ECE 661: Homework 10

Shane Fernandes
ferna202@purdue.edu

December 9, 2022

1 Logic

This homework consists of two parts:

1. PCA, LDA and Autoencoder for dimensionality reduction and nearest neighbor for face recognition.
2. Cascaded AdaBoost Classifiers for object detection.

1.1 PCA, LDA and Autoencoders for Dimensionality Reduction

1.1.1 Principal Component Analysis (PCA)

PCA is dimensionality reduction technique in which we transform the original dataset into a new coordinate system, where the axes are the principal components of the data. The principal components are the directions in the data that capture the most variance. By projecting the data onto these axes, we can reduce the dimensionality of the data.
The steps to perform PCA are as follows:

1. Read each image, convert into a column vector (i.e. $m \times n$ sized image converts to an $mn \times 1$ sized vector), and normalize each image vector to unit magnitude.
2. Calculate the global mean vector $\vec{m}$ of all N images:

$$\vec{m} = \frac{1}{N} \sum_{i=1}^{N} x_i$$

   where x_i is a normalized image vector
3. Find the covariance matrix by

$$C = \frac{1}{N} \sum_{i=1}^{N} x_i x_i^T$$

4. Calculate eigen-values from the Covariance matrix (use the computational trick when Covariance matrix size is too large, also remember to normalize these eigen vectors after using the trick).
5. Retain first P sorted eigen-vectors: $W_P = [w_1|w_2|...|w_p]$
6. Now we project all the training images onto this feature subspace using:

$$Y = W_P^T (X - \vec{m})$$

7. For classification, we can project each of the testing vectors onto the feature subspace using the above formula.
8. Here we are using a 1-NearestNeighbor classifier, hence we calculate the L2-norm between each projected testing vector and each element in the training feature subspace. The training label associated to the least distance for each test vector is our predicted test label.
9. For accuracy, we just compare each predicted test label and ground truth test label.

1.1.2 Linear Discriminant Analysis (LDA)

The goal of LDA is to find the directions in the underlying vector space that are maximally discriminating between the classes.

1. For C classes, we define the between-class scatter by

$$S_b = \frac{1}{|C|}\sum_{1}^{|C|}(\vec{m_i} - \vec{m})(\vec{m_i} - \vec{m})^T$$

   where $\vec{m_i}$ and $\vec{m}$ are the in-class and global means respectively.

2. The within-class scatter is given by

$$S_w = \frac{1}{|C|}\sum_{i=1}^{|C|}\frac{1}{|C_i|}\sum_{k=1}^{|C_i|}(\vec{x_k^i} - \vec{m_i})(\vec{x_k^i} - \vec{m_i})^T$$

3. The goal is to find the eigen values that are maximally discriminating between the classes, by maximizing the ratio between within-class and between-class scatter of the feature vectors.

4. This can be expressed mathematically by finding the eigenvectors that maximize the Fisher Discriminant function:

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

5. The eigen-vectors are found using the Yu-Yang method and steps 5-9 from PCA are repeated.

1.2 Cascaded AdaBoost Classifiers for object detection

The Viola-Jones method was used to create a cascaded AdaBoost classifer. It consists of the following steps:

1. **Feature Extraction**:

   - Haar filters of different sizes in horizontal and vertical orientations were used for feature extraction.
   - The filters are of the sizes 1×2 to $1 \times N$ for horizontal features and 2×1 to $N \times 1$ for the vertical features (M and N are the width and height of the image respectively).
   - A horizontal and vertical haar filter for M=4 and N=4 is of the format:

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

     and

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

   - While applying these filters, we use integral images for efficiency.

2. **Creating a Weak Classifier**:

   - Before creating a weak classifier, we need to initialize a weight matrix where the first M(number of positive samples) values are $\frac{1}{2M}$ and the next N(number of negative samples) values are $\frac{1}{2M}$
   - We can begin creating the weak classifier by first normalizing the weights. Then we sort each feature in ascending order across all images to separate the set into two classes with ¿ 50% accuracy. We use the feature values as thresholds.

