ECE661 Computer Vision HW10

Chengjun Guo
guo456@purdue.edu

December 2022

1 Logic

1.1 PCA

The first step of PCA is to vectorize the image. We first flatten the image into a column vector. Then we normalize the vectors. For N images, we can construct matrix

$$X = [\overrightarrow{x_0} - \overrightarrow{m}|\overrightarrow{x_1} - \overrightarrow{m}|\cdots|\overrightarrow{x_{N-1}} - \overrightarrow{m}]$$

where $\overrightarrow{x_i}$ are the vectors, $\overrightarrow{m}$ is the global mean. Then the covariance matrix can be constructed $C = XX^T$. The computational cost of eigenvectors of C is large since C is 16384 by 16384 matrix in this case.
The trick here is to calculate the eigenvectors $\overrightarrow{u}$ of X^TX which is a N by N matrix. The eigenvectors $\overrightarrow{w}$ of C are found by $\overrightarrow{w} = X\overrightarrow{u}$.
The logic of this trick is as follows:

$$X^TX\overrightarrow{u} = \lambda\overrightarrow{u}$$
$$XX^TX\overrightarrow{u} = \lambda X\overrightarrow{u}$$
$$XX^T(X\overrightarrow{u}) = \lambda X\overrightarrow{u}$$
$$XX^T\overrightarrow{w} = \lambda\overrightarrow{w}$$

Then the eigenvectors $\overrightarrow{w}$ are normalized.
The eigenvectors are sorted by the eigenvalues in descending order and the largest P are selected. P is the number of expected dimensions. The subspace W_P is constructed by:

$$W_P = [w_0|w_1|\cdots|w_{p-1}]$$

Then the image projection on the subspace known as the feature is:

$$\overrightarrow{y} = W_P^T(\overrightarrow{x_i} - \overrightarrow{m})$$

. Then test image is classified by the nearest neighbour of the feature values of train images.

1.2 LDA

We vectorize and normalize the image in the same way mentioned in PCA section. We want to find the dimension that maximize $J(w_i)$. The Fischer Discriminant Function can be calculated by:

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

where S_B is the between class scatter and S_W is within class scatter.

$$S_B = \frac{1}{|C|} \sum_{i=1}^{|C|} \left(\overrightarrow{\mathbf{m}}_i - \overline{\mathbf{m}}\right) \left(\overrightarrow{\mathbf{m}}_i - \overline{\mathbf{m}}\right)^T$$

$$S_W = \frac{1}{|C|} \sum_{i=1}^{|C|} \frac{1}{|C_i|} \sum_{k=1}^{|C_i|} \left(\overrightarrow{\mathbf{x}}_{ik} - \overrightarrow{\mathbf{m}}_i\right) \left(\overrightarrow{\mathbf{x}}_{ik} - \overrightarrow{\mathbf{m}}_i\right)^T$$

where $|C|$ means the number of images in class C ,$\overrightarrow{\mathbf{m}}_i$ is the class mean and $\overrightarrow{\mathbf{x}}_{ik}$ is image k in class i.
Using the same trick mentioned in PCA section, we can find the largest M eigenvectors Y and eigenvalues D_B of S_B. Now we can construct

$$Z = Y D_B^{-\frac{1}{2}}$$

Then we compute the eigenvectors U of $Z^T S_W Z$. Sort U with eigenvalues in ascending order and find the smallest P eigenvectors to form U_P. Normalize U_P, then the subspace can be formed by $W_P = ZU_P$. Same as the PCA, the feature value can be calculated with

$$\overrightarrow{y} = W_P^T(\overrightarrow{x_i} - \overrightarrow{m})$$

. The classification with the feature values is the same as the PCA section.

1.3 Cascaded classifier

The first step is to use HAAR filter to extract image features. We use 1×2,1×4 ,$\cdots$, 1×40 for horizontal direction and 2×1,4×1,$\cdots$, 40×1 for vertical direction. For the image size of 40 by 20, we have 12000 features.
Then we initialize the weights for M positives and N negatives. The weights are initialized to $\frac{1}{2M}$ for positives and $\frac{1}{2N}$ for negatives.
For each iteration. we construct weak classifiers:
a. normalize the weights
b. For each feature in all images, sort the feature, corresponding labels and weights. The weak classifier h threshold the set into two classes with accuracy larger than 50%(we use polarity p to make sure the separation have accuracy larger than 50%). The classification error is calculated by

