the vectorized images. In addition, we need to normalize $\mathbf{X}$ to achieve illumination invariant. That is, $\vec{\mathbf{x}}_i^T \vec{\mathbf{x}}_i = 1$.

1.1.2 Find Feature Set

To reduce the dimensionality by PCA, the feature set is formed by the eigenvectors $\vec{\mathbf{w}}_i$ of C corresponding to the K largest eigenvalues. We denotes it by $\mathbf{W}_K = [\vec{\mathbf{w}}_0 | \vec{\mathbf{w}}_1 | \cdots | \vec{\mathbf{w}}_{K-1}]$.
However, eigendecomposition of C has very high computational complexity and it also causes the numerical instability. Therefore, the computational trick is doing eigendecomposition on $\mathbf{X}^T\mathbf{X}$. The derivations as follows.

$$\mathbf{X}\mathbf{X}^T \vec{\mathbf{w}} = \lambda \vec{\mathbf{w}}$$
$$\mathbf{X}^T\mathbf{X} \vec{\mathbf{u}} = \lambda \vec{\mathbf{u}}$$
$$\mathbf{X}\mathbf{X}^T\mathbf{X} \vec{\mathbf{u}} = \lambda \mathbf{X} \vec{\mathbf{u}}$$
$$C(\mathbf{X}\vec{\mathbf{u}}) = \lambda(\mathbf{X}\vec{\mathbf{u}})$$

Hence, the eigenvectors $\vec{\mathbf{w}}$ is given by

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

Because this trick will not give us unity vectors, we need to normalize them before projection.

1.1.3 Training and Testing

Then the feature values of a given image is the projection on the subspace span by these eigenvectors. In other words, the feature values are given by

$$\vec{\mathbf{y}} = \mathbf{W}_K^T(\vec{\mathbf{x}} - \bar{\mathbf{m}})$$

Train a KNN model by the feature values of training dataset and test it by the feature values of testing dataset.

1.2 Linear Discriminant Analysis (LDA)

LDA tries to find the directions with maximal discriminate between the classes. That is, it wants to find the directions that can maximize the between-class scatter and minimize the within-class scatter.
Between-class scatter is defined by

$$S_B = \frac{1}{|C|}\sum_{i=1}^{|C|}(\vec{\mathbf{m}}_i - \bar{\mathbf{m}})(\vec{\mathbf{m}}_i - \bar{\mathbf{m}})^T$$

where $\vec{\mathbf{m}}_i$ is the class mean and $\bar{\mathbf{m}}$ is the global mean. $|C|$ is the number of classes.
Within-class scatter is defined by

$$S_W = \frac{1}{|C|}\sum_{i=1}^{|C|}\frac{1}{|C_i|}\sum_{k=1}^{|C_i|}(\vec{\mathbf{x}}_k^i - \vec{\mathbf{m}}_i)(\vec{\mathbf{x}}_k^i - \vec{\mathbf{m}}_i)^T$$

where $\vec{\mathbf{x}}_k^i$ is the kth image in the ith class and $|C_i|$ is the number of images in the ith class.
The Fisher discriminant function is defined by

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

1.2.1 Between-Class Scatter

Assume $S_B = \mathbf{X_B}\mathbf{X_B}^T$. We play the same trick as that in PCA. Use eigendecomposition on $\mathbf{X_B}^T\mathbf{X_B}$, then $\mathbf{V} = \mathbf{X_B}\mathbf{U}$ contains the eigenvectors of S_B, where $\mathbf{U}$ is the matrix of eigenvectors of $\mathbf{X_B}^T\mathbf{X_B}$.
Then we can obtain $\mathbf{Y}$ by retaining only M eigenvectors corresponding to M largest eigenvalues. Hence, $\mathbf{Y}^T S_B \mathbf{Y} = \mathbf{D_B}$, where $\mathbf{D_B}$ is the diagonal matrix with M largest eigenvalues.

