ECE 661 Computer Vision

Homework 10

Yi-Hong Liao | liao163@purdue.edu | PUID 0031850073

1. Logic

Implement PCA, LDA and autoencoder to classify human faces. Implement cascade Adaboost algorithm to detect cars.

2. Step

PCA

1. Vectorize each image.
2. Normalize each vectorized image to yield $\vec{x_i}$.
3. Calculate the covariance matrix

$$C = \frac{1}{N}\sum_{i=1}^{N}(\vec{x_i} - \vec{m})(\vec{x_i} - \vec{m})^T$$
$$= \frac{1}{N}\sum_{i=1}^{N} XX^T$$

   Where N is the total number of the images, $\vec{m}$ is the mean image vector
4. Perform eigen decomposition on XX^T then order its eigenvectors $\vec{u}$ according to their eigenvalues. Keep only p eigenvectors with p largest eigen value.
5. Compute the p largest eigenvectors using $\vec{w} = X\vec{u}$
6. Project the training and testing data onto the p dimensional space and use nearest neighbor to classify images.

LDA

1. Compute the global mean and class mean
2.

$$S_B = \frac{1}{|C|}\sum_{i=1}^{|C|}(\vec{m}_i - \vec{m})(\vec{m}_i - \vec{m})^T$$
$$S_w = \frac{1}{|C|}\sum_{i=1}^{|C|}\frac{1}{|C_i|}\sum_{k=1}^{|Ci|}\left(\vec{\mathrm{x}}_{\mathrm{k}}^{\mathrm{i}} - \vec{m}_i\right)\left(\vec{\mathrm{x}}_{\mathrm{k}}^{\mathrm{i}} - \vec{m}_i\right)^T$$

3. Eigendecomposite S_B and keep M eigenvectors with M largest eigenvalue. Let Y be the matrix formed by these eigenvectors.
4.

$$Y^T S_B Y = D_B$$

5.

$$Z = Y D_B^{-1/2}$$

6. Eigendecomposite $Z^T S_W Z$ to yield a matrix U of eigenvectors such that

$$U^T Z^T S_W Z U = D_W$$

Let $\widehat{U}$ denote the matrix of the eigenvectors retained from U

7.

$$W^T = \widehat{U}^T Z^T$$

8. Project the training and testing data onto the eigenvectors and use nearest neighbor to classify images.

Weak Classifier

1. Apply Haar vector filter in horizontal and vertical direction with various filter sizes to extract the feature from the images.
2. Loop through each feature and find the feature threshold and polarity that generates the minimum misclassification error.
3. Return the which feature is chosen and its threshold and polarity as the weak classifier.

AdaBoost

1. Using the training samples thrown up by the probability distribution $D_t(x_i)$, we select a weak classifier, denoted h_t.
2. Apply h_t to all of the training data.
3. Estimate the misclassification error ϵ_t.
4. Calculate the trust factor

$$\alpha_t = \frac{1}{2} ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$$

5. Update the $D_t(x_i)$

$$D_{t+1}(x_i) = \frac{D_t(x_i)e^{-\alpha_t y_i h_t(x_i)}}{Z_t}$$

6. Go back to step 1.
7. At the end of N iterations, construct the final classifier H as

$$H(x) = \sum_{t=1}^{N} \alpha_t\, h_t(x)$$

If $H > d$, it’s label 1, otherwise 0.

Classifier Cascade

1. Run AdaBoost at each stage of the cascade to meet to requirement TP: 0.99 and FP:0.3
2. Pass the detected true items to the next detector stage.
3. Cascade the stages to meet to requirement TP > 0.9, FP < 10e-6.