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

where T^+ and T^- is sum of positive sample weights and sum of negative weights, S^+ and S^- is sum of positive sample weights below current sample and sum of negative weights below current sample.
c. The weak classifier h_t would be selected with minimum error e_t. It will be recorded with the feature index, threshold, polarity and minimum error.
After The weak classifiers are found in a cascade stage, we calculate the confidence parameters

$$\beta = \frac{\epsilon}{1-\epsilon}$$

The weight for the next iteration would be

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

If the true detection rate is 1 and the false positive rate is 0.5, we break the loop.
Then the final strong classifier is constructed. If all of the αh_t is larger than the thresholds, it is detected as positive. Negative otherwise. Where

$$\alpha = \ln\left(\frac{1}{\beta}\right)$$

We break the iteration if the accumulated false positive rate is 0, else, we pass the positive examples and the false negative samples to the next iteration.

For testing, we apply each weak classifier in each cascade stages to each images in test. The predictions by each weak classifier is adjusted with the α from training. The positive images and the false negative images are left for the next iteration. The iteration stops if there is no negative images left.

2 Image and Data

2.1 Task1

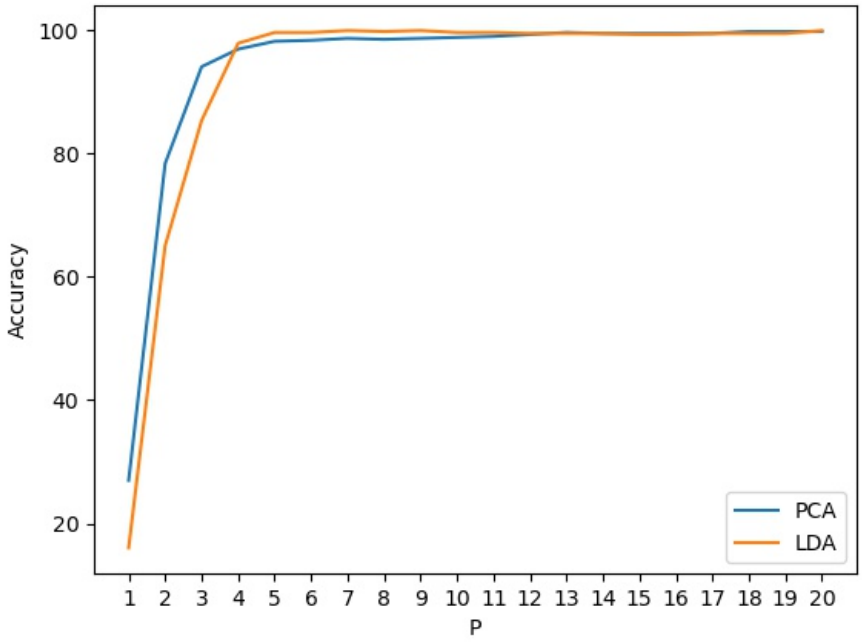

Figure 1: PCA vs LDA

2.2 Task2

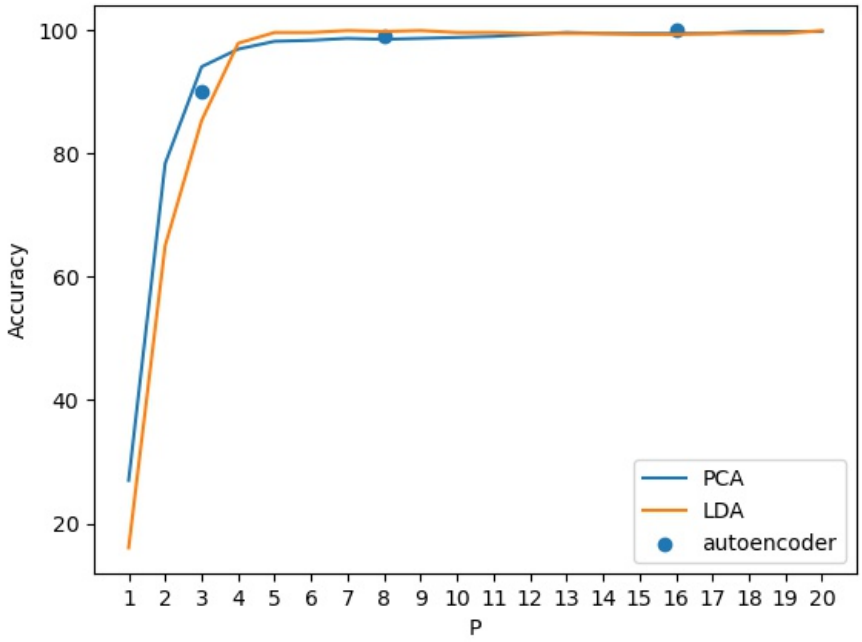

