ECE 66100 HW10 Report

Zhengxin Jiang
(jiang839@purdue.edu)

December 10, 2022

1 Task 1: Face Recognition using PCA and LDA

For the first task, we perform the face recognition task by first extracting low-dimension features using PCA and LDA then doing classification using the 1-Nearest Neighbor algorithm. The detailed implementations of PCA and LDA are shown below.

1.1 Principal Components Analysis (PCA)

The general idea of PCA is to reduce the dimensionality by selecting p eigenvectors correspond to the p largest eigenvalues of the covarience matrix. The image vectors are then projected to the p dimensions. The implementation steps of PCA include:

1. Vectorize all images in the training set, do normalization and subtract the mean to centralize the dataset. Let the initialized dataset be X.
2. Apply the computational trick: instead of computing decomposing the covariance matrix $C = XX^T$, we do eigendecomposition on X^TX, which gives N eigenvalues and eigenvectors. N is the number of data samples.
3. Sort and take the eigenvectors $\vec{u}$ correspond to the p largest eigenvalues.
4. Now we calculate the real eigenvectors by
$$\vec{w} = X\vec{u}$$
5. Finally we project the original image vectors to the p dimensions.

1.2 Linear Discriminant Analysis (LDA)

The idea of LDA is to find the vector directions that maximizes the between-class scatter and minimizes the within-class scatter along that direction. To achieve this we do the following steps:

1. Vectorize the images and do normalization.
2. Compute the global mean and the per-class means of the image vectors.
3. Compute the mean matrix M by subtracting the global mean from the per-class mean matrix.
4. Do eigendecomposition on M^TM and sort the eigenvalues and eigenvectors.
5. Discard the eigenvectors which their correspond eigenvalues are close to 0. Then form the real eigenvectors Y.
6. Construct the diagonal matrix D_B which contains the remained eigenvalues.
7. Construct a matrix Z as
$$Z = YD_B^{-\frac{1}{2}}$$

8. Compute the matrix X by subtracting the per-class means from the image vectors. Then do eigendecomposition on $(Z^T X)(Z^T X)^T$.
9. Sort and take the p largest eigenvalues and the corresponding eigenvectors. Project the image vectors to the p dimensions.

Once we have the dimension-reduced training data, we can then reduce the dimension of the testing data and do classification using 1-nn algorithm.

2 Task 2: Face Recognition using an Autoencoder

For this task we are provided with the pre-trained autoencoder. All we have to do is to apply the 1-nn algorithm on the encoded training and testing features.

3 Task 3: Object Detection using Cascaded AdaBoost Classifiers

3.1 Feature Extraction

Before we run the cascaded classifiers, we first do feature extraction on the image set. Referring to the previous year report, we use the 1D Harr filter with all possible sizes for the image. All horizontal and vertical filters are applied. An example of an 1×4 filter is

$$\begin{bmatrix} -1 & -1 & 1 & 1 \end{bmatrix}$$

After all filters are applied, we can extract 11940 features from each image.

3.2 Cascaded AdaBoost Classifiers

The AdaBoost Classifier go through the features to select weak classifiers and finally construct a strong classifier. By constructing cascaded AdaBoost classifiers in an iterative way, we can achieve a very low false positive rate for the recognition task.

For each AdaBoost classifier, we loop over all features and for each feature, we estimate its trust factor and update the probability distribution over the training data for next loop. A final classifier will be constructed once all features are tested.

For the cascaded classifiers, we mainly update the training set by discarding those correctly classified nagative samples.

The test is to simply extract features from the test dataset and run it through the trained cascaded classifiers.

