ECE 661: Homework 11

Xingguang Zhang

(Fall 2020)

1. Face recognition with PCA and LDA

Both PCA and LDA are dimension reduction techniques, they requires us to find a projection matrix W_k which can project the features of a set of multidimensional data X to a lower-dimensional space where the new features Y are easier to be separated.

1.1 Principle component analysis

PCA seeks to find the orthogonal directions along which the variance of the training data can be maximal.
First we need to normalize the input data to avoid the illumination influence. It's simply done by: $x_i = \frac{\vec{x}_i}{||\vec{x}_i||_2}$. We then align all training samples (assume there's N samples in total) by subtracting the mean of all training data m, where

$$m = \sum_{i=0}^{N} \vec{x}_i$$

The covariance matrix can be calculated by:

$$C = \boldsymbol{X}\boldsymbol{X}^T$$

Where

$$\boldsymbol{X} = [\vec{x}_0 - m|\vec{x}_1 - m|\cdots|\vec{x}_N - m]$$

The eigenvector indicates the primary direction of covariance matrix, so we will need to eigen-decompose C, which is to solve:

$$C\vec{w} = \lambda\vec{w}$$

Where $\vec{w}$ is eigenvector. However, since the high dimensional data $\boldsymbol{X}$ usually has more dimension than samples, it has much more columns than rows, which will make C very large and cost high to decompose. We can use the trick that rather than solve C, we decompose $\boldsymbol{X}^T\boldsymbol{X}$, a smaller matrix. If the eigenvector of $\boldsymbol{X}^T\boldsymbol{X}$ is $\vec{u}$, we have

$$\boldsymbol{X}^T\boldsymbol{X}\vec{u} = \lambda\vec{u}$$

Since $\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{X}\vec{u} = \lambda\boldsymbol{X}\vec{u}$, we can finally find $\vec{w}$ by $\vec{w} = \boldsymbol{X}\vec{u}$ If we want to reduce the dimension of x to k, we will need k eigenvectors with the largest k eigenvalues. Such projection matrix is formed by:

$$\boldsymbol{W}_k = [\vec{w}_1|\vec{w}_2|\cdots|\vec{w}_k]$$

Then we can transform both training and validation data from $\vec{x}$ to

$$\vec{y} = \boldsymbol{W}_k^T(\vec{x} - m)$$

1.2 Linear Discriminant Analysis (LDA)

LDA aims to find the directions that can maximize the between-class scatter and minimize the within-class scatter.
The between class scatter is defined as:

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

Where $\vec{m}_i$ is the mean of all samples in class i and $\vec{m}$ is the mean of all samples, $|C|$ is the number of classes.
The within-class scatter is defined as

$$S_W = \frac{1}{|C|}\sum_{i=0}^{|C|}\frac{1}{|C_i|}\sum_{k=1}^{|C_i|}(\vec{x}_k^i - \vec{m}_i)(\vec{x}_k^i - \vec{m}_i)^T$$

Where $|C_i|$ is the number of images in class i, $\vec{x}_k^i$ is the k-th image feature in the i-th class. The goal of LDA is to find the projection matrix $\boldsymbol{W}$ that can maximize the Fisher's discriminant function:

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

It can be shown that the $\boldsymbol{W}$ that maximizes $J(\boldsymbol{W})$ can be solved by solving

$$S_B\boldsymbol{W} = \lambda S_W \boldsymbol{W}$$

We can solve this by first do eigen-decomposition on $S_B\boldsymbol{W} = \frac{1}{|C|}MM^T$, just as we did in PCA, we can use the same trick and solve the eigenvectors of M^TM first, the eigenvectors of the former is $\vec{v}$ and the later is $\vec{u}$, then we have $\vec{v} = M\vec{u}$ The matrix of eigenvectors is

$$Y = [\vec{v}_1|\vec{v}_2|\cdots|\vec{v}_C]$$

And the diagnal matrix of eigenvalues of S_B is D_B
Then we can solve $\boldsymbol{W}$ by eigen decomposition on $Z^T S_W Z$, where

$$Z = YD_B^T$$

The eigenvector is $\vec{u}$, and finally $\boldsymbol{W}$ can be solved by stacking

$$\vec{w}_i = Z\vec{u}_i$$

For example, if we want the dimension of the dimension reduced feature as K, the projection matrix is

$$\boldsymbol{W}_k = [\vec{w}_1|\vec{w}_2|\cdots|\vec{w}_K]$$

1.3 NN classifier

We assign labels for the testing data after dimension reduction by the label of the nearest training sample in the lower dimension space.