- Calculate the error for a threshold and use the least error as the threshold. The error is given by

$$error = (S^+ + (T^- - S^-), S^- + (T^+ - S^+))$$

where the first value in the bracket is if polarity is 1 and second is if polarity is -1. In the above equation

  - S^+ : sum of weights of positive examples with feature value greater than the threshold
  - S^- : sum of weights of negative examples with feature value less than the threshold
  - T^+ : sum of weights of positive examples
  - T^- : sum of weights of negative examples

- The feature with the minimum error value is selected and its corresponding index, feature value, polarity, and error is returned as a list.
- We iteratively check for the feature with least error values and return the best weak classifier.

3. **Cascading Weak Classifiers to make a strong classifier**:

- We combine many weak classifiers to form a strong classifier in a stage.
- For each best weak classifier found, we find a few parameters which help us find the next best weak classifier.
-

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

and

$$\alpha = \ln \frac{1}{\beta}$$

where ϵ is the error associated with that classifier.

- We then update the weight matrix for the next classifier by multiplying it with $\beta^{1-\delta}$, where δ is the difference between the label and the prediction.
- We then compute the predictions of the best weak classifier in a stage by multiplying α with the predictions which is the intermediate output of the cascade. We also keep track of all alphas calculated for the number of weak classifiers in a cascade with the classifier values.
- We then determine the final classifications by thresholding the product calculated above using the α list.
- We also calculate the false positive and negative rates and return the output of the final best weak classifier in the stage. We stop whenever we satisfy a stopping criterion related to the FPR and TPR At the end of each stage, we remove the correctly classified negative examples from current set.

4. **Using Stages to combine strong classifiers**:

- We aggregate strong classifiers to achieve very low FPR.
- The maximum number of stages were set to 10 and the training was stopped at a stage which did not have any more false postives left.

5. **Testing**:

- For testing we simply propagate each example through each classifier in each cascade and adjust the outputs using the α values from each classifier and also threshold it using the same method as in the training phase.
- We also have to remove the correctly classified negative examples from current set to accurately measure the FPR as a function of stages.

- To calculate FPR and FNR we use the following formulae:

$$FPR = \frac{\#\text{ of misclassified negative image}}{\#\text{ of negative images}}$$