4 Results

4.1 Task 1 & 2

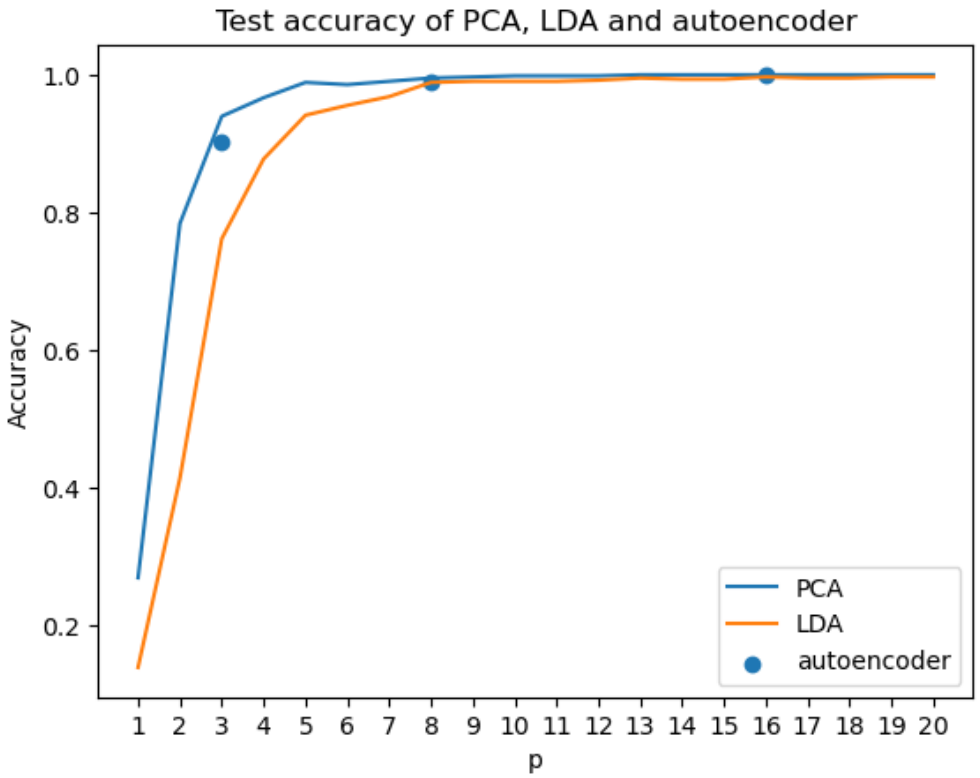

Figure 1: Accuracy plots of PCA, LDA and Autoencoder

For the performance comparison, PCA has better overall performance than LDA. And the autoencoder has the performance close to PCA.