3. Result

- **Task 1 & Task 2**

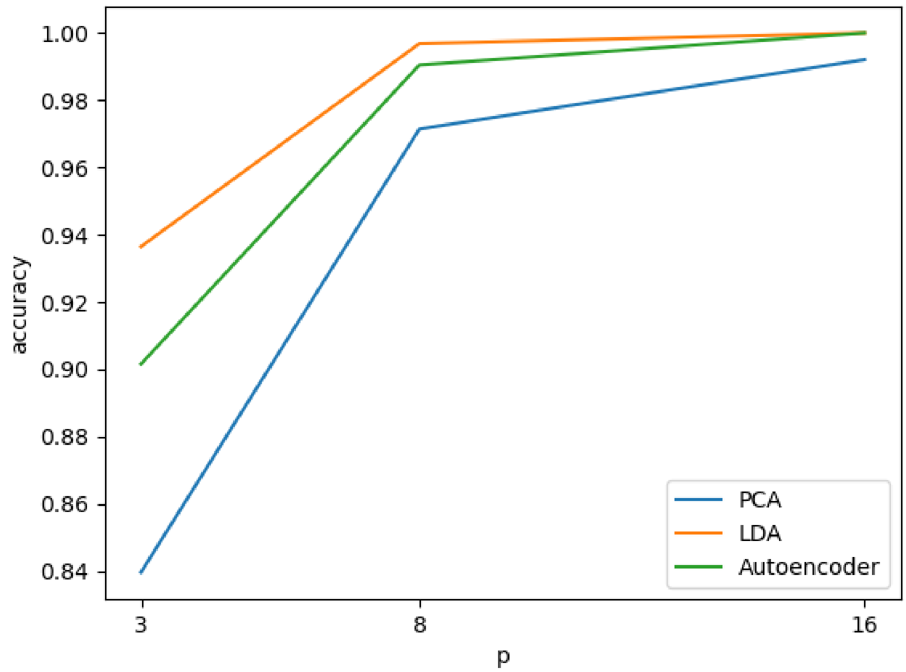

Figure 1: Classification accuracy as a function of p.

As p increases, the accuracy increases. The PCA classifier gives the worst result. LDA and autoencoder converge to 100 % accuracy and LDA method converge faster than the autoencoder method.

- **Task 3**

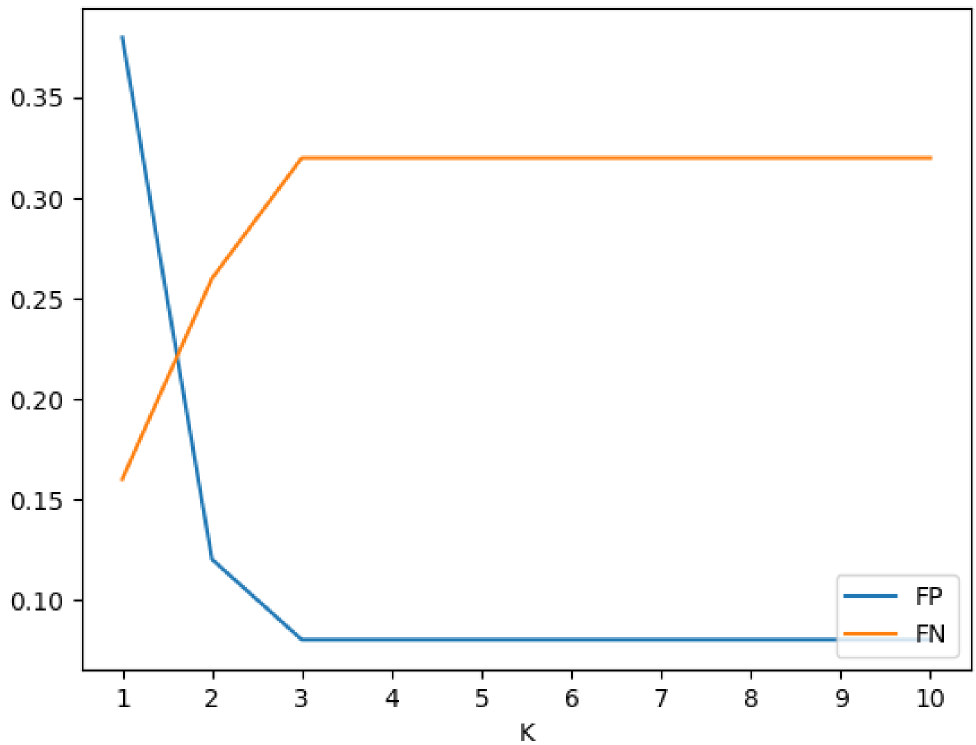

Figure 2: False positive (FP) rate and false negative (FN) rate after first 10 stages of the cascade.

For task 3, I try to achieve TP rate 0.99 and FP rate 0.3 for every stage in the cascade. However, when I set the threshold d to satisfy the TP rate requirement on each stage, the FP rate decreases extremely slowly, especially if there is a lot of training data. This is probably because the number of feature vectors I extract is not enough or the feature vectors can't properly distinguish the images. Therefore, I only manage to use 100 images in training and testing dataset. In this case, the result in Figure 2 shows that the FP rate goes down and FN rate increases as we expected.