1.4 result and observation

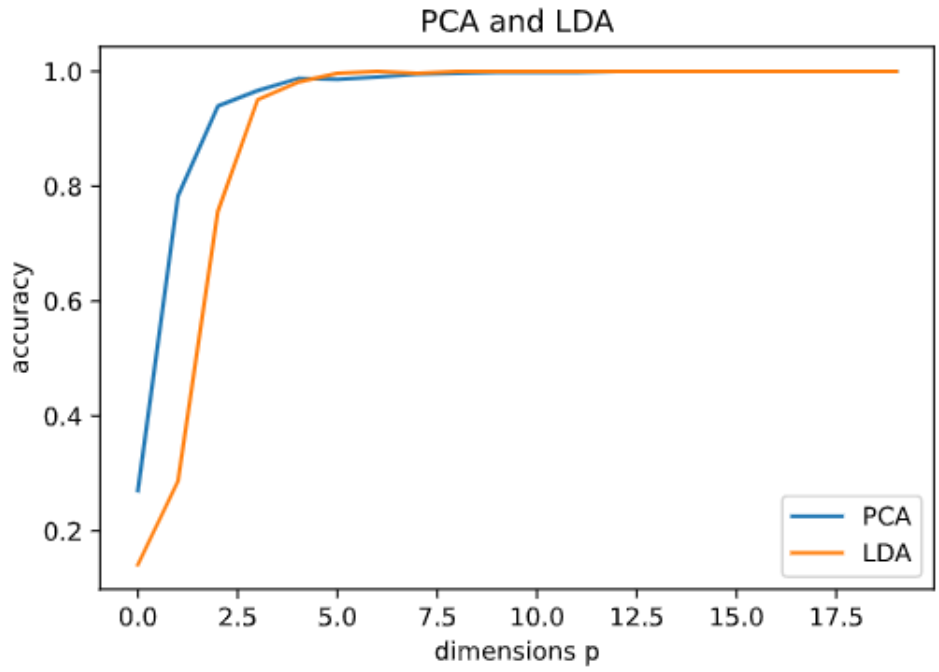

Figure 1: Training steps and accuracy for PCA and LDA

We can find that when the target dimension is small, PCA is better to determine which is the most significant feature direction, while the LDA is faster in convergence. The LDA achieve 100% accuracy when $p = 9$ and PCA achieve it when $p = 13$

Task 2 Car 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.

feature extraction

All images in the traning and testing sets are all in size 20×40. The goal of our classifier is to separate all cars from all images.
We can define a series of simple and weak classifiers and boost them to a strong classifier. Both the feature and the model will be extremely simple in this task. We used Haar-like boundary detectors to make multiple features. There are verticle and horizontal types of Haar-like filters, for horizontal type Haar-like filter, they have shape $1 \times 2k$, where $1 \leq k \leq 20$, the left half of the filter are all 1 and the right half are all 0.

$$\underbrace{1, 1, \cdots, 1}_{k}, \underbrace{-1, -1, \cdots, -1}_{k}$$

For the verticle ones, they have shape $2h \times 2$, with the top half all 1's and bottom half all -1's. We used cv2.integral to make the computation faster.

Training

The first step for training is to initialize the weights of the samples with equal possibility. Given m positive and n negative samples, the weight for positive samples is set to $\frac{1}{2m}$, and the weight for negative samples is set to $\frac{1}{2n}$. The weight needs to be normalized in every iteration.
Sort all the training samples with respect to each feature by ascending order. Compute the error by:

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

Where 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, S^- is the sum of negative sample weights below the current sample.
Since we only have two class, if the error rate is larger than 50%, we can just reverse it. We define a polarity $p \in \{+1, -1\}$ to indicate whether the classification should reverse (which means if the value is above the threshold, we assign the sample a negative prediction).
Loop over all features and we will find the minimal e_t, which defines the best weak classifier, we can find the feature f_t associated with the minimal error and θ_t which is the threshold. The classifier at step t is defined as $h_t(x) = h(x, f_t, p_t, \theta_t)$.
Then we need to update the weights for the next iteration by