1.2.2 Within-Class Scatter

Construct $\mathbf{Z} = \mathbf{Y}\mathbf{D_B}^{-\frac{1}{2}}$ and apply eigendecomposition on $\mathbf{Z}^T S_W \mathbf{Z}$. Then the eigenvectors for maximizing the discriminant function is given by

$$\vec{\mathbf{w}} = \mathbf{Z}\vec{\mathbf{u}}_W$$

where $\vec{\mathbf{u}}_W$ is the eigenvectors obtained from eigendecomposition of $\mathbf{Z}^T S_W \mathbf{Z}$. Retaining only K eigenvectors to form $\mathbf{W}_K$.

1.2.3 Training and Testing

The training and testing procedures are the same as what we did in PCA.

1.3 Performance Evaluation

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

1.4 Result

Our result of PCA and LDA shown in Figure 1 and Table 1. Only when the number of eigenvectors is 1 and 2, PCA performs better than LDA. However, LDA converges faster than PCA. LDA achieves 100% when number of vectors is 7, but PCA achieves it as number of vectors is 12.

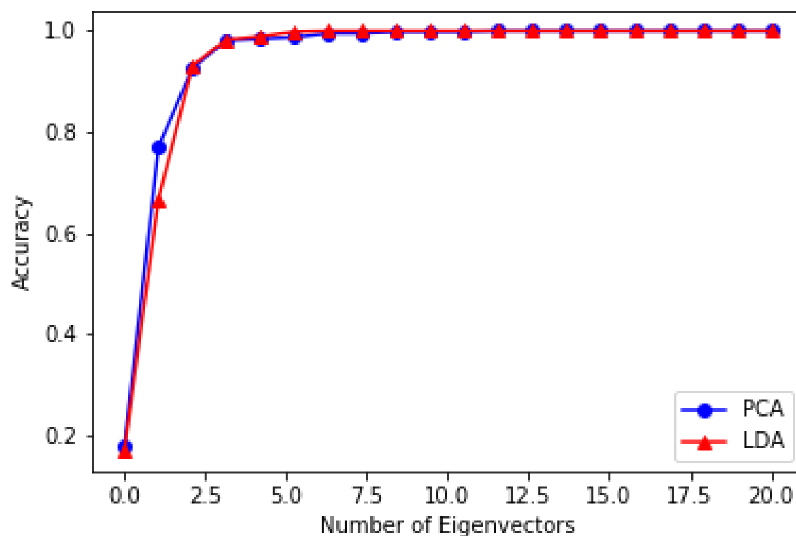

Figure 1: The comparison of accuracy between PCA and LDA

Number of Eigenvectors	PCA Accuracy	LDA Accuracy
1	0.1777	0.1682
2	0.7698	0.6666
3	0.9269	0.9317
4	0.9809	0.9825
5	0.9841	0.9888
6	0.9873	0.9984
7	0.9952	1
8	0.9952	1
9	0.9984	1
10	0.9984	1
11	0.9984	1
12	1	1
13	1	1
14	1	1
15	1	1
16	1	1
17	1	1
18	1	1
19	1	1
20	1	1

Table 1: Accuracy of PCA and LDA w.r.t. number of eigenvectors.

2 Object Detection with Cascaded AdaBoost Classification

In this homework, we are going to use Viola and Jones algorithm to construct an object detector which cascades multiple strong classifiers built by AdaBoost algorithm.

2.1 Feature Extraction

The one of the advantages of AdaBoost is that we do not need to define very strong features to make the classification result better. It aggregates the weak classifiers to form a strong classifier. To achieve this, we need to have a large number of features. However, they can be simple. So, in this homework we adopt Haar filter with different size and orientations to extract many features for training AdaBoost classifier.
For horizontal direction, we use 1×2, 1×4, $\cdots$, 1×40. And 2×2, 4×2, $\cdots$, 20×2 for vertical direction. Therefore, the total number of features is 11900. Note that the size of the images in database is 20×40. Moreover, we use method of integral image to reduce computational cost.