Figure 2: PCA, LDA, Autoencoder

2.3 Task3

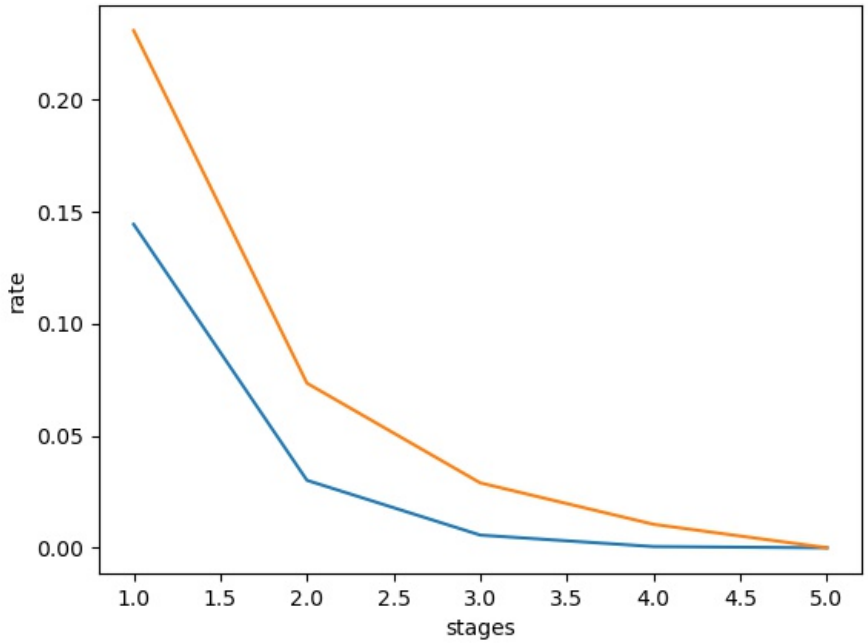

Figure 3: FP rate, FN rate vs k stage

3 Observation

3.1 PCA vs LDA

PCA converges faster in the first several p, however it slows down at dimension 3. LDA converges faster from 3 and achieves 100% accuracy first.

3.2 Autoencoder vs PCA and LDA

Autoencoder have better performance than LDA but worse performance than PCA at p=3. When P=8, it is better than PCA but worse than LDA. At p=16, it achieves 100% accuracy.

4 Code

4.1 Task1

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

def vectorize_image(img):
    img = np.array(img)
    img_vec = img.flatten()
    img_vec = img_vec/np.linalg.norm(img)
    return img_vec


def PCA(img_vec, P):
    #decompose XTX for u
    eigenValues, eigenVectors = np.linalg.eig(np.dot(
        img_vec.T,img_vec))
    idx = np.argsort(-eigenValues)
    eigenValues = eigenValues[idx]
    eigenVectors = eigenVectors[:, idx]
    #calculate w
    w = np.dot(img_vec, eigenVectors)
    w = w / np.linalg.norm(img_vec,axis=0)
    return w[:,:P]


def LDA(img_vec,vec_mean ,P, num_classes=30, num_samples
    =21):
    class_means = np.zeros((16384,num_classes)) # between
         class
    in_class_diff = np.zeros(img_vec.shape) # within
        class
    for i in range(num_classes):
        class_mean = np.mean(img_vec[:,i*num_samples:(i
            +1)*num_samples],axis=1)
        class_means[:,i] = class_mean
        in_class_diff[:,i*num_samples:(i+1)*num_samples]
            = img_vec[:,i*num_samples:(i+1)*num_samples] -
```