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

If error for i-th sample is 0, it is classified correctly, we don't need to modified it's weight. If e is larger, then the sample need to gain more attention, hence, we need to enlarger its weight by $\beta_t = \frac{e_t}{1-e_t}$ Fianally we can cascade a strong classifier by taking use of all weak classifiers. For each classifier, it has a weight defined by

$$\alpha_t = log(\frac{1}{\beta_t})$$

The final strong classifier is $C(x) = 1$ if $\sum_{t=1}^{T} \alpha_t h_t(x) \geq threshold$.
To achieve true detection rate as 1, we set threshold is the minimum value of positive samples only in training process. During testing, we set the threshold as $0.5\sum_{t=1}^{T} \alpha_t$. The aggregation of weak classifiers will terminate until we reach the maximal steps (here we define it as **50**) or we have 0 false negative and less than 0.3 false positive rate.

Cascade strong classifiers

$h()$ is the weak classifier and $C()$ is the strong classifier, we will need to cascade strong classifiers. That is, we will try to eliminate false positives from all negative samples. Only the positive samples and negative samples that are falsely classified as positive will be send from the former strong classifier to the next classifier, until the maximal stage number (here we define as **10**) is achieved or there's no false positive.
An more efficient way is, we can directly pass all samples through all cascaded classifiers, and only take those classified in by every classifier as positive as the predicted positive, while others are all negative.
We usually use false positive rate (FPR) and false negative rate (FNR) to measure the performance, which are:

$$\text{FPR} = \frac{Number\ of\ misclassified\ negative\ test\ samples}{totalnumberofnegativetestsamples}$$

$$\text{FNR} = \frac{Number\ of\ misclassified\ positive\ test\ samples}{totalnumberofpositivetestsamples}$$

Training results

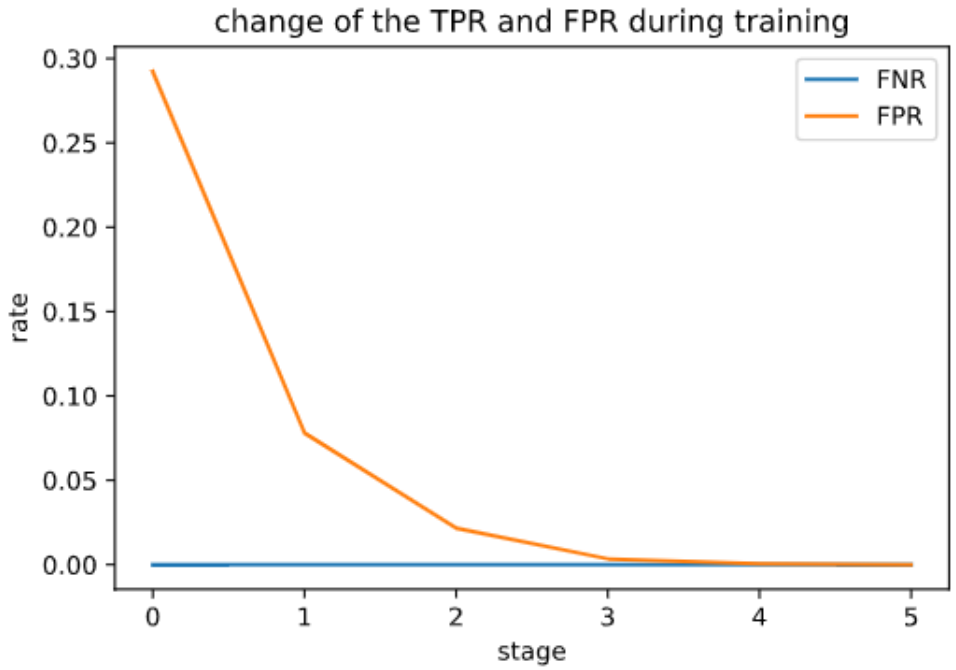

Figure 2: True positive rate and false positive rate during training

The details are shown below:

stage	number of weak classifiers	FNR	FPR	number of false positive samples
1	16	0	0.2924	514
2	29	0	0.0779	137
3	32	0	0.0216	38
4	20	0	0.0034	6
5	6	0	0.0006	1
6	1	0	0	0