2.2 Training

2.2.1 Construct A Strong Classifier

Step 1: Initialize the weights of the samples with equal distribution. That is, $\frac{1}{2m}$ and $\frac{1}{2l}$ for negative and positive samples respectively, where m and l are the total number of negative and positive samples respectively.
Step 2: Normalize the weights and construct an ordered list of all the training samples w.r.t. each feature. Then we can find the minimum error based on the following equation.

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

T^+ is the total sum of positive sample weights, and T^- is the total sum of negative sample weights. S^+ is the sum of positive sample weights below current sample. Similarly, S^- is the sum of negative sample weights below the current sample.
Once we found the minimum error ϵ_t, we can define the best weak classifier $h_t(x) = h(x, f_t, p_t, \theta_t)$ for current iteration t. f_t is the feature, p_t is the polarity, and θ_t is the threshold value that minimize the error.
Step 3: Update the weights for the next iteration by

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

where $e_i = 0$ if sample x_i is classified correctly; otherwise, $e_i = 1$. $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$.
Step 4: Last, check whether the aggregated weak classifiers achieves criteria. That is, the true detection rate is 1 and the false positive rate is 0.5. If it is not, then we go back to **Step 2** for finding next classifier.
The final strong classifier is

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

where $\alpha_t = \log\frac{1}{\beta_t}$. To achieve true detection rate is 1, we set threshold is the minimum value of positive samples only in training process.

2.2.2 Cascade Strong Classifiers

Step 1: Construct a strong classifier by AdaBoost as we described in last section.
Step 2: Check whether the accumulated false positive rate is zero. If it is not, then we only select the misclassified negative samples with all positive samples to the next run (**Step 1**).

2.3 Testing

Step 1: Classify samples by a single strong classifier.

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

Step 2: Choose next strong classifier until the data processed by all the strong classifiers which are cascaded as our object detector.

2.4 Performance Evaluation

False positive rate:

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

False negative rate:

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

2.5 Result

Training Result:

Stage	# of Weak Classifiers	# of Negative Images	# of Misclassified Negative Images	Stage FP Rate
1	8	1758	791	0.4499
2	16	791	336	0.4248
3	21	336	140	0.4167
4	21	140	67	0.4786
5	19	67	31	0.4627
6	11	31	14	0.4516
7	10	14	4	0.2857
8	5	4	0	0

Table 2: The false positive rate of each stage.

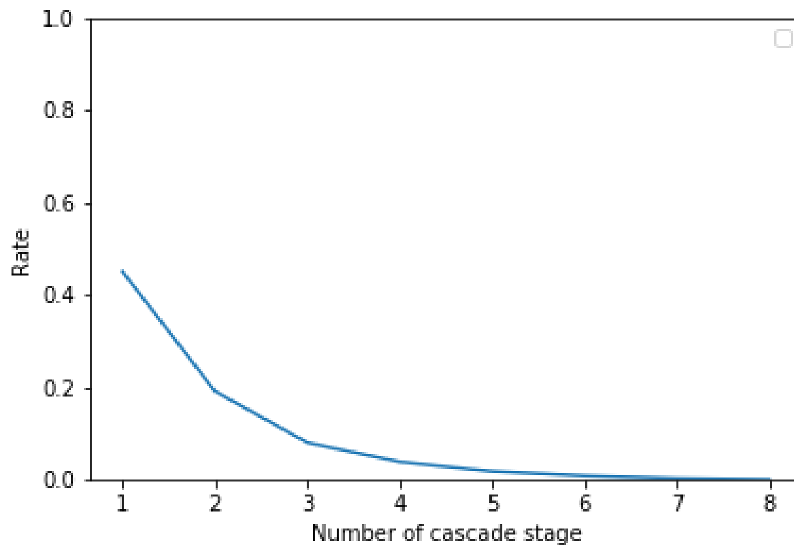

Figure 2: The accumulated false positive rate of each stage in train dataset.