4. Source Code

Task 1 & 2

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

# In[1]:


import numpy as np
import matplotlib.pyplot as plt
import cv2
import math
from scipy.spatial import distance
from scipy.linalg import null_space
from pathlib import Path
import os
from os import listdir
import re

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


# In[2]:


def load_dataset(folder_dir):
    # get the path/directory
    images = []
    labels = []
    i = 0;
    for img in os.listdir(folder_dir):
        # check if the image ends with jpg
        if img.endswith(".png"):
            image = cv2.imread(folder_dir+'\\'+img)

            if(image is not None):
                label = img.split('_')[0]
                if len(image.shape) > 2:
                    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
                image_vec = image.flatten()

                # normalize images
                image_vec = image_vec / np.linalg.norm(image_vec)

                images.append(image_vec)
                labels.append(label)
#                   i = i+1
#                   if i % 20 == 0:
#                       print(i)
```

```
    return np.transpose(np.stack(images)), labels

def PCA(images, p):
    N = images.shape[1]
    m = np.mean(images, axis=1)
    xm = np.transpose(np.transpose(images) - m)
    CT = (1/N)*np.transpose(xm).dot(xm)

    w, v = np.linalg.eig(CT)
    sorted_v = np.transpose([vp for _, vp in sorted(zip(w , v), key =
lambda wv: wv[0], reverse = True)])

    eig_vecs = xm.dot(sorted_v)
    eig_vecs = eig_vecs / np.linalg.norm(eig_vecs)

    return eig_vecs[:, 0:p], m

def LDA(images, labels, p):
    n = images.shape[0]

    # global mean
    m = np.mean(images, axis=1)

    # in class mean
    mi = []
    SW_vec = []
    unique_labels = np.unique(labels)
    C = unique_labels.shape[0]
    for unique_label in unique_labels:
        sum_image = np.zeros(n)
        ci = 0
        for i, label in enumerate(labels):
            if label == unique_label:
                ci += 1
                sum_image += images[:, i]
        mi_tmp = sum_image / ci
        for i, label in enumerate(labels):
            if label == unique_label:
                SW_vec.append(images[:, i] - mi_tmp)
        mi.append(mi_tmp)
    SW_vec = np.transpose(SW_vec)
    mi = np.transpose(np.stack(mi))


    xm = np.transpose(np.transpose(mi) - m)
    SBT = (1/C)*np.transpose(xm).dot(xm)

    w, v = np.linalg.eig(SBT)
    sorted_w = sorted(w , reverse = True)
    sorted_v = np.transpose([vp for _, vp in sorted(zip(w , v), key =
lambda wv: wv[0], reverse = True)])

    eig_vecs0 = xm.dot(sorted_v)
    eig_vecs0 = eig_vecs0 / np.linalg.norm(eig_vecs0)


    Y = eig_vecs0[:, :-1]
```

```
    DB = np.eye(C-1) * sorted_w[:-1]
    Z = Y.dot(np.sqrt(np.linalg.inv(DB)))

    ZT_SW_Z = np.dot(np.dot(np.transpose(Z), SW_vec),
np.transpose(np.dot(np.transpose(Z), SW_vec)))

    w, v = np.linalg.eig(ZT_SW_Z)
    sorted_w = sorted(w , reverse = True)
    sorted_v = np.transpose([vp for _, vp in sorted(zip(w , v), key =
lambda wv: wv[0], reverse = True)])

    eig_vecs1 = Z.dot(sorted_v)
    eig_vecs1 = eig_vecs1 / np.linalg.norm(eig_vecs1)

    return eig_vecs1[:, 1:p+1], m



def project_to_subspace(ws, images, m):
    images = np.transpose(np.transpose(images) - m)
    return np.transpose(ws).dot(images)

def predict_labels(images, trainImgs, trainLabels, ws, m):
    N = images.shape[1]
    trainVecs = np.transpose(project_to_subspace(ws, trainImgs, m))
    testVecs = project_to_subspace(ws, images, m)
    predicted_labels = []
    for i in range(N):
        testVec = testVecs[:, i]
        dists = np.sqrt(np.sum((trainVecs - testVec) ** 2, 1))
        predicted_label = trainLabels[np.argmin(dists)]
        predicted_labels.append(predicted_label)
    return predicted_labels

def calculate_accuracy(gt_labels, pred_labels):
    N = len(pred_labels)
    correct = 0
    for i in range(N):
        if pred_labels[i] == gt_labels[i]:
            correct += 1
    return correct/N