4.2 Task 3

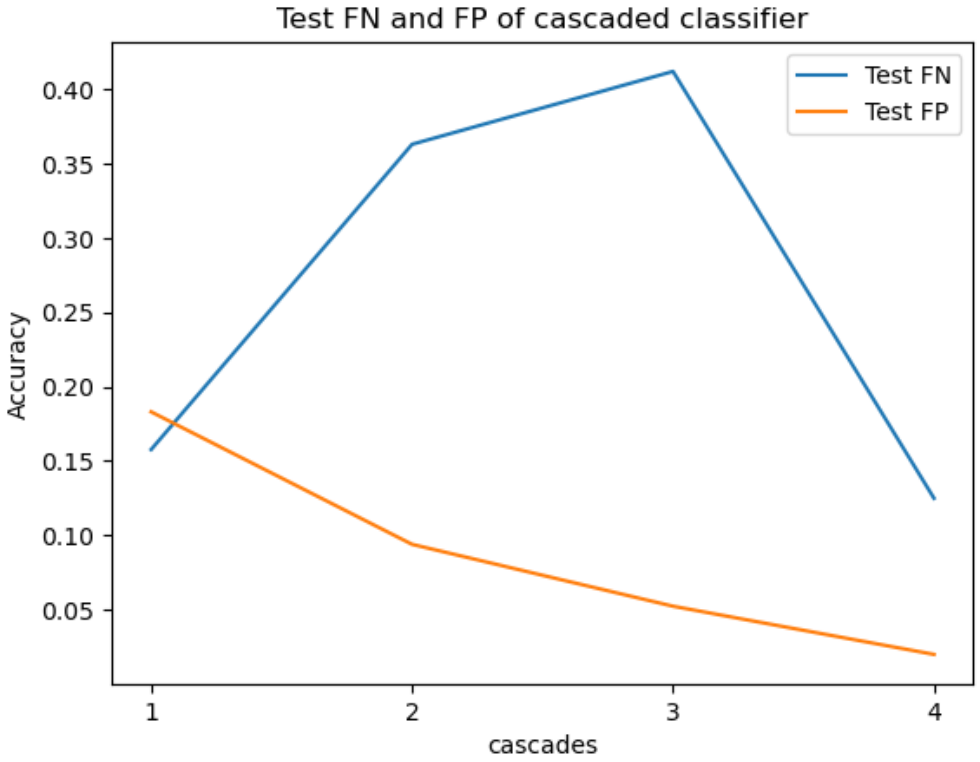

Figure 2: FP and FN plots of the cascaded classifier

Note: In my implementation I only use one weak classifier per cascade. Since there should be the cost of increasing false negative rate, the drop of FP might due to the selection of single classifier.