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

2 Results & Observations

2.1 Task 1 & 2

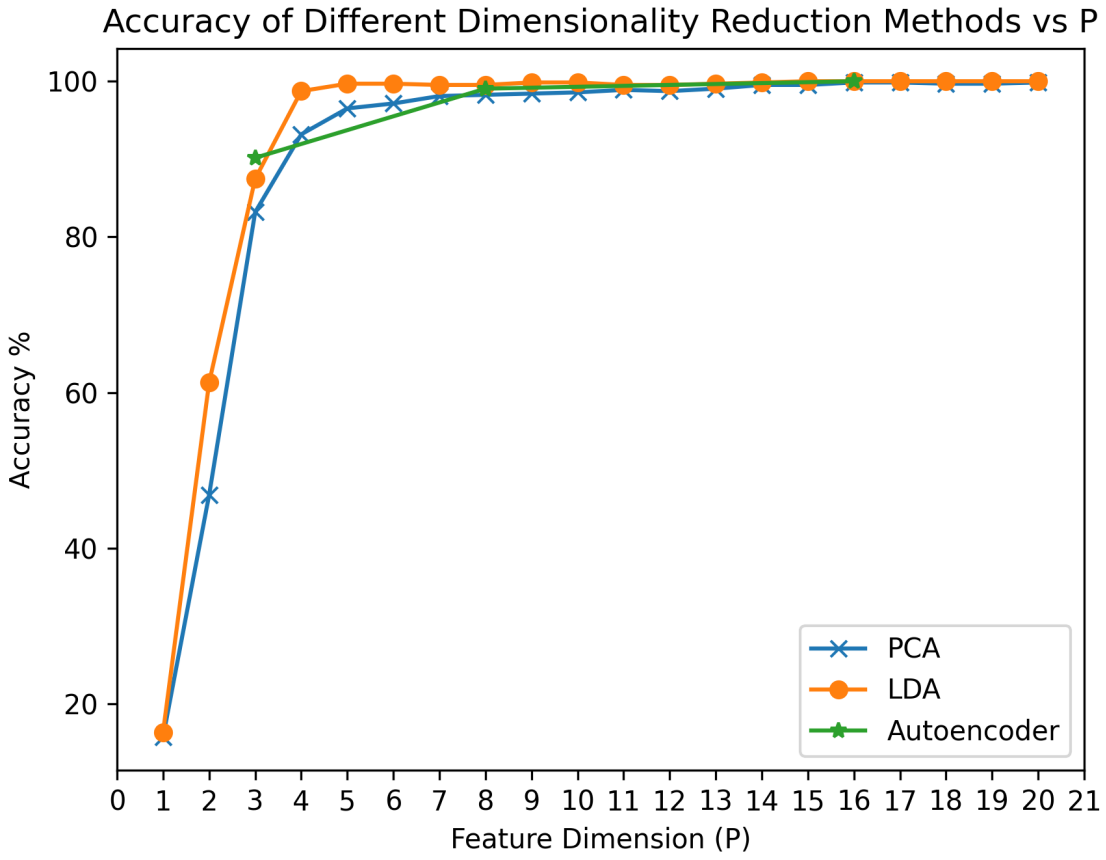

Figure 1: Accuracy Plot of Different Dimensionality Reduction Methods vs Feature Dimension Size

- The performance of PCA and LDA is comparable at higher dimensionality but at lower dimensionality LDA tends to perform better.
- The results are in agreement with theoretical expectations that LDA performs better than PCA, and reaches 100% classification accuracy before PCA while increasing the dimensionality
- For the autoencoder approach we observe better results than PCA and LDA at lower dimensionalities while it reaches comparable levels of performance at higher dimensionalities.