Table 1: Adaboost performance during training

Testing results

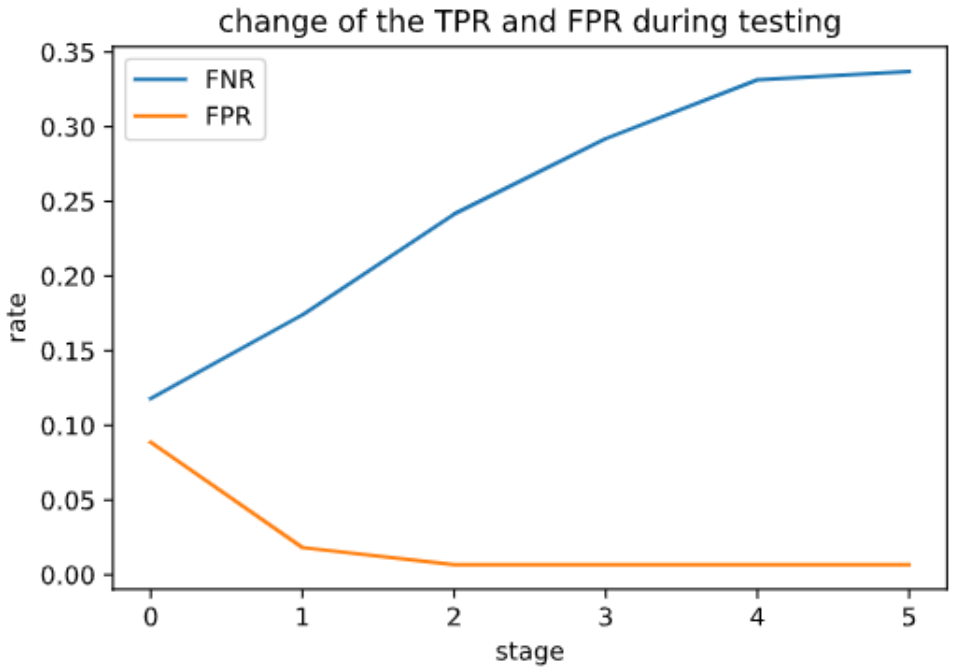

Figure 3: True positive rate and false positive rate during testing

The details are shown below:

stage	number of weak classifiers	FNR	FPR
1	16	0.1180	0.0886
2	29	0.1741	0.0182
3	32	0.2416	0.0068
4	20	0.2921	0.0068
5	6	0.3314	0.0068
6	1	0.3371	0.0068