5 Source code

```
# ECE661 HW10
# Zhengxin Jiang
# jiang839


######## Task 1 ########

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

CLASS_NUM = 30
IMG_PER_CLASS_NUM = 21
IMG_SIZE = 128*128

# read the image set and do vectorization
def getVectorizedImages(path):

    imgvecs_train = []
    imgvecs_test = []

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

        img = cv2.imread(path+'train/'+filename)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        imgvec = np.ravel(img)
        imgvecs_train.append(imgvec/np.linalg.norm(imgvec))

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

        img = cv2.imread(path+'test/'+filename)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        imgvec = np.ravel(img)
        imgvecs_test.append(imgvec/np.linalg.norm(imgvec))

    return np.array(imgvecs_train), np.array(imgvecs_test)


# Extract P dimensions feature using PCA
def pca(imgvecs, P):

    imgvecs_meaned = imgvecs - np.mean(imgvecs, axis = 0)

    # use the computation trick in lecture notes
    C = np.dot(imgvecs_meaned, imgvecs_meaned.T)

    eigenvals, eigenvecs = np.linalg.eig(C)
```

```
    idx_sorted = np.argsort(eigenvals)[::-1]
    eigenvecs_sorted = eigenvecs[:,idx_sorted]
    eigenvecs_P = eigenvecs_sorted[:,0:P]

    w = []
    for i in range(P):

        wvec = np.dot(imgvecs_meaned.T, eigenvecs_P[:,i])
        w.append(wvec/np.linalg.norm(wvec))

    w = np.array(w)
    imgvecs_reduced = np.dot(w, imgvecs_meaned.T).T

    return w, imgvecs_reduced


# Extract P dimensions feature using LDA
def lda(imgvecs, P):

    globalmean = np.mean(imgvecs, axis = 0)
    classmeans = np.zeros((CLASS_NUM, IMG_SIZE))
    classimgvecs_meaned = np.zeros((IMG_PER_CLASS_NUM*CLASS_NUM, IMG_SIZE))

    # centralize
    for i in range(CLASS_NUM):

        classmean = np.mean(imgvecs[i*IMG_PER_CLASS_NUM:(i+1)*IMG_PER_CLASS_NUM], axis =
            0)
        classmeans[i] = classmean
        classimgvecs_meaned[i*IMG_PER_CLASS_NUM:(i+1)*IMG_PER_CLASS_NUM] = imgvecs[i*
            IMG_PER_CLASS_NUM:(i+1)*IMG_PER_CLASS_NUM] - classmean

    class_meaned = classmeans - globalmean

    # eigen decomposition
    M = np.dot(class_meaned, class_meaned.T)

    eigenvals, eigenvecs = np.linalg.eig(M)
    idx_sorted = np.argsort(eigenvals)[::-1]
    eigenvals_sorted = eigenvals[idx_sorted]
    # print(eigenvals_sorted)
    eigenvecs_sorted = eigenvecs[:,idx_sorted]

    # check the sorted eigenvalues and discard the last one
    # calculate the true eigen vectors
    Y = []
    for i in range(CLASS_NUM-1):

        Yvec = np.dot(class_meaned.T, eigenvecs_sorted[:,i])
        Y.append(Yvec/np.linalg.norm(Yvec))

    Y = np.array(Y).T
    new_eigenvals = eigenvals_sorted[:-1]
```

```
    # eigen decomposition
    DB = np.diag(new_eigenvals)
    Z = np.dot(Y, np.sqrt(np.linalg.inv(DB)))

    M_ = np.dot(np.dot(Z.T, classimgvecs_meaned.T), np.dot(Z.T, classimgvecs_meaned.T).T)

    eigenvals, eigenvecs = np.linalg.eig(M_)
    idx_sorted = np.argsort(eigenvals)[::-1]
    eigenvecs_sorted = eigenvecs[:,idx_sorted]
    eigenvecs_P = eigenvecs_sorted[:,0:P]

    # calculate the true eigen vectors again
    w = []
    for i in range(P):

        wvec = np.dot(Z, eigenvecs_P[:,i])
        w.append(wvec/np.linalg.norm(wvec))

    w = np.array(w)
    imgvecs_reduced = np.dot(w, (imgvecs - globalmean).T).T

    return w, imgvecs_reduced

# Get the test accuracy using 1-nn classifier
def testAccuracy(imgvecs_test, vecmean, imgvecs_reduced, eigenvecs):

    match = 0

    for i in range(len(imgvecs_test)):

        truelabel = i//IMG_PER_CLASS_NUM

        imgvec = imgvecs_test[i] - vecmean
        imgvec_reduced = np.dot(eigenvecs, imgvec.T).T

        dist = np.sqrt(np.sum((imgvecs_reduced - imgvec_reduced)**2, axis = 1))

        # using 1-nn
        predictlabel = np.argmin(dist)//IMG_PER_CLASS_NUM

        if truelabel==predictlabel:
            match += 1

    return match/len(imgvecs_test)

if __name__ == '__main__':

    imgpath = './'
# print(os.listdir(imgpath+'train/'))

    imgvecs_train, imgvecs_test = getVectorizedImages(imgpath)
    vecmean = np.mean(imgvecs_train, axis = 0)

    pca_acc = []
```

```
    lda_acc = []

    for P in range(1, 21):

        eigenvecs_pca, imgvecs_reduced_pca = pca(imgvecs_train, P)
        eigenvecs_lda, imgvecs_reduced_lda = lda(imgvecs_train, P)

        pca_acc.append(testAccuracy(imgvecs_test, vecmean, imgvecs_reduced_pca,
            eigenvecs_pca))