2.2 Task 3

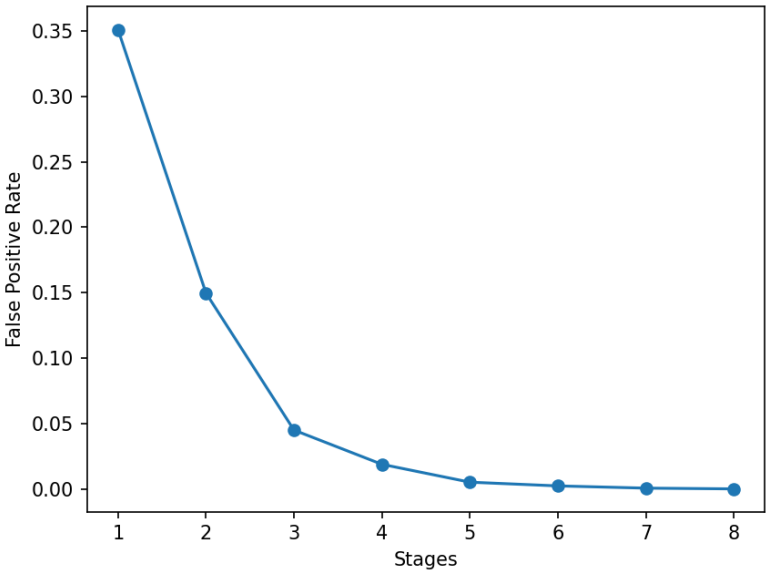

Figure 2: FPR during Training vs Number of Stages

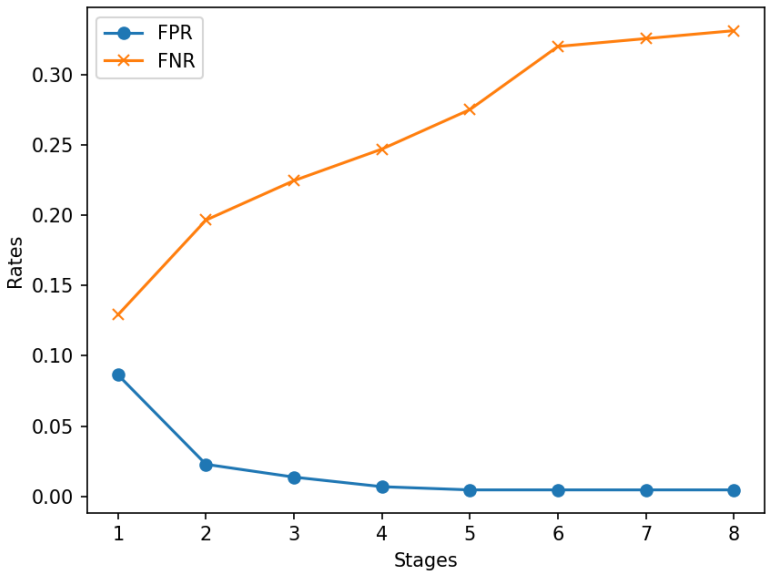

Figure 3: FPR and FNR during Testing vs Number of Stages

- We can observe that during training as we increase the number of stages the FPR decreases till we reach FPR = 0.
- During testing we can see that we reach a very low FPR with more stages but this comes at the cost of increasing FNR. This is very design specific and this tradeoff can be tuned in the program.