```
            class_mean.reshape(-1,1)
    between_class_diff = class_means - vec_mean.reshape
        (-1,1)
    eigenValues, eigenVectors = np.linalg.eig(np.dot(
        between_class_diff.T, between_class_diff))
    idx = np.argsort(-eigenValues)
    eigenValues = eigenValues[idx]
    eigenVectors = eigenVectors[:, idx]
    V = np.dot(between_class_diff, eigenVectors)
    DB = np.eye(num_classes) * (1/np.sqrt(eigenValues))
    Z = np.dot(V,DB)
    X = np.dot(Z.T, in_class_diff)
    eigenValues, eigenVectors = np.linalg.eig(np.dot(X, X
        .T))
    idx = np.argsort(-eigenValues)
    eigenValues = eigenValues[idx]
    eigenVectors = eigenVectors[:, idx]
    w = np.dot(eigenVectors[:P],Z.T).T
    return w


def get_1NN(train,test):
    dist = np.sqrt(np.sum(np.subtract(train,test.reshape
        (-1,1)) ** 2,axis=0))
    idx = np.argmin(dist)
    label = math.ceil((idx+1) / 21)
    # print(idx,label)
    return label



def task1():
    train_path = 'FaceRecognition/train/'
    test_path = 'FaceRecognition/test/'
    train_vec = []
    test_vec = []
    for img in os.listdir(train_path):
        # print(img)
        train_vec.append(vectorize_image(cv2.imread(
            train_path+img,0)))
    for img in os.listdir(test_path):
        test_vec.append(vectorize_image(cv2.imread(
            test_path + img, 0)))
    train_vec = np.array(train_vec)
    test_vec = np.array(test_vec)
    # subtract mean
```

```
    vec_mean = np.mean(train_vec,axis=0)
    train_vec = train_vec - vec_mean # 630 x 16384
    test_vec = test_vec - vec_mean
    pca_acc = []
    lda_acc = []
    for P in range(1,21):
        pca_w = PCA(train_vec.T,P)
        lda_w = LDA(train_vec.T,vec_mean,P)
        pca_train = np.dot(pca_w.T, train_vec.T) # (1,
            630)
        lda_train = np.dot(lda_w.T, train_vec.T)
        pca_correct = 0
        lda_correct = 0
        for i in range(1,631):
            label = math.ceil(i / 21)
            pca_test = np.dot(pca_w.T,test_vec[i-1])
            lda_test = np.dot(lda_w.T, test_vec[i-1])
            pred = get_1NN(pca_train,pca_test)
            if label == pred:
                pca_correct += 1
            pred = get_1NN(lda_train, lda_test)
            if label == pred:
                lda_correct += 1
        pca_acc.append(pca_correct/630 * 100)
        lda_acc.append(lda_correct/630 * 100)
    plt.figure()
    plt.xlabel('P')
    plt.ylabel('Accuracy')
    plt.plot(range(1,21),pca_acc, label='PCA')
    plt.plot(range(1,21),lda_acc, label='LDA')
    plt.xticks(range(1, 21))
    plt.legend()
    plt.savefig('Accuracy_vs_P.jpg')
    # print(pca_acc,lda_acc)
    with open('data.txt', 'wb') as fh:
        pickle.dump(pca_acc, fh)
        pickle.dump(lda_acc, fh)


if __name__ == '__main__':
    task1()
```

4.2 Task2

```
import os
os.environ['KMP_DUPLICATE_LIB_OK']='True'
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
from hw10 import *
import pickle

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 = 3  # [3, 8, 16]
acc = []
for p in [3, 8, 16]:
    training = False
    TRAIN_DATA_PATH = 'FaceRecognition/train/'
    EVAL_DATA_PATH = 'FaceRecognition/test/'
    LOAD_PATH = f'model_{p}.pt'
    OUT_PATH = 'out/'
#####################################