        lda_acc.append(testAccuracy(imgvecs_test, vecmean, imgvecs_reduced_lda,
            eigenvecs_lda))

    # plot autoencoder accuracy together
    ae_acc = [0.9015873015873016, 0.9904761904761905, 1.0]

    p = range(1, 21)

    plt.figure()
    plt.plot(p, pca_acc, label= 'PCA')
    plt.plot(p, lda_acc, label= 'LDA')
    plt.scatter([3,8,16], ae_acc, label= 'autoencoder')

    plt.title('Test␣accuracy␣of␣PCA,␣LDA␣and␣autoencoder')
    plt.ylabel('Accuracy')
    plt.xlabel('p')
    plt.xticks(range(1, 21))
    plt.legend()
    plt.show()


######## Task 2 ########
import os

import numpy as np
import torch
from torch import nn, optim
from PIL import Image
from torch.autograd import Variable
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms


class DataBuilder(Dataset):
    def __init__(self, path):
        self.path = path
        self.image_list = [f for f in os.listdir(path) if f.endswith('.png')]
        self.label_list = [int(f.split('_')[0]) for f in self.image_list]
        self.len = len(self.image_list)
        self.aug = transforms.Compose([
            transforms.Resize((64, 64)),
            transforms.ToTensor(),
        ])

    def __getitem__(self, index):
```

```
        fn = os.path.join(self.path, self.image_list[index])
        x = Image.open(fn).convert('RGB')
        x = self.aug(x)
        return {'x': x, 'y': self.label_list[index]}

    def __len__(self):
        return self.len


class Autoencoder(nn.Module):

    def __init__(self, encoded_space_dim):
        super().__init__()
        self.encoded_space_dim = encoded_space_dim
        ### Convolutional section
        self.encoder_cnn = nn.Sequential(
            nn.Conv2d(3, 8, 3, stride=2, padding=1),
            nn.LeakyReLU(True),
            nn.Conv2d(8, 16, 3, stride=2, padding=1),
            nn.LeakyReLU(True),
            nn.Conv2d(16, 32, 3, stride=2, padding=1),
            nn.LeakyReLU(True),
            nn.Conv2d(32, 64, 3, stride=2, padding=1),
            nn.LeakyReLU(True)
        )
        ### Flatten layer
        self.flatten = nn.Flatten(start_dim=1)
        ### Linear section
        self.encoder_lin = nn.Sequential(
            nn.Linear(4 * 4 * 64, 128),
            nn.LeakyReLU(True),
            nn.Linear(128, encoded_space_dim * 2)
        )
        self.decoder_lin = nn.Sequential(
            nn.Linear(encoded_space_dim, 128),
            nn.LeakyReLU(True),
            nn.Linear(128, 4 * 4 * 64),
            nn.LeakyReLU(True)
        )
        self.unflatten = nn.Unflatten(dim=1,
                                      unflattened_size=(64, 4, 4))
        self.decoder_conv = nn.Sequential(
            nn.ConvTranspose2d(64, 32, 3, stride=2,
                               padding=1, output_padding=1),
            nn.BatchNorm2d(32),
            nn.LeakyReLU(True),
            nn.ConvTranspose2d(32, 16, 3, stride=2,
                               padding=1, output_padding=1),
            nn.BatchNorm2d(16),
            nn.LeakyReLU(True),
            nn.ConvTranspose2d(16, 8, 3, stride=2,
                               padding=1, output_padding=1),
            nn.BatchNorm2d(8),
            nn.LeakyReLU(True),
```

```
            nn.ConvTranspose2d(8, 3, 3, stride=2,
                               padding=1, output_padding=1)
        )

    def encode(self, x):
        x = self.encoder_cnn(x)
        x = self.flatten(x)
        x = self.encoder_lin(x)
        mu, logvar = x[:, :self.encoded_space_dim], x[:, self.encoded_space_dim:]
        return mu, logvar

    def decode(self, z):
        x = self.decoder_lin(z)
        x = self.unflatten(x)
        x = self.decoder_conv(x)
        x = torch.sigmoid(x)
        return x

    @staticmethod
    def reparameterize(mu, logvar):
        std = logvar.mul(0.5).exp_()
        eps = Variable(std.data.new(std.size()).normal_())
        return eps.mul(std).add_(mu)


class VaeLoss(nn.Module):
    def __init__(self):
        super(VaeLoss, self).__init__()
        self.mse_loss = nn.MSELoss(reduction="sum")

    def forward(self, xhat, x, mu, logvar):
        loss_MSE = self.mse_loss(xhat, x)
        loss_KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
        return loss_MSE + loss_KLD


def train(epoch):
    model.train()
    train_loss = 0

    for batch_idx, data in enumerate(trainloader):
        optimizer.zero_grad()
        mu, logvar = model.encode(data['x'])
        z = model.reparameterize(mu, logvar)
        xhat = model.decode(z)