Testing Result: The false positive rate decreases as the number of cascade stage increases. The final false positive rate is 0.0022. In addition, the false negative rate increases as the number of cascade stage increases. And the final false negative rate is 0.3932.

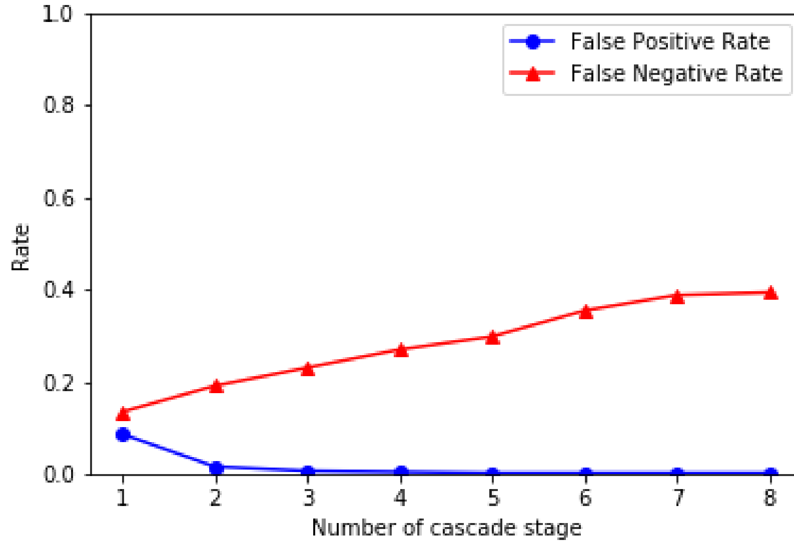

Figure 3: The accumulated false positive rate and false negative rate of each stage in test dataset.

Stage	# of Weak Classifiers	FP Rate	FN Rate
1	8	0.0863	0.1348
2	16	0.0159	0.1910
3	21	0.0068	0.2303
4	21	0.0045	0.2696
5	19	0.0022	0.2977
6	11	0.0022	0.3539
7	10	0.0022	0.3876
8	5	0.0022	0.3932

Table 3: The accumulated false positive rate and false negative rate of each stage in test dataset.

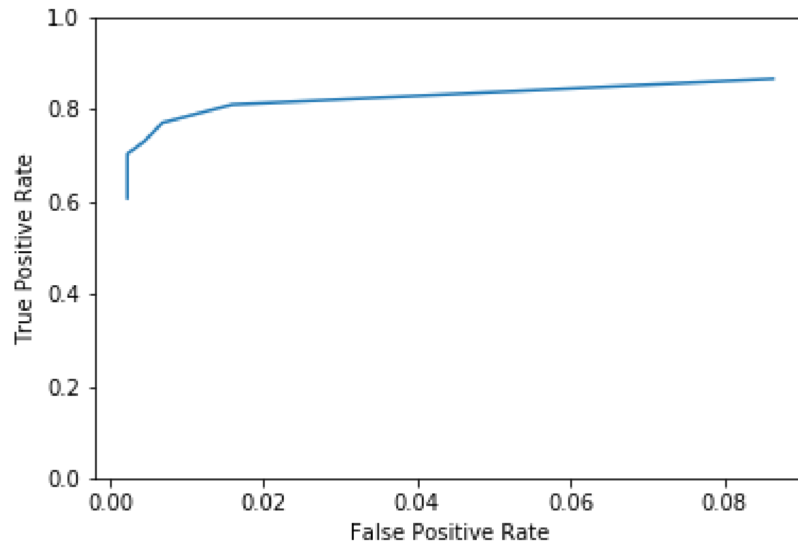

Figure 4: The ROC curve of test dataset.