# In[3]:


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)))

def predict_labels_autoenc(X_test, X_train, y_train):
    N = X_test.shape[0]
```

```
    predicted_labels = []
    for i in range(N):
        testVec = X_test[i, :]
        dists = np.sqrt(np.sum((X_train - testVec) ** 2, 1))
        predicted_label = trainLabels[np.argmin(dists)]
        predicted_labels.append(predicted_label)
    return predicted_labels


# In[4]:


if __name__ == '__main__':
    path = Path("C:/Users/yhosc/Desktop/ECE661/HW10/")
    outputPath = Path("C:/Users/yhosc/Desktop/ECE661/HW10")

    trainImgs, trainLabels = load_dataset(str(path / "FaceRecognition" /
"train"))
    testImgs, testLabels = load_dataset(str(path / "FaceRecognition" /
"test"))

    ps = [3, 8, 16]
    accuracies_PCA = []
    accuracies_LDA = []

    for p in ps:
        ws, m = PCA(trainImgs, p)
        predicted_labels = predict_labels(testImgs, trainImgs,
trainLabels, ws, m)
        accuracy = calculate_accuracy(testLabels, predicted_labels)
        accuracies_PCA.append(accuracy)

        ws, m =  LDA(trainImgs, trainLabels, p)
        predicted_labels = predict_labels(testImgs, trainImgs,
trainLabels, ws, m)
        accuracy = calculate_accuracy(testLabels, predicted_labels)
        accuracies_LDA.append(accuracy)

    print(accuracies_PCA)
    print(accuracies_LDA)


# In[5]:


# Change these
ps = [3, 8, 16]
training = False
TRAIN_DATA_PATH = str(path / "FaceRecognition" / "train")
EVAL_DATA_PATH = str(path / "FaceRecognition" / "test")
OUT_PATH = str(path)

accuracies_autoenc = []
for p in ps:
    LOAD_PATH = str(path) + f'/weights/model_{p}.pt'
    model = Autoencoder(p)
```

```python
    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)

    predicted_labels = predict_labels_autoenc(X_test, X_train, y_train)
    accuracy = calculate_accuracy(testLabels, predicted_labels)
    accuracies_autoenc.append(accuracy)
print(accuracies_autoenc)