Table 2: Adaboost performance on testing

Code

Task1

```
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt

def PCA(X):
    C = np.dot(X.T, X)
    lamda, u = np.linalg.eig(C)
```

```
    w = np.dot(X, u)
    sw = np.sqrt(np.sum(w**2, axis=0, keepdims = True))
    norm_w = w / sw
    return norm_w

def LDA(X, meanImg, meanImgClass, num_Sample):
    M = meanImgClass - meanImg
    eigvalues, eigenvectors = np.linalg.eig(np.dot(M.T, M))
    evidx = eigvalues.argsort()[::-1][:len(eigvalues)]
    u = eigenvectors[:, evidx]
    Y = np.dot(M, u)
    Db_sqrt = np.diag(np.power(eigvalues[evidx], -0.5))
    Z = np.dot(Y, Db_sqrt)

    Xw = X - np.repeat(meanImgClass, num_Sample, 1)
    ZSwZ = np.dot(np.dot(Z.T, Xw), np.dot(Z.T, Xw).T)
    Uev, U = np.linalg.eig(ZSwZ)
    U_hat = U[:, Uev.argsort()]
    Wk = np.dot(Z, U_hat)
    return Wk


def NN(train, test, train_label):
    ntest = np.sum(test**2, axis=1).reshape((-1, 1))
    ntrain = np.sum(train**2, axis=1).reshape((1, -1))
    tt = np.dot(test, train.T)
    edistance = np.sqrt(ntest - 2 * tt + ntrain)
    idx = edistance.argmin(axis=1)
    prediction = train_label[idx]
    acc = np.sum(prediction==test_label) / ntest.shape[0]
    return prediction

train_folder = 'DB1/train'
test_folder = 'DB1/test'
num_Class = 30
num_Sample = 21
train_data, train_label = [], []
test_data, test_label = [], []
for c in range(num_Class):
    for s in range(1, 1+num_Sample):
        img_path = os.path.join(train_folder, '{:02d}_{:02d}.png'.format(c+1, s))
        img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
        train_data.append(img.reshape((-1, 1)))
        train_label.append(c)

for c in range(num_Class):
    for s in range(1, 1+num_Sample):
        img_path = os.path.join(test_folder, '{:02d}_{:02d}.png'.format(c+1, s))
        img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
        test_data.append(img.reshape((-1, 1)))
        test_label.append(c)

train_vec = np.hstack(train_data)
train_vec = train_vec / np.linalg.norm(train_vec, axis=0, keepdims=True)
train_label = np.array(train_label)
test_vec = np.hstack(test_data)
test_vec = test_vec / np.linalg.norm(test_vec, axis=0, keepdims=True)
test_label = np.array(test_label)
```

```
mean_train = np.mean(train_vec, axis=1, keepdims=True)
std_train = np.std(train_vec - mean_train, axis=1, keepdims=True)
norm_train = (train_vec - mean_train) #/ std_train
norm_test = (test_vec - mean_train) #/ std_train

acc_pca = []
W = PCA(norm_train)
for p in range(1, 21):
    ev = W[:, :p]
    pca_train = np.dot(ev.T, norm_train)
    pca_test = np.dot(ev.T, norm_test)
    prediction = NN(pca_train.T, pca_test.T, train_label)
    acc = np.sum(prediction==test_label) / test_label.shape[0]
    acc_pca.append(acc)
    print("subspace dimensionality: {}, accuracy: {:.3f}%)".format(p, acc*100))


class_mean = np.zeros((train_vec.shape[0], num_Class))
for i in range(num_Class):
    idx1 = num_Sample * i
    idx2 = idx1 + num_Sample
    class_mean[:,i] = np.mean(train_vec[:,idx1:idx2], 1)

W = LDA(train_vec, mean_train, class_mean, num_Sample)
acc_LDA = []
for p in range(1, 21):
    Wp = W[:, :p]
    lda_train = np.dot(Wp.T, norm_train)
    lda_test = np.dot(Wp.T, norm_test)

    prediction = NN(lda_train.T, lda_test.T, train_label)
    acc = np.sum(prediction==test_label) / test_label.shape[0]
    acc_LDA.append(acc)
    print("subspace dimensionality: {}, accuracy: {:.3f}%)".format(p, acc*100))


steps = [i for i in range(len(acc_LDA))]
plt.plot(acc_pca, label='PCA')
plt.plot(acc_LDA, label = 'LDA')
plt.xlabel('dimensions p')
plt.ylabel('accuracy')
plt.title('PCA and LDA')
plt.legend()
plt.show()
```