        loss = vae_loss(xhat, data['x'], mu, logvar)
        loss.backward()
        train_loss += loss.item()
        optimizer.step()

    print('====> Epoch: {} Average loss: {:.4f}'.format(
        epoch, train_loss / len(trainloader.dataset)))
```

```
##################################
# Change these
p = 16 # [3, 8, 16]
training = False
TRAIN_DATA_PATH = './train'
EVAL_DATA_PATH = './test'
LOAD_PATH = f'./model_{p}.pt'
OUT_PATH = './aeout'
##################################

model = Autoencoder(p)

if training:
    epochs = 100
    log_interval = 1
    trainloader = DataLoader(
        dataset=DataBuilder(TRAIN_DATA_PATH),
        batch_size=12,
        shuffle=True,
    )
    optimizer = optim.Adam(model.parameters(), lr=1e-3)
    vae_loss = VaeLoss()
    for epoch in range(1, epochs + 1):
        train(epoch)
    torch.save(model.state_dict(), os.path.join(OUT_PATH, f'model_{p}.pt'))
else:
    trainloader = DataLoader(
        dataset=DataBuilder(TRAIN_DATA_PATH),
        batch_size=1,
    )
    model.load_state_dict(torch.load(LOAD_PATH))
    model.eval()

    X_train, y_train = [], []
    for batch_idx, data in enumerate(trainloader):
        mu, logvar = model.encode(data['x'])
        z = mu.detach().cpu().numpy().flatten()
        X_train.append(z)
        y_train.append(data['y'].item())
    X_train = np.stack(X_train)
    y_train = np.array(y_train)

    testloader = DataLoader(
        dataset=DataBuilder(EVAL_DATA_PATH),
        batch_size=1,
    )
    X_test, y_test = [], []
    for batch_idx, data in enumerate(testloader):
        mu, logvar = model.encode(data['x'])
        z = mu.detach().cpu().numpy().flatten()
        X_test.append(z)
        y_test.append(data['y'].item())
    X_test = np.stack(X_test)
    y_test = np.array(y_test)
```

```
    ###################################
    # Your code starts here
# print(X_train.shape)
# print(X_test.shape)

    match = 0

    for i in range(len(X_test)):

        truelabel = y_test[i]

        dist = np.sqrt(np.sum((X_train - X_test[i])**2, axis = 1))

        # using 1-nn
        predictlabel = y_train[np.argmin(dist)]

        if truelabel==predictlabel:
            match += 1

    acc = match/len(X_test)
    print(acc)


    ###################################

######## Task 3 ########
import numpy as np
import cv2
import math
import matplotlib.pyplot as plt
import os


# Apply filter and get the feature vector for an image
def imgFeatureVec(img):

    featurevec = []
    filterw = np.arange(2, img.shape[1], 2)
    filterh = np.arange(2, img.shape[0], 2)

    # Apply Haar filter
    for w in filterw:
        for x in range(img.shape[1] - w +1):
            for y in range(img.shape[0]):

                pval = np.sum(img[y, x+int(w/2):x+w]).astype(np.int32)
                nval = np.sum(img[y, x:x+int(w/2)]).astype(np.int32)

                featurevec.append(pval-nval)

    for h in filterh:
        for x in range(img.shape[1]):
            for y in range(img.shape[0] - h +1):
```

```
                pval = np.sum(img[y+int(h/2):y+h, x]).astype(np.int32)
                nval = np.sum(img[y:y+int(h/2), x]).astype(np.int32)

                featurevec.append(pval-nval)

    return featurevec

# read the image set and do vectorization
def getFeatureVectors(path):

    features_train_p = []
    features_train_n = []
    features_test_p = []
    features_test_n = []

    for filename in os.listdir(path+'train/positive/'):

        img = cv2.imread(path+'train/positive/'+filename)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        featurevec = imgFeatureVec(img)
        features_train_p.append(featurevec)

    for filename in os.listdir(path+'train/negative/'):

        img = cv2.imread(path+'train/negative/'+filename)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        featurevec = imgFeatureVec(img)
        features_train_n.append(featurevec)

    for filename in os.listdir(path+'test/positive/'):

        img = cv2.imread(path+'test/positive/'+filename)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        featurevec = imgFeatureVec(img)
        features_test_p.append(featurevec)

    for filename in os.listdir(path+'test/negative/'):

        img = cv2.imread(path+'test/negative/'+filename)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        featurevec = imgFeatureVec(img)
        features_test_n.append(featurevec)


    return np.array(features_train_p), np.array(features_train_n), np.array(