# In[10]:


plt.plot(ps, accuracies_PCA, ps, accuracies_LDA, ps, accuracies_autoenc)
plt.xticks(ps)
plt.xlabel('p')
plt.ylabel('accuracy')
plt.legend(["PCA", "LDA", "Autoencoder"], loc ="lower right")
plt.savefig('classification.png')
plt.show()
```

Task 3

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

# In[248]:


import numpy as np
import matplotlib.pyplot as plt
import cv2
import math
from scipy.spatial import distance
from scipy.linalg import null_space
from pathlib import Path
import os
from os import listdir
import re
from sklearn import preprocessing
import pickle


# In[249]:


def extract_features(folder_dir, label, num_data):
    features = []
    labels = []
    n = 0;
    for img in os.listdir(folder_dir):
        # check if the image ends with jpg
        if img.endswith(".png"):
            image = cv2.imread(folder_dir+'\\'+img)
            if(image is not None):
                if len(image.shape) > 2:
                    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
                feature = []
                h, w = image.shape[0], image.shape[1]
                f_ws = range(2, int(w/2), 2)
                f_hs = range(2, int(h/2), 2)

#                 f_ws = [4]
#                 f_hs = [4]

                # extract horizontal features
                for f_w in f_ws:
                    for j in range(h):
                        for i in range(w - f_w + 1):
                            neg = np.sum(image[j,
i:int((i+f_w)/2)]).astype(np.int32)
                            pos = np.sum(image[j,
int((i+f_w)/2):(i+f_w)]).astype(np.int32)
                            feature.append(pos-neg)

                # extract vertical features
                for f_h in f_hs:
```

```
                    for i in range(w):
                        for j in range(h - f_h + 1):
                            neg = np.sum(image[j:int((j+f_h)/2),
i]).astype(np.int32)
                            pos = np.sum(image[int((j+f_h)/2):(j+f_h),
i]).astype(np.int32)
                            feature.append(pos-neg)

                features.append(feature)
                labels.append(label)
                n += 1
                if n % 200 == 0:
                    print(n)
                if num_data != -1:
                    if n == num_data:
                        break

    features = np.array(features)
    labels = np.array(labels)
#     features = preprocessing.normalize(features, axis = 0)
#     print(np.sum(features[:, 0]**2))

    return features, labels

def find_best_weak_classifier(features, labels, Dts, used_f_idx):

    best_classifier = []

    min_e = 2
    for f in range(len(features[0])):
        if f not in used_f_idx:
            feature = features[:, f]

            # sort features
            sorted_feature = np.array(sorted(feature))
            sorted_labels = np.array([x for _, x in sorted(zip(feature,
labels))])
            sorted_Dts = np.array([x for _, x in sorted(zip(feature,
Dts))])

            # calculate values for errors
            pos_dts = sorted_Dts*sorted_labels
            neg_dts = sorted_Dts*(1-sorted_labels)

            SP = np.cumsum(pos_dts)
            SN = np.cumsum(neg_dts)
            TP = np.sum(pos_dts)
            TN = np.sum(neg_dts)

            error_1 = SP + TN - SN
            error_2 = SN + TP - SP

            curr_min_e1 = np.min(error_1)
            curr_min_e2 = np.min(error_2)
            curr_min_e = np.minimum(curr_min_e1, curr_min_e2)

            if curr_min_e < min_e:
```

```
                min_e = curr_min_e
                f_idx = f
                if curr_min_e1 < curr_min_e2:
                    polarity = 1
                    threshold = sorted_feature[np.argmin(error_1)]
                    htx = feature > threshold
                else:
                    polarity = -1
                    threshold = sorted_feature[np.argmin(error_2)]
                    htx = feature <= threshold

    best_classifier = [f_idx, threshold, polarity, min_e, htx*1]

    return best_classifier


def adaboost(all_features, all_true_labels, true_idx, TP_target,
FP_target, max_iter):
    features = all_features[true_idx, :]
    true_labels = all_true_labels[true_idx]

    N = len(features)
    Dts = (1/N)*np.ones(N)

    alphas = []
    classifiers = []
    used_f_idx = []
    TP = 0
    FP_rate = 1
    for t in range(max_iter):
        print(t)
        best_classifier = find_best_weak_classifier(features, true_labels,
Dts, used_f_idx)
        classifiers.append(best_classifier)

        f_idx, threshold, polarity, min_e, htx = best_classifier
        used_f_idx.append(f_idx)
#         print("f: ", f_idx, " min_e: ", min_e)

        # apply all the weak classifiers
        d, TP_rate, pred_labels, reach_TP_target =