Task2

```
import numpy as np
import cv2
import os
import math
import matplotlib.pyplot as plt

def itgboxsum(itgImg, h0, w0, h1, w1):
    A = itgImg[h0, w0]
    B = itgImg[h0, w1]
    C = itgImg[h1, w0]
```

```
    D = itgImg[h1, w1]
    return D - C - B + A


def create_HARR(h, w):
    '''
    Create Harr-like filters, only return the shape discription
    '''
    kernel_shapes = []
    # horizontal:
    for i in range(1, int(w/2)+1):
        h_win, w_win = 1, i
        h_shift, w_shift = 0, i
        kernel_shapes.append(((h_win, w_win), (h_shift, w_shift)))
    # vertical:
    for j in range(1, int(h/2)+1):
        h_win, w_win = j, 2
        h_shift, w_shift = j, 0
        kernel_shapes.append(((h_win, w_win), (h_shift, w_shift)))
    return kernel_shapes


def make_feature(img):
    '''
    Construct features by a series Harr-like filters
    '''
    h, w = img.shape
    itgImg = cv2.integral(img)
    filters = create_HARR(h, w)
    features = []
    for (h_win, w_win), (h_shift, w_shift) in filters:
        filter_h, filter_w = h_win + h_shift, w_win + w_shift
        for r in range(h - filter_h + 1):
            for c in range(w - filter_w + 1):
                hp0, wp0, hp1, wp1 = r, c, r + h_win, c + w_win
                s_pos = itgboxsum(itgImg, hp0, wp0, hp1, wp1)
                hn0, wn0, hn1, wn1 = r+h_shift, c+w_shift, r + filter_h, c +
   filter_w
                s_neg = itgboxsum(itgImg, hn0, wn0, hn1, wn1)
                features.append(s_pos-s_neg)
    return np.array(features)


def weakClassifer(feature, weights, labels, nPos, nNeg):
    '''
    Training a weak classifier
    '''
    weights = weights / np.sum(weights)
    idx = np.argsort(feature, axis=1)

    sorted_feature = np.zeros_like(feature)
    sorted_weights = np.zeros_like(feature).astype(np.float64)
    sorted_label = np.zeros_like(feature)
    for i in range(feature.shape[0]):
        sorted_feature[i] = feature[i][idx[i]].copy()
        sorted_weights[i] = weights[idx[i]].copy()
        sorted_label[i] = labels[idx[i]].copy()

    # compute error for all features parallelly
```

```
    T_pos = np.sum(weights[np.where(labels == 1)])
    T_neg = np.sum(weights[np.where(labels == 0)])
    S_pos = np.cumsum(sorted_weights * sorted_label, axis=1)
    S_neg = np.cumsum(sorted_weights, axis=1) - S_pos
    e0 = S_pos + T_neg - S_neg
    e1 = S_neg + T_pos - S_pos
    bestErr = np.inf
    best_idx_feature = 0
    # print(np.sum(sorted_label), np.sum(S_pos), np.sum(sorted_weights))
    for i in range(feature.shape[0]):
        minE = np.minimum(e0[i], e1[i])
        idx_minE = np.argmin(minE)

        if e0[i, idx_minE] <= e1[i, idx_minE]:
            p = 1
            prediction = (feature[i] >= sorted_feature[i, idx_minE])
            prediction = prediction.astype(int)
            clsfr = [i, sorted_feature[i, idx_minE], p, e0[i, idx_minE]]
        else:
            p = -1
            prediction = (feature[i] < sorted_feature[i, idx_minE])
            prediction = prediction.astype(int)
            clsfr = [i, sorted_feature[i, idx_minE], p, e1[i, idx_minE]]

        if minE[idx_minE] < bestErr:
            classifier = clsfr
            final_prediction = prediction
            bestErr = minE[idx_minE]

    return classifier, final_prediction


def strongClassifer(train_pos, train_neg):
    '''
    Construct a strong classifier by aggregating multiple weak ones
    '''
    nPos = train_pos.shape[1]
    nNeg = train_neg.shape[1]
    feature = np.concatenate((train_pos, train_neg), axis=1)
    weights = np.array([0.5/nPos]*nPos + [0.5/nNeg]*nNeg)
    labels = np.array([1]*nPos + [0]*nNeg)

    alphas = []
    prediction_list = []
    weak_clfrs = []
    for i in range(50):
        # Run weak classifier and get its result
        clfr, pred = weakClassifer(feature, weights, labels, nPos, nNeg)
        weak_clfrs.append(clfr)
        beta = (clfr[3]+1e-5) / (1-clfr[3]) # prevent the beta=0
        alpha = np.log(1 / beta)

        weights = weights * (beta ** (1 - np.abs(labels-pred)))

        alphas.append(alpha)
        prediction_list.append(pred)

        alpha_array = np.array([alphas]).T
        feature_array = np.array(prediction_list).T
```

```
        sf = np.dot(feature_array, alpha_array)

        th = np.min(sf[:nPos])

        Cx = np.zeros(sf.shape)
        # Assign labels and measure the performance
        Cx[sf >= th] = 1
        fpr = np.sum(Cx[nPos:])/nNeg
        # print('      ', beta, np.sum(Cx[nPos:]))
        tpr = np.sum(Cx[:nPos])/nPos

        tnr = 1 - fpr