    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
        crr = 0
        for i in range(len(X_test)):
            pred = get_1NN(X_train.T, X_test[i])
            if pred == y_test[i]:
                crr = crr + 1
        acc.append(crr/630*100)
with open('data.txt', 'rb') as fh:
    pca_acc=pickle.load(fh)
    lda_acc=pickle.load(fh)
plt.figure()
plt.xlabel('P')
plt.ylabel('Accuracy')
plt.plot(range(1,21),pca_acc, label='PCA')
plt.plot(range(1,21),lda_acc, label='LDA')
plt.scatter([3, 8, 16],acc,label='autoencoder')
plt.xticks(range(1, 21))
plt.legend()
plt.savefig('task2.jpg')
        ####################################
```

4.3 Task3

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


def extract_feature(image):
    # two types of features:
    # 0 1 and 0
    #         1
    # image of size 40 width 20 height
    hfilter_halflen = np.arange(1,21)
    vfilter_halflen = np.arange(1,11)
    hfeatures = []
    vfeatures = []
    for i in hfilter_halflen:
        for y in range(image.shape[0]):   # loop through
            each row
            for x in range(image.shape[1]-(i*2)+1):     #
                loop through the row with hfilter
                left = np.sum(image[y,x:x+i+1]).astype(np
                    .int32)
                right = np.sum(image[y,x+i+1:x+(2*i)+1]).
                    astype(np.int32)
                hfeatures.append(right-left)
    hfeatures = np.array(hfeatures) # 8000
    for i in vfilter_halflen:
        for x in range(image.shape[1]):   # loop through
            each col
            for y in range(image.shape[0] - (i * 2) + 1):
                # loop through the col with vfilter
                top = np.sum(image[y:y+i+1, x]).astype(np
                    .int32)
                bot = np.sum(image[y+i+1:y+(2*i)+1, x]).
                    astype(np.int32)
                vfeatures.append(bot - top)
```

```
    vfeatures = np.array(vfeatures) # 4000
    return np.concatenate((hfeatures,vfeatures))


def save_feature():
    train_path_pos = 'CarDetection/train/positive/'
    train_path_neg = 'CarDetection/train/negative/'
    test_path_pos = 'CarDetection/test/positive/'
    test_path_neg = 'CarDetection/test/negative/'
    f = open('task3.txt', 'wb')
    features = []
    for img in os.listdir(train_path_pos):
        image = cv2.imread(train_path_pos+img,0)
        feature = extract_feature(image)
        features.append(feature)
    pickle.dump(features, f)
    features = []
    for img in os.listdir(train_path_neg):
        image = cv2.imread(train_path_neg+img, 0)
        feature = extract_feature(image)
        features.append(feature)
    pickle.dump(features, f)
    features = []
    for img in os.listdir(test_path_pos):
        image = cv2.imread(test_path_pos+img, 0)
        feature = extract_feature(image)
        features.append(feature)
    pickle.dump(features, f)
    features = []
    for img in os.listdir(test_path_neg):
        image = cv2.imread(test_path_neg+img, 0)
        feature = extract_feature(image)
        features.append(feature)
    pickle.dump(features, f)


def cascade_stage(features, labels,num_classifier=1):
    weights = np.ones(labels.shape)
    nPos = np.count_nonzero(labels)
    weights[:nPos] = 1/(2 * nPos)  # pos
    weights[nPos:] = 1/(2 * (labels.shape[0]-nPos))  #
        neg
    weights = weights/np.sum(weights)
    alpha = []
    classifiers = []
    sumalpha = np.zeros((features.shape[0],1))
```

```
    talpha = 0
    for i in range(num_classifier):
        idx,threshold,p,minErr,pred =
            cascade_weak_classifier(features, labels,
            weights) # [i,threshold,p,minErr,pred]
        print(features.shape,labels.shape,pred.shape)
        beta = minErr / (1-minErr)
        alpha = np.log(1/beta)
        classifiers.append([idx,threshold,p,minErr,pred,
            alpha])
        weights = weights * (beta ** (1-np.abs(pred-
            labels)))
        sumalpha = sumalpha + alpha * pred
        talpha = talpha + alpha * 0.5
        strong_pred = sumalpha >= talpha
        FP = np.sum(strong_pred[nPos:]==1) / (labels.
            shape[0]-nPos)
        FN = np.sum(strong_pred[:nPos]==0) / nPos
        if FP <= 0.5 and FN==0:
            break
    # misclassified
    negfeature_tmp = features[nPos:, :]
    negfeature_mc = negfeature_tmp[np.where(pred[nPos:]
        == 1), :][0].reshape(-1,features.shape[1])
    feature_left = np.vstack((features[:nPos, :],
        negfeature_mc))
    labels_left = np.vstack((labels[:nPos, :], np.zeros(