apply_classifiers_with_targetTP(features, classifiers, alphas,
true_labels, TP_target)
#         d, TP_rate, pred_labels, reach_TP_target =
apply_classifiers(features, classifiers, alphas, true_labels, TP_target)
#         print("d: ", d)

        N = np.sum(true_labels == 0)
        if N == 0:
            break
        FP = np.sum(np.logical_and(pred_labels == 1, true_labels == 0))
        FP_rate = FP / N

        print("TP_rate: ", TP_rate)
        print("FP_rate: ", FP_rate)

#         if reach_TP_target:
```

```
        if FP_rate < FP_target:
            break

        # update probability distribution
        # compute confidence parameters
        alpha = math.log((1-min_e) / (min_e + 1e-10))
        alphas.append(alpha)

#         numerator = Dts*np.exp(-alpha*true_labels*htx)
        numerator = Dts*np.exp(-2*alpha*(-2*abs(true_labels-htx)+1))
        Dts = numerator / np.sum(numerator)

    return classifiers, alphas, d, FP_rate

def create_cascade(features, labels, num_stages):
    TP_target = 0.95
    FP_target = 0.5
    new_idx = np.arange(features.shape[0])
    cascades = []
    classifications = np.zeros(features.shape[0])
    FPs = []
    FNs = []
    accuracies = []
    for k in range(num_stages):
        classifiers, alphas, d, FP_rate = adaboost(features, labels,
new_idx, TP_target, FP_target, 100)
        cascades.append([classifiers, alphas, d, FP_rate])
        new_classifies = apply_classifiers_with_d(features, new_idx,
classifiers, alphas, d)
        classifications[new_idx] = new_classifies
        new_idx = new_idx[np.where(new_classifies == 1)[0]]
        print("true_idx: ", len(new_idx))

        P = np.sum(labels == 1)
        N = np.sum(labels == 0)
        FP = np.sum(np.logical_and(classifications == 1, labels == 0))
        FN = np.sum(np.logical_and(classifications == 0, labels == 1))
        FP_rate = FP / N
        FN_rate = FN / P
        accuracy = calculate_accuracy(labels, classifications)

        FPs.append(FP_rate)
        FNs.append(FN_rate)
        accuracies.append(accuracy)
        print("FP: ", FP_rate, " FN: ", FN_rate, " accuracy: ", accuracy)

        ################
#         classifiers, alphas, d, FP_rate = adaboost(features, labels,
new_idx, TP_target, FP_target, 10)
#         cascades.append([classifiers, alphas, d, FP_rate])
#         new_classifies = apply_classifiers_with_d(features, new_idx,
classifiers, alphas, d)
#         classifications[new_idx] = new_classifies
#         neg_idx = new_idx[np.where(new_classifies == 0)[0]]
#         new_idx = new_idx[np.where(new_classifies == 1)[0]]

#         print("true_idx: ", len(new_idx))
```

```
#         print("neg_idx: ", len(neg_idx))

#         P = np.sum(labels == 1)
#         N = np.sum(labels == 0)
#         FP = np.sum(np.logical_and(classifications == 1, labels == 0))
#         FN = np.sum(np.logical_and(classifications == 0, labels == 1))
#         FP_rate = FP / N
#         FN_rate = FN / P
#         accuracy = calculate_accuracy(labels, classifications)

#         FPs.append(FP_rate)
#         FNs.append(FN_rate)
#         accuracies.append(accuracy)
#         print("FP: ", FP_rate, " FN: ", FN_rate, " accuracy: ",
accuracy)
    return cascades

def test_cascade(features, labels, cascades):
    new_idx = np.arange(features.shape[0])
    classifications = np.zeros(features.shape[0])
    FPs = []
    FNs = []
    accuracies = []
    Ks = []
    K = 0
    for cascade in cascades:
        K += 1
        classifiers, alphas, d, FP_rate = cascade
        new_classifies = apply_classifiers_with_d(features, new_idx,
classifiers, alphas, d)
        classifications[new_idx] = new_classifies
        new_idx = new_idx[np.where(new_classifies == 1)[0]]

        P = np.sum(labels == 1)
        N = np.sum(labels == 0)
        FP = np.sum(np.logical_and(classifications == 1, labels == 0))
        FN = np.sum(np.logical_and(classifications == 0, labels == 1))
        FP_rate = FP / N
        FN_rate = FN / P
        accuracy = calculate_accuracy(labels, classifications)

        FPs.append(FP_rate)
        FNs.append(FN_rate)
        accuracies.append(accuracy)
        Ks.append(K)
    return FPs, FNs, accuracies, Ks


def apply_classifiers_with_targetTP(features, classifiers, alphas,
true_labels, TP_target):
    features = np.array(features)
    sumHs = np.zeros(features.shape[0])
    for classifier, alpha in zip(classifiers, alphas):
        f_idx, threshold, polarity, min_e, htx = classifier
        feature = features[:, f_idx]
        if polarity == 1:
            sumHs += alpha * (feature > threshold)*1
```

```
        else:
            sumHs += alpha * (feature <= threshold)*1
    print("zeros 1: ", np.sum(np.logical_and(true_labels == 1, sumHs ==
0)))
    reach_target = False
    d = 0
    max_TP_rate = 0