        fnr = 1 - tpr

        print('step: {}: FNR: {}, FPR: {}, FP: {}'.format(i, fnr, np.sum(Cx[nPos
    :])/1758, np.sum(Cx[nPos:])))
        if tpr >= 1 and fpr <= 0.3:
            break
    cascade_log = {'weak_clfrs':weak_clfrs, 'alphas':alpha_array, 'tpr': tpr, 'fpr
    ':np.sum(Cx[nPos:])/1758, 'fnr':fnr, 'tnr':tnr}
    neg_samples_pred = Cx[nPos:]
    neg_idx = np.where(neg_samples_pred ==1)[0]
    new_train_neg = train_neg[:, neg_idx].copy()
    return cascade_log, new_train_neg

def cascade_classify(cascade_log, data_pos, data_neg):
    '''
    Use the cascaded strong classifiers to classify the testing samples
    '''
    nPos = data_pos.shape[1]
    nNeg = data_neg.shape[1]
    feature = np.concatenate((data_pos, data_neg), axis=1)
    pred_list = []
    # for each stage, get the prediction and collect them from every stage
    for classifer in cascade_log:
        weak_pred = []
        alpha_array = classifer['alphas']
        weak_classifers = classifer['weak_clfrs']
        # for the weak classifiers in the strong classifier
        for cid in range(len(weak_classifers)):
            f_idx = weak_classifers[cid][0]
            th = weak_classifers[cid][1]
            p = weak_classifers[cid][2]
            current_feature = feature[f_idx]
            if p == 1:
                classifResult = np.array(current_feature >= th).astype(int)
            else:
                classifResult = np.array(current_feature < th).astype(int)

            weak_pred.append(classifResult)
        wp_array = np.array(weak_pred).T
        agg_pred = np.dot(wp_array, alpha_array)
        threshold = np.sum(alpha_array) * 0.5
        Cx = np.zeros((agg_pred.shape[0], 1))
        Cx[agg_pred >= threshold] = 1
        pred_list.append(Cx)
    return pred_list
```

```
train_path = '/home/xingguang/Documents/ECE661/hw11/DB2/train'
test_path = '/home/xingguang/Documents/ECE661/hw11/DB2/test'
train_pos = []
train_pos_dir = os.path.join(train_path, 'positive')
for path in os.listdir(train_pos_dir):
    img_path = os.path.join(train_pos_dir, path)
    img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
    train_pos.append(make_feature(img))
train_pos = np.vstack(train_pos).T

train_neg = []
train_neg_dir = os.path.join(train_path, 'negative')
for path in os.listdir(train_neg_dir):
    img_path = os.path.join(train_neg_dir, path)
    img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
    train_neg.append(make_feature(img))
train_neg = np.vstack(train_neg).T

test_pos = []
test_pos_dir = os.path.join(test_path, 'positive')
for path in os.listdir(test_pos_dir):
    img_path = os.path.join(test_pos_dir, path)
    img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
    test_pos.append(make_feature(img))
test_pos = np.vstack(test_pos).T

test_neg = []
test_neg_dir = os.path.join(test_path, 'negative')
for path in os.listdir(test_neg_dir):
    img_path = os.path.join(test_neg_dir, path)
    img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
    test_neg.append(make_feature(img))
test_neg = np.vstack(test_neg).T

# start training
data_pos = train_pos.copy()
data_neg = train_neg.copy()
results = []
train_FN = []
train_FP = []
# build cascaded classifiers using at most 10 stages
for i in range(10):
    result_log, data_neg = strongClassifer(data_pos, data_neg)
    results.append(result_log)
    print('cascade {}, fnr: {}, fpr: {}'.format(i, result_log['fnr'], result_log['
    fpr']))
    train_FN.append(result_log['fnr'])
    train_FP.append(result_log['fpr'])
    if data_neg.shape[1] == 0:
        break


# test the cascaded classifier and collect results
result = cascade_classify(results, test_pos, test_neg)
test_FP = []
test_FN = []
ntpos = test_pos.shape[1]
ntneg = test_neg.shape[1]
for i in range(len(results)):
```

```
    if i == 0:
        f = result[0]
    else:
        f = np.concatenate(result[:i+1], axis=1)
    ff = np.sum(f, axis=1)
    pred = np.zeros(ff.shape)
    pred[ff == i+1]=1
    test_FP.append(np.sum(pred[ntpos:])/ntneg)
    test_FN.append(1 - np.sum(pred[:ntpos])/ntpos)
print(test_FP)
print(test_FN)

steps = [i for i in range(len(train_FN))]
plt.plot(train_FN, label='FNR')
plt.plot(train_FP, label = 'FPR')
plt.xlabel('stage')
plt.ylabel('rate')
plt.title('change of the TPR and FPR during training')
plt.legend()
plt.show()

steps = [i+1 for i in range(len(test_FP))]
plt.plot(test_FN, label='FNR')
plt.plot(test_FP, label = 'FPR')
plt.xlabel('stage')
plt.ylabel('rate')
plt.title('change of the TPR and FPR during testing')
plt.legend()
plt.show()
```