        features_test_p), np.array(features_test_n)


imgpath = './'
```

```
# features_train_p, features_train_n, features_test_p, features_test_n =
    getFeatureVectors(imgpath)
# np.savetxt('features_train_p.txt', features_train_p, '%d')
# np.savetxt('features_train_n.txt', features_train_n, '%d')
# np.savetxt('features_test_p.txt', features_test_p, '%d')
# np.savetxt('features_test_n.txt', features_test_n, '%d')

features_train_p = np.loadtxt('features_train_p.txt', dtype = np.int16)
features_train_n = np.loadtxt('features_train_n.txt', dtype = np.int16)
features_test_p = np.loadtxt('features_test_p.txt', dtype = np.int16)
features_test_n = np.loadtxt('features_test_n.txt', dtype = np.int16)

#training
features_train = np.concatenate((features_train_p,features_train_n), axis=0)
labels_train = np.zeros(len(features_train))
labels_train[:len(features_train_p)] = 1

features_test = np.concatenate((features_test_p,features_test_n), axis=0)
labels_test = np.zeros(len(features_test))
labels_test[:len(features_test_p)] = 1

fn_test_list = []
fp_test_list = []

# go through cascades
for c in range(5):

    print(f'Cascade␣{c}')

    feat_num = features_train.shape[1]
    p_num = np.count_nonzero(labels_train)
    n_num = len(labels_train) - p_num

    # initialize weights
    weights = np.zeros(len(features_train))
    weights[:p_num] = 1/(p_num*2)
    weights[p_num:] = 1/(n_num*2)

    # find the weak classifier
    classifier = None
    min_error = float('inf')

    for i in range(feat_num):

        featvec = features_train[:, i]

        idx_sorted = np.argsort(featvec)

        features_sorted = featvec[idx_sorted]
        labels_sorted = labels_train[idx_sorted]
        weights_sorted = weights[idx_sorted]

        # calc error for every threshold
        pweights_sum = np.sum(weights[labels_train == 1])
```

```
        nweights_sum = np.sum(weights[labels_train == 0])

        pweights_sorted = weights_sorted.copy()
        pweights_sorted[np.where(labels_sorted == 0)[0]] = 0
        nweights_sorted = weights_sorted.copy()
        nweights_sorted[np.where(labels_sorted == 1)[0]] = 0

        pweights_sorted_cumsum = np.cumsum(pweights_sorted)
        nweights_sorted_cumsum = np.cumsum(nweights_sorted)

        errminus = pweights_sum - pweights_sorted_cumsum + nweights_sorted_cumsum
        errplus = nweights_sum - nweights_sorted_cumsum + pweights_sorted_cumsum
        err = np.concatenate((errplus, errminus))

        curr_min_err = np.min(err)

        # update the selected feature
        if curr_min_err<min_error:

            min_error = curr_min_err

            tempidx = err.argmin()
            featidx = tempidx % (p_num+n_num)
            polarity = tempidx // (p_num+n_num)
            threshold = features_sorted[featidx]

            if polarity == 0:
                idx_classified_positive = featvec >= threshold
                fn = 1 - np.count_nonzero(idx_classified_positive[:p_num])/p_num
                fp = np.count_nonzero(idx_classified_positive[p_num:])/n_num

            else:
                idx_classified_positive = featvec < threshold
                fn = 1 - np.count_nonzero(idx_classified_positive[:p_num])/p_num
                fp = np.count_nonzero(idx_classified_positive[p_num:])/n_num


            classifier = [featidx, polarity, threshold, (fn, fp), idx_classified_positive]

    # update dataset
    pfeatures_new = features_train[:p_num]
    nfeatures = features_train[p_num:]
    nfeatures_new = nfeatures[np.where(idx_classified_positive[p_num:] == 1)[0]]

    features_train = np.concatenate((pfeatures_new, nfeatures_new), axis=0)
    labels_train = np.zeros(len(features_train))
    labels_train[:p_num] = 1

    #testing
    featvec_test = features_test[:, featidx]
    p_num_test = np.count_nonzero(labels_test)
    n_num_test = len(labels_test) - p_num_test

    if polarity == 0:
```

```
        test_classified_positive = featvec_test >= threshold
        fn_test = 1 - np.count_nonzero(test_classified_positive[:p_num_test])/p_num_test
        fp_test = np.count_nonzero(test_classified_positive[p_num_test:])/n_num_test

    else:
        test_classified_positive = featvec_test < threshold
        fn_test = 1 - np.count_nonzero(test_classified_positive[:p_num_test])/p_num_test
        fp_test = np.count_nonzero(test_classified_positive[p_num_test:])/n_num_test



    fn_test_list.append(fn_test)
    fp_test_list.append(fp_test)
    print(classifier)
    print(fn_test, fp_test)
```