3 Code Listing

Task 1 & 2

```
import numpy as np
import cv2
import glob
import os
import matplotlib.pyplot as plt
from matplotlib import rcParams
import autoencoder


def readImages(root_dir, sub_dir):
    # saves images to npz format if reading for the first time, else loads the images
    ↪  from the npz file
    if not os.path.exists(f'{root_dir}_{sub_dir}.npz'):
        img_list = []
        label_list = []
        img_path_list = glob.glob(os.path.join(root_dir, sub_dir, f'*'))
        for pth in img_path_list:
            # read image and append to array
            im_flat = cv2.imread(pth, 0).flatten()
            img_list.append(im_flat)
            # create array for labels
            f_name = os.path.split(pth)[-1]
            f_name = os.path.splitext(f_name)[0]
            c_num = int(f_name.split('_')[0])
            label_list.append(c_num)

        np.savez_compressed(f'{root_dir}_{sub_dir}', x=np.array(img_list),
        ↪  y=np.array(label_list))

    imgs = np.load(f'{root_dir}_{sub_dir}.npz', allow_pickle=True)
    return imgs['x'], imgs['y']


def normalize(x):
    x = np.divide(x, np.expand_dims(np.linalg.norm(x, axis=1), axis=-1))
    return x


def calcPCA(x, p=None):
    x = np.transpose(x)  # arranged as column vectors
    m = np.mean(x, axis=1, keepdims=True)
    x = x - m

    # using computational trick
    cov = np.dot(np.transpose(x), x)  # calc X'X
    _, _, ut = np.linalg.svd(cov)  # eigen vecs of X'X

    # multipy by X to get eigenvecs of XX'
    w = np.dot(x, ut)
    w = normalize(w)
```

```
    # retain first p eigen vecs
    w_p = w[:, :p]

    return w_p, m


def calcLDA(x, y, p):
    num_clss = np.max(y)
    dat_per_class = np.zeros(num_clss)
    for val in y: dat_per_class[val - 1] += 1  # count number of samples in each class
    cl_ms = np.zeros((num_clss, x.shape[1]))

    # compute per class means
    for data, lbl in zip(x, y): cl_ms[lbl - 1] += data
    cl_ms = np.transpose(cl_ms / np.expand_dims(dat_per_class, axis=1))
    x = np.transpose(x)
    m = np.mean(cl_ms, axis=1, keepdims=True)  # global mean
    m_i = cl_ms - m
    x_i = x - m

    # -----yu-yang method-----#
    # using computational trick
    sb = np.dot(np.transpose(m_i), m_i)
    _, d, ut = np.linalg.svd(sb)  # eigen vecs of X'X

    # multipy by X to get eigenvecs of XX'
    w = np.dot(m_i, ut)
    w = normalize(w)

    # remove eigen-vec corresponding to eigenval closest to 0; last one in this case
    y = w[:, :-1]
    D = d[:-1]

    D_b = np.diag(D)
    Z = np.dot(y, np.sqrt(np.linalg.inv(D_b)))
    zt_x = np.dot(np.transpose(Z), x_i)  ##
    sw = np.dot(zt_x, np.transpose(zt_x))
    _, d, u_t = np.linalg.svd(sw)

    # multipy by X to get eigenvecs of XX'
    w = np.dot(Z, u_t)
    w = normalize(w)

    # retain first p eigen vecs
    w_p = w[:, :p]
    return w_p, m


# 1 nearest neighbor classifier
def oneNN(ft_space, p_x_test, y_train):
    p_x_test = np.expand_dims(p_x_test, axis=1)
    ft = np.transpose(ft_space)
    pxt = np.transpose(p_x_test)
    # calculate L2 distance between test vec and trained feature subspace
```

```
    dist = np.sqrt(np.linalg.norm(ft - pxt, axis=1))
    pred_label = y_train[np.argmin(dist)]
    return pred_label


# TASK 1: PCA and LDA Features Classification
def classifier(x_train, x_test, y_train, y_test, p):
    # calculate principal components and linear discriminants
    W_p_PCA, m = calcPCA(x_train, p)
    W_p_LDA, _ = calcLDA(x_train, y_train, p)

    x_train = np.transpose(x_train)
    x_test = np.transpose(x_test)

    # project onto respective subspaces
    PCA_space = np.dot(np.transpose(W_p_PCA), x_train - m)
    LDA_space = np.dot(np.transpose(W_p_LDA), x_train - m)

    num_test_samples = len(y_test)
    x_testm = x_test - m

    pred_lbls_PCA = []
    pred_lbls_LDA = []
    # make predictions using nearest neighbor classifier
    for i in range(num_test_samples):
        PCA_x_test = np.dot(np.transpose(W_p_PCA), x_testm[:, i])
        LDA_x_test = np.dot(np.transpose(W_p_LDA), x_testm[:, i])
        pred_PCA = oneNN(PCA_space, PCA_x_test, y_train)
        pred_LDA = oneNN(LDA_space, LDA_x_test, y_train)
        pred_lbls_PCA.append(pred_PCA)
        pred_lbls_LDA.append(pred_LDA)

    # calculate classification accuracy
    accuracy_PCA = np.mean(np.equal(np.array(pred_lbls_PCA), y_test)) * 100
    accuracy_LDA = np.mean(np.equal(np.array(pred_lbls_LDA), y_test)) * 100

    return accuracy_PCA, accuracy_LDA


# TASK 2: Autoencoder Features Classification
def autoenc_classifier(X_train, Y_train, X_test, Y_test):
    pred_lbls = []
    num_test_samples = len(Y_test)
    for i in range(num_test_samples):
        pred = oneNN(np.transpose(X_train), X_test[i], Y_train)
        pred_lbls.append(pred)

    accuracy = np.mean(np.equal(np.array(pred_lbls), Y_test)) * 100
    return accuracy


def main():
    root_dir = 'FaceRecognition'
```

```
    x_train, y_train = readImages(root_dir, 'train')
    x_test, y_test = readImages(root_dir, 'test')
    x_train, x_test = normalize(x_train), normalize(x_test)

    acc_PCA, acc_LDA, acc_auto = [], [], []

    # evaluate PCA and LDA classifiers for different subspace dimensions
    for p in range(1, 21):
        pca_acc, lda_acc = classifier(x_train, x_test, y_train, y_test, p)
        acc_PCA.append(pca_acc)
        acc_LDA.append(lda_acc)

    # evaluate autoencoder classifier for different subspace dimensions
    for p in [3, 8, 16]:
        X_train, Y_train, X_test, Y_test = autoencoder.auto_encoder(p)
        auto_acc = autoenc_classifier(X_train, Y_train, X_test, Y_test)
        acc_auto.append(auto_acc)

    # plot accuracy vs subspace dimension
    rcParams['figure.dpi'] = 300
    plt.figure(figsize=(9, 7))
    fig, ax = plt.subplots()
    ps = np.arange(1, 21, 1)
    ax.plot(ps, acc_PCA, label="PCA", marker='x')
    ax.plot(ps, acc_LDA, label="LDA", marker='o')
    ax.plot([3, 8, 16], acc_auto, label="Autoencoder", marker="*")
    ax.set_title("Accuracy of Different Dimensionality Reduction Methods vs P")
    ax.set_ylabel("Accuracy %")
    ax.set_xlabel("Feature Dimension (P)")
    ax.set_xticks(np.arange(0, 22, 1))
    ax.legend()
    plt.show()


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

Task 3