3 Source Code

3.1 Part 1: PCA

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Nov 20 21:53:01 2018
PCA+knn for face recognition
@author: wehuang
"""
import os
import numpy as np
import cv2
from sklearn.neighbors import KNeighborsClassifier

nClasses = 30
nSamples = 21

def loadData(mypath):
    files = os.listdir(mypath)
    files.sort()
    imgVec = []
    for i in range(len(files)):
        img = cv2.imread(mypath+files[i])
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        imgVec.append(gray.reshape((1,-1)))
    imgVec = np.asarray(imgVec).reshape((len(files),-1))
    imgVec = imgVec.transpose()
    imgVec = imgVec/np.linalg.norm(imgVec, axis=0)
    meanG = np.mean(imgVec, axis=1)
    imgVecNorm = imgVec-meanG[:,None]

    return imgVecNorm

mypath = 'ECE661_2018_hw10_DB1/train/'
imgVecNorm = loadData(mypath)

mypath_t = 'ECE661_2018_hw10_DB1/test/'
imgVecNorm_t = loadData(mypath_t)

labels = []
for c in range(nClasses):
    tmp = np.ones((nSamples, 1))
    labels.extend(tmp[:,0]*(c+1))
labels = np.asarray(labels)

#PCA
d, u = np.linalg.eig(imgVecNorm.transpose().dot(imgVecNorm))#
    630x630
idx = np.argsort(-1*d)
u = u[:,idx]
w = imgVecNorm.dot(u)
w = w/np.linalg.norm(w, axis=0)
```

```
score = []
c = np.zeros((len(labels), 1))
for K in range(30):
    subSpace = w[:,:K+1]
    feature_train = np.dot(subSpace.transpose(), imgVecNorm)
    feature_test = np.dot(subSpace.transpose(), imgVecNorm_t)
    #KNN
    classifier = KNeighborsClassifier(n_neighbors=1)
    classifier.fit(feature_train.transpose(), labels) #[
        nSamples, nFeatures]
    pred = classifier.predict(feature_test.transpose())
    c[pred == labels] = 1
    score += [np.sum(c)/len(labels)]
```

3.2 Part 1: LDA

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Nov 21 13:10:12 2018
LDA+knn for face recognition
@author: wehuang
"""
import os
import numpy as np
import cv2
from sklearn.neighbors import KNeighborsClassifier

nClasses = 30
nSamples = 21

def loadData(mypath):
    files = os.listdir(mypath)
    files.sort()
    imgVec = []
    for i in range(len(files)):
        img = cv2.imread(mypath+files[i])
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        imgVec.append(gray.reshape((1,-1)))
    imgVec = np.asarray(imgVec).reshape((len(files),-1))
    imgVec = imgVec.transpose()
    imgVec = imgVec/np.linalg.norm(imgVec, axis=0)
    meanG = np.mean(imgVec, axis=1)

    return imgVec, meanG

mypath = 'ECE661_2018_hw10_DB1/train/'
imgVec, meanG = loadData(mypath)

mypath_t = 'ECE661_2018_hw10_DB1/test/'
imgVec_t, meanG_t = loadData(mypath_t)

labels = []
```

```
for c in range(nClasses):
    tmp = np.ones((nSamples, 1))
    labels.extend(tmp[:,0]*(c+1))
labels = np.asarray(labels)

#LDA
meanI = np.zeros((imgVec.shape[0],nClasses))#class mean
imgVecDiff = np.zeros(imgVec.shape)#diff btw sample and its
    class mean (within-class)
for c in range(nClasses):
    meanI[:,c] = np.mean(imgVec[:,c*nSamples:(c+1)*nSamples],
        axis=1)
    imgVecDiff[:,c*nSamples:(c+1)*nSamples] = imgVec[:,c*
        nSamples:(c+1)*nSamples]-meanI[:,c,None]

meanB = meanI - meanG[:,None];#diff btw class mean and global
    mean (between-class)
#compute between-class scatter
#SB = meanB.dot(meanB.transpose())
d, u = np.linalg.eig(meanB.transpose().dot(meanB))#30x30
idx = np.argsort(-1*d)
d = d[idx]
u = u[:,idx]
V = meanB.dot(u)#eigenvectors of SB
DB = np.eye(nClasses)*(d**(-0.5))
Z = V.dot(DB)#16384x30
#compute within-class scatter
#SW = imgVecDiff.dot(imgVecDiff.transpose())
X = np.dot(Z.transpose(), imgVecDiff)#30x630
dw, uw = np.linalg.eig(X.dot(X.transpose()))#30x30
idx = np.argsort(dw)
uw = uw[:,idx]

score = []
c = np.zeros((len(labels), 1))
for K in range(30):
    subSpace = Z.dot(uw[:,:K+1])#16384x(K+1)
    subSpace = subSpace/np.linalg.norm(subSpace, axis=0)
    feature_train = np.dot(subSpace.transpose(), imgVec-meanG
        [:,None])
    feature_test = np.dot(subSpace.transpose(), imgVec_t-
        meanG_t[:,None])
    #KNN
    classifier = KNeighborsClassifier(n_neighbors=1)
    classifier.fit(feature_train.transpose(), labels) #[
        nSamples, nFeatures]
    pred = classifier.predict(feature_test.transpose())
    c[pred == labels] = 1
    score += [np.sum(c)/len(labels)]
```