    min_diff = 1
    cand_ds = sumHs
    for cand_d, true_label in zip(cand_ds, true_labels):
        if true_label == 1:
            pred_labels = (sumHs >= cand_d)*1
            P = np.sum(true_labels == 1)
            TP = np.sum(np.logical_and(pred_labels == 1, true_labels ==
1))
            TP_rate = TP / P
#             if abs(TP_rate-TP_target) < min_diff:
#                 min_diff = abs(TP_rate-TP_target)
#                 max_TP_rate = TP_rate
#                 d = cand_d
#                 reach_target = True
            if TP_rate > max_TP_rate:
                max_TP_rate = TP_rate
                d = cand_d
                if TP_rate >= TP_target:
                    reach_target = True

    pred_labels = (sumHs >= d)*1
    print("d: ", d)
    return d, max_TP_rate, pred_labels, reach_target

def apply_classifiers_with_d(all_features, true_idx, classifiers, alphas,
d):

    features = all_features[true_idx, :]

    sumH = np.zeros(features.shape[0])
    for classifier, alpha in zip(classifiers, alphas):
        f_idx, threshold, polarity, min_e, htx= classifier
        feature = features[:, f_idx]
        if polarity == 1:
            sumH += alpha * (feature > threshold)*1
        else:
            sumH += alpha * (feature <= threshold)*1

    classifications = (sumH >= d)*1
    return classifications

def apply_classifiers(features, classifiers, alphas, true_labels,
TP_target):
    features = np.array(features)
    sumH = np.zeros(features.shape[0])
    reach_TP_target = False

    for classifier, alpha in zip(classifiers, alphas):
        f_idx, threshold, polarity, min_e, htx= classifier
        feature = features[:, f_idx]
```

```
        if polarity == 1:
            sumH += alpha * (feature > threshold)*1
        else:
            sumH += alpha * (feature <= threshold)*1

    sum_alpha = np.sum(alphas)
    d = 0.5*sum_alpha
    pred_labels = (sumH >= d)*1

    P = np.sum(true_labels == 1)
    TP = np.sum(np.logical_and(pred_labels == 1, true_labels == 1))
    TP_rate = TP / P
    if TP_rate >= TP_target:
        reach_TP_target = True

    return d, TP_rate, pred_labels, reach_TP_target

def calculate_accuracy(gt_labels, pred_labels):
    N = len(pred_labels)
    correct = 0
    for i in range(N):
        if pred_labels[i] == gt_labels[i]:
            correct += 1
    return correct/N





# In[250]:


if __name__ == '__main__':
    path = Path("C:/Users/yhosc/Desktop/ECE661/HW10/")
    outputPath = Path("C:/Users/yhosc/Desktop/ECE661/HW10")

    if os.path.exists(str(path / "train_data.pickle")):
        print("load training data")
        with open('train_data.pickle', 'rb') as handle:
            train_features, train_labels = pickle.load(handle)
    else:
        print("creating training data...")

        neg_features, neg_labels = extract_features(str(path /
"CarDetection" / "train" / "negative"), 0, 50)
        pos_features, pos_labels = extract_features(str(path /
"CarDetection" / "train" / "positive"), 1, 50)
        train_features = np.concatenate((neg_features, pos_features))
        train_labels = np.concatenate((neg_labels, pos_labels))

        with open('train_data.pickle', 'wb') as handle:
            pickle.dump([train_features, train_labels], handle,
protocol=pickle.HIGHEST_PROTOCOL)
        print("training data saved")

    if os.path.exists(str(path / "test_data.pickle")):
```

```
        print("load testing data")
        with open('test_data.pickle', 'rb') as handle:
            test_features, test_labels = pickle.load(handle)
    else:
        print("creating testing data...")
        neg_features, neg_labels = extract_features(str(path /
"CarDetection" / "test" / "negative"), 0, 50)
        pos_features, pos_labels = extract_features(str(path /
"CarDetection" / "test" / "positive"), 1, 50)
        test_features = np.concatenate((neg_features, pos_features))
        test_labels = np.concatenate((neg_labels, pos_labels))

        with open('test_data.pickle', 'wb') as handle:
            pickle.dump([test_features, test_labels], handle,
protocol=pickle.HIGHEST_PROTOCOL)
        print("testing data saved")

    if os.path.exists(str(path / "cascades.pickle")):
        print("load cascades")
        with open('cascades.pickle', 'rb') as handle:
            cascades = pickle.load(handle)
    else:
        cascades = create_cascade(train_features, train_labels, 10)
        with open('cascades.pickle', 'wb') as handle:
            pickle.dump(cascades, handle,
protocol=pickle.HIGHEST_PROTOCOL)

    FPs, FNs, accuracies, Ks = test_cascade(test_features, test_labels,
cascades)
    print(FPs, FNs, accuracies, Ks)


# In[251]:


plt.plot(Ks, FPs, Ks, FNs)
plt.xticks(Ks)
plt.xlabel('K')
plt.legend(["FP", "FN"], loc ="lower right")
plt.savefig('adaboost.png')
plt.show()
```