```
import numpy as np
import cv2
import glob
import os
import matplotlib.pyplot as plt
from matplotlib import rcParams


def readImages(root_dir, sub_dir):
    # saves images to npz format if reading for the first time, else loads the images
    ↪  from the npz file
    if not os.path.exists(f'{root_dir}/{sub_dir}.npz'):
```

```
        img_list = []
        label_list = []
        img_path_list = glob.glob(os.path.join(root_dir, sub_dir, f'*'))
        for pth in img_path_list:
            # read image and append to array
            im = cv2.imread(pth, 0)
            img_list.append(im)

        np.savez_compressed(f'{root_dir}/{sub_dir}', x=np.array(img_list))

    imgs = np.load(f'{root_dir}/{sub_dir}.npz', allow_pickle=True)
    return imgs['x']


# reffered to past years solutions to implement various functions,
# mainly from 2020 and 2018 solution no 1
# few functions are derived directly from other solutions
def integral_img(img):
    intg_img = np.cumsum(np.cumsum(img, axis=0), axis=1).astype(np.int32)
    return intg_img


def rect_sum(img, coords):
    # takes integral image and calculates the value of the pixel given the filter size in
    ↪  terms of coords
    r, c, off_x, off_y = coords
    A, B, C, D = img[r, c], img[r, c + off_x], img[r + off_y, c], img[
        r + off_y, c + off_x]
    rectsum = A + D - B - C
    return rectsum


def extract_features(img):
    h, w = img.shape
    feats = []
    ct1, ct2 = 0, 0
    # horizontal features
    # b_sum refers to pixels covered by -1 part of filter
    # w_sum refers to pixels covered by 1 part of filter
    for fs in range(2, w, 2):
        for i in range(h - 1):
            for j in range(w - fs):
                coord_b = [i, j, fs // 2, 1]
                coord_w = [i, j + fs // 2, fs // 2, 1]
                # print(fs,coord_b,coord_w)
                b_sum = rect_sum(img, coord_b)
                w_sum = rect_sum(img, coord_w)
                feat_val = w_sum - b_sum
                feats.append(feat_val)
                ct1 += 1

    # vertical features
    for fs in range(2, h, 2):
        for j in range(w - 1):
```

```
            for i in range(h - fs):
                coord_b = [i, j, 1, fs // 2]
                coord_w = [i + fs // 2, j, 1, fs // 2]
                # print(fs, coord_b, coord_w)
                b_sum = rect_sum(img, coord_b)
                w_sum = rect_sum(img, coord_w)
                feat_val = w_sum - b_sum
                feats.append(feat_val)
                ct2 += 1

    # print(ct1, ct2, ct1 + ct2)
    return feats


def calc_feat_mat(imgs, name):
    if not os.path.exists(f'{name}.npz'):
        feat_mat = []
        for im in imgs:
            int_img = integral_img(im)
            feat_vec = extract_features(int_img)
            feat_mat.append(feat_vec)
        np.savez_compressed(f'{name}', x=np.transpose(np.array(feat_mat)))

    arr = np.load(f'{name}.npz', allow_pickle=True)
    return arr['x']


# construct weak classifier
def weak_classifier(weight, feats, labels):
    least_err = np.inf
    # normalising weights arr
    weight = np.divide(weight, np.sum(weight))

    # calc sum of positive and negative example weights
    Tp = np.sum(weight[np.where(labels == 1)])
    Tn = np.sum(weight[np.where(labels == 0)])

    # sort each feature in ascending order across all images
    # returns the indices of sorted features
    idx = np.argsort(feats, axis=1)
    # sort features, weights and labels according to idx
    sorted_feats = np.take_along_axis(feats, idx, axis=1)
    sorted_weight = np.take_along_axis(np.expand_dims(weight, axis=0), idx, axis=1)
    sorted_labels = np.take_along_axis(np.expand_dims(labels, axis=0), idx, axis=1)