3.3 Part 2: Feature Extraction

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

```
"""
Created on Mon Nov 26 21:05:21 2018
Generate features for the images
@author: wehuang
"""
import os
import numpy as np
import cv2
import pickle

def boxSum(ptA, ptB, ptC, ptD, imgIntegral):
    A = imgIntegral[np.int(ptA[0])][np.int(ptA[1])]
    B = imgIntegral[np.int(ptB[0])][np.int(ptB[1])]
    C = imgIntegral[np.int(ptC[0])][np.int(ptC[1])]
    D = imgIntegral[np.int(ptD[0])][np.int(ptD[1])]

    return D-B-C+A

def computeFeature(mypath):
    files = os.listdir(mypath)
    files.sort()

    img = cv2.imread(mypath+files[0])
    imgAll = np.zeros((img.shape[0], img.shape[1], len(files))
        )
    for i in range(len(files)):
        img = cv2.imread(mypath+files[i])
        imgAll[:,:,i] = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    #compute integral image
    imgIntegral = np.cumsum(np.cumsum(imgAll, axis=1), axis=0)
    imgIntegral = np.concatenate((np.zeros((img.shape[0],1,len
        (files))), imgIntegral), axis=1)
    imgIntegral = np.concatenate((np.zeros((1,img.shape[1]+1,
        len(files))), imgIntegral), axis=0)

    features = []
    for n in range(np.int(img.shape[1]/2)):#number of filters
        filtSize = (n+1)*2
        for j in range(img.shape[0]):#vertical
            for i in range(img.shape[1]+1-filtSize):#
                horizontal
                boxSum0 = boxSum([j,i], [j,i+filtSize/2], [j
                    +1,i], [j+1,i+filtSize/2], imgIntegral)
                boxSum1 = boxSum([j,i+filtSize/2], [j,i+
                    filtSize], [j+1,i+filtSize/2], [j+1,i+
                    filtSize], imgIntegral)
                features.append((boxSum1-boxSum0).reshape
                    ((1,-1)))

    for n in range(np.int(img.shape[0]/2)):#number of filters
        filtSize = (n+1)*2
        for j in range(img.shape[0]+1-filtSize):#vertical
            for i in range(img.shape[1]+1-2):#horizontal
```

```
                boxSum0 = boxSum([j,i], [j,i+2], [j+filtSize
                    /2,i], [j+filtSize/2,i+2], imgIntegral)
                boxSum1 = boxSum([j+filtSize/2,i], [j+filtSize
                    /2,i+2], [j+filtSize,i], [j+filtSize,i+2],
                     imgIntegral)
                features.append((boxSum0-boxSum1).reshape
                    ((1,-1)))

    return np.asarray(features).reshape((len(features),-1))

mypath = 'ECE661_2018_hw10_DB2/test/positive/'
features = computeFeature(mypath)
file = open('test_pos', 'wb')
pickle.dump(features, file)
file.close()

mypath = 'ECE661_2018_hw10_DB2/test/negative/'
features = computeFeature(mypath)
file = open('test_neg', 'wb')
pickle.dump(features, file)
file.close()
```