        len(negfeature_mc)).reshape(-1,1)))
    return feature_left, labels_left, FP, FN, classifiers


def cascade_weak_classifier(features, labels, weights):
# ref: https://engineering.purdue.edu/RVL/ECE661_2018/
    Homeworks/HW10/2BestSolutions/2.pdf
    # loop through features
    bestError = np.inf
    bestClassifier = None
    for i in range(features.shape[1]):
        sort_idx = np.argsort(features[:,i])
        sortedfeature = features[:,i][sort_idx]
        sortedlabel = labels[sort_idx]
        sortedweight = weights[sort_idx]
        pos_idx = np.where(sortedlabel == 1)
        neg_idx = np.where(sortedlabel == 0)
        sortedposw = np.zeros(sortedweight.shape)
```

```
        sortedposw[pos_idx] = sortedweight[pos_idx]  #
            pos left neg zero out
        sortednegw = np.zeros(sortedweight.shape)
        sortednegw[neg_idx] = sortedweight[neg_idx]  #
            neg left pos zero out
        TposW = np.sum(weights[np.where(labels==1)])
        TnegW = np.sum(weights[np.where(labels==0)])
        SposW = np.cumsum(sortedposw)
        SnegW = np.cumsum(sortednegw)
        # two types error
        err1 = (SposW+TnegW-SnegW).reshape(-1,1)
        err2 = (SnegW+TposW-SposW).reshape(-1,1)
        err = np.hstack((err1,err2))
        minIdx = np.unravel_index(err.argmin(), err.shape
            )
        minErr = err[minIdx]
        if minErr < bestError:
            bestError = minErr
            threshold = sortedfeature[minIdx[0]]
            p = 1 if minIdx[1]==0 else -1
            pred = features[:,i].reshape(-1,1) >=
                threshold if minIdx[1]==0 else features[:,
                i].reshape(-1,1) < threshold
            bestClassifier = [i,threshold,p,minErr,pred]
    return bestClassifier


def test_cascade(cascade,features):
    sumalpha = np.zeros((features.shape[0],1))
    talpha = 0
    for classifier in cascade:
        idx,threshold,p,minErr,pred,alpha = classifier
        prediction = features[:,idx].reshape(-1,1) >=
            threshold if p == 1 else features[:,idx].
            reshape(-1,1) < threshold
        sumalpha = sumalpha + alpha * 0.5
        talpha = talpha + alpha * 0.5
    return sumalpha >= talpha

def task3():
    with open('task3.txt', 'rb') as fh:
        train_pos = pickle.load(fh)
        train_neg = pickle.load(fh)
        test_pos = pickle.load(fh)
        test_neg = pickle.load(fh)
    train_pos = np.array(train_pos)      # 710
```

```
train_neg = np.array(train_neg)      # 1758
test_pos = np.array(test_pos)    # 178
test_neg = np.array(test_neg)    # 440
train = np.vstack((train_pos,train_neg))
train_label = np.vstack((np.ones((train_pos.shape
    [0],1)),np.zeros((train_neg.shape[0],1))))  # 2468
     x 1
# train = np.concatenate((train,train_label),axis=1)
test = np.vstack((test_pos, test_neg))
test_label = np.vstack((np.ones((test_pos.shape[0],
    1)), np.zeros((test_neg.shape[0], 1))))  # 618 x 1
# test = np.concatenate((test, test_label), axis=1)
features = train
labels = train_label
stages = []
fpl = []
fnl = []
fpr = 1
fnr = 1
for i in range(20):
    print('stage'+str(i), features.shape,labels.shape
        )
    features, labels, FP, FN, classifiers =
        cascade_stage(features,labels)
    stages.append(classifiers)
    fpr = fpr * FP
    fnr = fnr * FN
    print(fpr,fnr)
    fpl.append(fpr)
    fnl.append(fnr)
    if np.all(labels):
        break
# plt.figure()
# plt.plot(range(1,len(fpl)+1), fpl, label='FP')
# plt.plot(range(1,len(fpl)+1), fnl, label='FN')
# plt.ylabel('rate')
# plt.xlabel('stages')
# plt.savefig('performance.jpg')
labels = test_label
features = test
nPos = np.sum(labels == 1)
nNeg = np.sum(labels == 0)
TotalPos = nPos
TotalNeg = nNeg
nMisPos = 0
nCorNeg = 0
```

```
    fpl = []
    fnl = []
    for cascade in stages:
        prediction = test_cascade(cascade,features)
        nMisPos = nMisPos + nPos - np.sum(prediction[:
            nPos] == 1)
        nCorNeg = nCorNeg + nNeg - np.sum(prediction[nPos
            :] == 1)
        fpl.append(nMisPos/TotalPos)
        fnl.append(1-(nCorNeg/TotalNeg))
        features = features[np.where(prediction==1)][0]
        labels = labels[np.where(prediction==1)[0]]
        nPos = np.sum(labels == 1)
        nNeg = np.sum(labels == 0)
        if nPos == 0:
            break
    plt.figure()
    plt.plot(range(1,len(fpl)+1), fpl, label='FP')
    plt.plot(range(1,len(fpl)+1), fnl, label='FN')
    plt.ylabel('rate')
    plt.xlabel('stages')
    plt.savefig('task3.jpg')


if __name__ == '__main__':
    # save_feature()
    task3()
```