    # sum of pos and negative sample weights for each feature less than threshold
    Sp = np.cumsum(sorted_weight * sorted_labels, axis=1)
    Sn = np.cumsum(sorted_weight, axis=1) - Sp
    # errors
    err1 = Sp + (Tn - Sn)
    err2 = Sn + (Tp - Sp)

    best_weak_cl = None
    f_pred = None
```

```
    # iterate through all features
    for i, (feat, s_feat, er1, er2) in enumerate(zip(feats, sorted_feats, err1, err2)):
        # calculate min val between error arrays
        min_err = np.minimum(er1, er2)
        # get index of min value to access params which will give the least error rate
        min_err_idx = np.argmin(min_err)

        # classify using threshold and set polarity depending on feature val
        # being less or more than threshold
        if er1[min_err_idx] <= er2[min_err_idx]:
            pol = 1
            pred = (feat >= s_feat[min_err_idx])
            e = er1[min_err_idx]

        else:
            pol = -1
            pred = (feat < s_feat[min_err_idx])
            e = er2[min_err_idx]

        # create a list with classifier parameters
        # [feature index, threshold(feature val), polarity, error]
        c_params = [i, s_feat[min_err_idx], pol, e]

        # set the current classifier as the best weak classifier
        # if it has lesser error than any of the previous classifers
        if min_err[min_err_idx] < least_err:
            best_weak_cl = c_params
            f_pred = pred.astype(np.uint8)
            least_err = min_err[min_err_idx]

    return best_weak_cl, f_pred


# cascade multiple weak classifiers to create a strong classifier
def cascade(tr_pos_feat, tr_neg_feat, pos_lbls, neg_lbls, reqTpr, reqFpr, maxNumWcl):
    cl_list = []
    alpha_list = []
    pred_list = []
    tpr, fpr, tnr, fnr, rep_fpr = 0, 0, 0, 0, 0

    numP = tr_pos_feat.shape[1]
    numN = tr_neg_feat.shape[1]
    weights = np.ones(numP) * 0.5 / numP  # set pos weights
    weights = np.append(weights, np.ones(numN) * 0.5 / numN)  # append neg weights
    # combine pos and neg feats into a single arr
    feats = np.hstack((tr_pos_feat, tr_neg_feat))
    labels = np.hstack((pos_lbls, neg_lbls))

    # run in range till max possible num of weak classifiers
    for n in range(maxNumWcl):
        best_weak_cl, f_pred = weak_classifier(weights, feats, labels)
        cl_list.append(best_weak_cl)
```

```
            eps = best_weak_cl[-1]
            beta = eps / (1 - eps + 1e-6)
            alpha = np.log(1 / beta + 1e-6)  # to prevent division by zero
            alpha_list.append(alpha)

            # update weights
            weights = weights * np.power(beta, (1 - np.abs(labels - f_pred)))

            # compute output of the cascade
            pred_list.append(f_pred)
            feat_arr = np.transpose(np.array(pred_list))
            alpha_arr = np.transpose(np.array([alpha_list]))
            casc_out = np.dot(feat_arr, alpha_arr)

            # get threshold based on minimum val of cascade output in first numP entries
            thresh = np.min(casc_out[:numP])
            Cx = np.zeros_like(casc_out)
            # set final output of cascade wherever the cascade output is greater than
            ↪  threshold
            Cx[casc_out >= thresh] = 1

            # calculate true positive and false positive rates
            tpr = np.sum(Cx[:numP]) / numP
            fpr = np.sum(Cx[numP:]) / numN
            tnr = 1 - fpr
            fnr = 1 - tpr
            rep_fpr = np.sum(Cx[numP:]) / 1758

            if tpr >= reqTpr and fpr <= reqFpr:
                print(f'Num Classifier in stage is {n + 1}')
                break

        metrics = np.array([tpr, rep_fpr, tnr, fnr])
        # remove correctly classified negative examples from current set
        neg_preds = Cx[numP:]
        neg_ids = np.where(np.equal(neg_preds, 1))[0]  ###
        refined_neg_feats = tr_neg_feat[:, neg_ids]
        new_neg_labels = np.zeros(len(neg_ids))
        print(f'False Postives in stage = {len(neg_ids)}')
        cl_list = np.array(cl_list)
        alpha_list = np.array(alpha_list)
        return refined_neg_feats, new_neg_labels, cl_list, alpha_list, metrics


    def main():
        root_dir = 'CarDetection'
        # load images
        train_pos, train_neg = readImages(root_dir, 'train/positive'), readImages(root_dir,
        ↪  'train/negative')
        test_pos, test_neg = readImages(root_dir, 'test/positive'), readImages(root_dir,
        ↪  'test/negative')