3.4 Part 2: AdaBoost Training

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Nov 27 19:29:11 2018
AdaBoost training
@author: wehuang
"""
import numpy as np
import pickle
from cascadeWeak import cascadeWeak
import matplotlib.pyplot as plt

file = open('train_pos', 'rb')
train_pos = pickle.load(file)
file.close()

file = open('train_neg', 'rb')
train_neg = pickle.load(file)
file.close()

nPos = train_pos.shape[1]
nNeg = train_neg.shape[1]
nNegOrg = nNeg
FP = []
print('Initial # of negative samples: ',nNeg)
feature = np.concatenate((train_pos, train_neg), axis=1)
classifier = []
for s in range(10):
    print('Stage ', s+1)
    casWeak = cascadeWeak(feature, nPos, nNeg)
```

```
    classifier.append(casWeak)
    if (len(casWeak.idx) == nPos):
        break

    nNeg = len(casWeak.idx)-nPos
    print('# of negative samples: ',nNeg)
    feature_tmp = feature[:,casWeak.idx]
    feature = feature_tmp
    FP.append(nNeg/nNegOrg)
```

3.5 Part 2: Cascade Weak Classifiers

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Nov 28 14:54:46 2018
Cascade weak classifiers->strong classifier
@author: wehuang
"""
import numpy as np

class sClassifier():
    idx = []
    nWeak = 0
    ht = []

def cascadeWeak(feature, nPos, nNeg):
    #Step 1: initialize weights
    weights = np.concatenate((np.ones((1,nPos))*0.5/nPos, np.
        ones((1,nNeg))*0.5/nNeg), axis=1)
    labels = np.concatenate((np.ones((1,nPos)), np.zeros((1,
        nNeg))), axis=1)

    casWeak = sClassifier()
    alpha = []
    h = []
    ht = []
    for t in range(25):
        #Step 2: find the best weak classifier
        #--normalize weights
        weights = weights/np.sum(weights)
        #--construct an ordered list of all samples w.r.t.
            each feature
        weightsSort = np.tile(weights, (len(feature),1))
        labelsSort = np.tile(labels, (len(feature),1))
        idx = np.argsort(feature, axis=1)
        row = np.arange(len(feature)).reshape((-1,1))
        weightsSort = weightsSort[row, idx]
        labelsSort = labelsSort[row, idx]
        #--cumulative sum of weights
        TposW = np.sum(weights[:,:nPos])
        TnegW = np.sum(weights[:,nPos:])
        SposW = np.cumsum(weightsSort*labelsSort, axis=1)
        SnegW = np.cumsum(weightsSort, axis=1)-SposW
```

```
        #--two types error
        err = np.zeros((feature.shape[0],feature.shape[1],2))
        err[:,:,0] = SposW+TnegW-SnegW
        err[:,:,1] = SnegW+TposW-SposW
        min_idx = np.unravel_index(np.argmin(err), err.shape)
        minErr = err[min_idx]
        #--define the best weak classifier
        fi = min_idx[0]
        sortIdx = idx[fi,:]
        pred_tmp = np.zeros((feature.shape[1],1))
        pred = np.zeros((feature.shape[1],1))
        if min_idx[2] == 0:
            p = -1
            pred_tmp[min_idx[1]+1:] = 1
        else:
            p = 1
            pred_tmp[:min_idx[1]+1] = 1
        pred[sortIdx] = pred_tmp
        featureSort = feature[fi,:]
        featureSort = featureSort[sortIdx]
        if min_idx[1] == 0:
            theta = featureSort[0]-0.01
        elif min_idx[1] == -1:
            theta = featureSort[-1]+0.01
        else:
            theta = np.mean(featureSort[min_idx[1]-1:min_idx
                [1]+1])
        #Step 3: update weights for next run
        beta = minErr/(1-minErr)
        alpha.append(np.log(1/beta))
        h.append(pred.transpose())
        ht.append([fi, theta, p, np.log(1/beta)])
        weights = weights*(beta**(1-np.abs(labels-pred.
            transpose())))

        #Step 4: check if it meets criteria? (strong
            classifier)
        s = np.dot(np.asarray(h).transpose(), np.asarray(alpha
            ))
        th = np.min(s[:nPos])
        s_pred = np.zeros(s.shape)
        s_pred[s>=th] = 1

        print(np.sum(s_pred[nPos:])/nNeg)
        if (np.sum(s_pred[nPos:])/nNeg < 0.5):
            break
    #output the strong classifier
    updateIdx = []
    updateIdx.extend(np.arange(nPos))
    misNegIdx = [nPos+x for x in range(nNeg) if s_pred[nPos+x]
        == 1]
    updateIdx.extend(misNegIdx)
    casWeak.idx = np.asarray(updateIdx)
    casWeak.nWeak = t+1
```

```
    casWeak.ht = ht

    return casWeak
```

3.6 Part 2: AdaBoost Testing

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Nov 28 16:49:21 2018
AdaBoost testing
@author: wehuang
"""
import numpy as np
import pickle
import matplotlib.pyplot as plt

file = open('test_pos', 'rb')
test_pos = pickle.load(file)
file.close()

file = open('test_neg', 'rb')
test_neg = pickle.load(file)
file.close()

file = open('classifier', 'rb')
classifier = pickle.load(file)
file.close()

nPosOrg = test_pos.shape[1]
nNegOrg = test_neg.shape[1]
print('Total positive samples: ',nPosOrg)
print('Total negative samples: ',nNegOrg)
nPos = test_pos.shape[1]
nNeg = test_neg.shape[1]
posSample = test_pos
negSample = test_neg

nMisPos = 0
nCorNeg = 0
FN = []
FP = []
for s in range(len(classifier)):
    print('Stage: ',s+1)
    print('# of positive samples: ',nPos)
    print('# of negative samples: ',nNeg)
    ht = np.asarray(classifier[s].ht)
    fi = ht[:,0].astype(int)
    theta = ht[:,1]
    p = ht[:,2]
    alpha = ht[:,3]
    pred_th = 0.5*np.sum(alpha)

    #classify
```

```
feature = np.concatenate((posSample, negSample), axis=1)
predWeak = np.zeros((len(ht),feature.shape[1]))
feature_tmp = feature[fi ,:]
tmpWeak = (p*theta)[:,None]-p[:,None]*feature_tmp
predWeak[tmpWeak>=0] = 1

predStrong = np.zeros((feature.shape[1],1))
tmpStrong = np.dot(predWeak.transpose(),alpha)
predStrong[tmpStrong>=pred_th] = 1

#compute FN and FP
posCorIdx = [x for x in range(nPos) if predStrong[x] == 1]
negErrIdx = [x for x in range(nNeg) if predStrong[x+nPos]
    == 1]
print('# of correct positive samples: ',len(posCorIdx))
print('# of error negative samples: ',len(negErrIdx))
#update false negative
nMisPos += (nPos-len(posCorIdx))
FN.append(nMisPos/nPosOrg)
#update false positive
nCorNeg += (nNeg-len(negErrIdx))
FP.append((nNegOrg-nCorNeg)/nNegOrg)

posSample = posSample[:,posCorIdx]
negSample = negSample[:,negErrIdx]
nPos = len(posCorIdx)
nNeg = len(negErrIdx)
```