        # generate labels
        tr_pos_lbl = np.ones((train_pos.shape[0]))
```

```
tr_neg_lbl = np.zeros((train_neg.shape[0]))
# te_pos_lbl = np.ones((test_pos.shape[0]))
# te_neg_lbl = np.zeros((test_neg.shape[0]))

# calculate features and save to npz if not already saved
tr_pos_ft = calc_feat_mat(train_pos, 'train_pos')
tr_neg_ft = calc_feat_mat(train_neg, 'train_neg')
te_pos_ft = calc_feat_mat(test_pos, 'test_pos')
te_neg_ft = calc_feat_mat(test_neg, 'test_neg')

# training
if not os.path.exists(f'stage_cl_al.npz'):
    max_stages = 10
    stage_cl_list = []
    stage_alpha_list = []
    # create cascade with max num stages
    tr_fpr_arr = []
    for stage in range(max_stages):
        tr_neg_ft, tr_neg_lbl, cl_list, alpha_list, metrics = cascade(tr_pos_ft,
        ↪  tr_neg_ft, tr_pos_lbl, tr_neg_lbl,
                                                                      1,
                                                                      0.45,
                                                                      50)
        _, fpr, _, _ = metrics

        # append classifiers for testing later
        stage_cl_list.append(cl_list)
        stage_alpha_list.append(alpha_list)
        # append fpr rates while training
        tr_fpr_arr.append(fpr)
        # if there are no more negative samples, stop training
        if tr_neg_ft.shape[1] == 0:
            print(f'Training Stopped at Stage: {stage + 1}')
            break
    # save cascade (classifiers and alphas)
    np.savez_compressed(f'stage_cl_al', x=np.array(stage_cl_list, dtype=object),
                        y=np.array(stage_alpha_list, dtype=object))
    print(tr_fpr_arr)

# testing
# load saved cascade
stage_cl_al = np.load(f'stage_cl_al.npz', allow_pickle=True)
print(f'stage_cl_al loaded')
# unpack classifiers and alphas
stage_cl = stage_cl_al['x']
stage_al = stage_cl_al['y']

test_feats = np.hstack((te_pos_ft, te_neg_ft))

pred_arr = []
wt_mul = 0.5
# iterate through every stage's classifiers
for cls, als in zip(stage_cl, stage_al):
    wpred_arr = []
```

```
        # print(len(cls), len(als))
        # iterate through each weak classifier
        for cl in cls:
            # unpack classifier params
            i, thresh, pol, _ = cl
            # go to feature index in test samples
            current_feat = test_feats[int(i)]
            # classify
            test_pred = current_feat >= thresh if pol == 1 else current_feat <= thresh
            wpred_arr.append(np.array(test_pred).astype(np.uint8))

        wpred_arr = np.transpose(np.array(wpred_arr))
        # cascade output
        c_out = np.dot(wpred_arr, als)
        thr = np.sum(als) * wt_mul
        cx = np.zeros_like(c_out)
        cx[c_out >= thr] = 1
        pred_arr.append(cx)

    pred_arr = np.array(pred_arr)
    # print(pred_arr.shape)
    test_numP = test_pos.shape[0]
    test_numN = test_neg.shape[0]

    te_fpr_list = []
    te_fnr_list = []

    # evaluate predicted labels
    for i in range(pred_arr.shape[0]):
        ps_ = pred_arr[0].copy()
        # this loop removes correctly classified negative examples from the pred label
        ↪ set
        for j in range(i+1):
            ps_= np.logical_and(ps_,pred_arr[j])

        ref_pred = ps_
        te_fpr_list.append(np.sum(ref_pred[test_numP:]) / test_numN)
        te_fnr_list.append(1 - np.sum(ref_pred[:test_numP]) / test_numP)

    print(te_fpr_list)
    print(te_fnr_list)



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