Task 1: Face Recognition using PCA and LDA

PCA and LDA are techniques commonly used for analyzing large datasets that contain a high number of dimensions or features. These statistical techniques help reduce the dimensions of the data while preserving maximum amount of information.

1.1 PCA Background and Methodology

PCA creates a lower dimension subspace of the original feature space where the projected data will have the highest variance.

Brief Summary of the process:

- Each image is vectorized and added to the feature matrix where each column is the feature vector of an image from the dataset.
- The feature vectors are normalized such that they have zero mean and unit variance.
- The lower K-dimensional subspace of the original data has a bases that is formed by the K largest eigenvectors of the covariance matrix **C** and hence the largest K eigenvectors of **C** are computed.

  $$\mathbf{C} = \frac{1}{N}\sum_{i=0}^{N-1}(\vec{x_i} - \vec{m})(\vec{x_i} - \vec{m})^T$$

  where $\vec{x_i}$ is i^{th} image vector and $\vec{m}$ is the global mean vector

  - With $\mathbf{X} = \vec{x_i} - \vec{m}$ the inner product $\mathbf{X}\mathbf{X}^T$ will be very large and computational slow for eigenvector decomposition.
  - Therefore the eigenvector decomposition is performed on $\mathbf{X}^T\mathbf{X}$ to get $\vec{u}$, which can be used to compute $\vec{w}$ the eigenvectors of $\mathbf{X}\mathbf{X}^T$ through:

    $$\vec{w} = \mathbf{X}\vec{u}$$

  - Since the rank of $\mathbf{X}\mathbf{X}^T$ is N (number of images), at most N $\vec{w}$ eigenvectors can be picked. Hence, the first K largest eigenvectors are picked where K< N.
- Once the K largest eigenvectors w_K are computed, all the training samples are projected onto its dimensional space by:

  $$\mathbf{Y} = \vec{w_k}\mathbf{X}$$

- Similarly, the testing samples are projected onto the same dimensional space.
- Using the k-nearest neighbors method in the lower dimensional space, each testing sample is compared with all the training samples to determine its label.

1.2 LDA Background and Methodology

LDA creates a lower dimension subspace of the original feature space by finding the direction in the underlying vector space that are maximally discriminating between the classes. This direction is maximally discriminating between the classes if it simultaneously maximizes the between-class scatter and minimizes the within-class scatter along that direction

Brief Summary of the process:

- Each image is vectorized and added to the feature matrix where each column is the feature vector of an image from the dataset.
- The feature vectors are normalized such that they have zero mean and unit variance.
- The eigenvectors $\vec{\mathbf{w}}$ of underlying space are computed such that they maximize the ratio of between-class scatter to within-class scatter. This is ratio is called the Fisher Discriminant Function:
$$J(\vec{w}) = \frac{\vec{w}S_B\vec{w}}{\vec{w}S_W\vec{w}}$$
where:
$$S_B = \frac{1}{C}\sum_{i=1}^{C}(\vec{m}_i - \vec{m})(\vec{m}_i - \vec{m})^T$$
$$S_W = \frac{1}{C}\sum_{i=1}^{C}\frac{1}{C_i}\sum_{k=1}^{C_i}(\vec{x_k}^i - \vec{m}_i)(\vec{x_k}^i - \vec{m}_i)^T$$
C: number of classes C_i: number of images in class i. $\vec{x_k}^i$: k^{th} image in class i
$\vec{m}$: Global mean $\vec{m}_i$: Class mean
- The eigendecomposition of $S_W^{-1}S_B$ can be used to get the eigenvectors $\vec{\mathbf{w}}$
  - Since S_W can be singular sometimes, S_W^{-1} will not exist. Hence Yu and Yang's algorithm is used.
  - S_B is diagolized through eigendecomposition and the diagonal matrix D_B of the eigenvalues is created such that $\mathrm{Y^TS_BY = D_B}$
  - The matrix $\mathbf{Z} = \mathrm{YD_B^{-1/2}}$ is computed and the $\mathbf{Z}^TS_W\mathbf{Z}$ is diagonlized through eigendecomposition to get $\mathbf{U}$ eigenvector matrix
  - The LDA eigenvectors $\vec{\mathbf{w}}$ are then computed as $\vec{\mathbf{w}} = \mathbf{ZU}^T$
- Once the K largest eigenvectors $\vec{\mathbf{w}}_K$ are computed, all the training samples are projected onto its dimensional space by:
$$\mathbf{Y} = \vec{\mathbf{w}}_K\mathbf{X}$$
- Similarly, the testing samples are projected onto the same dimensional space.
- Using the k-nearest neighbors method in the lower dimensional space, each testing sample is compared with all the training samples to determine its label.

1.3: Results and Observations

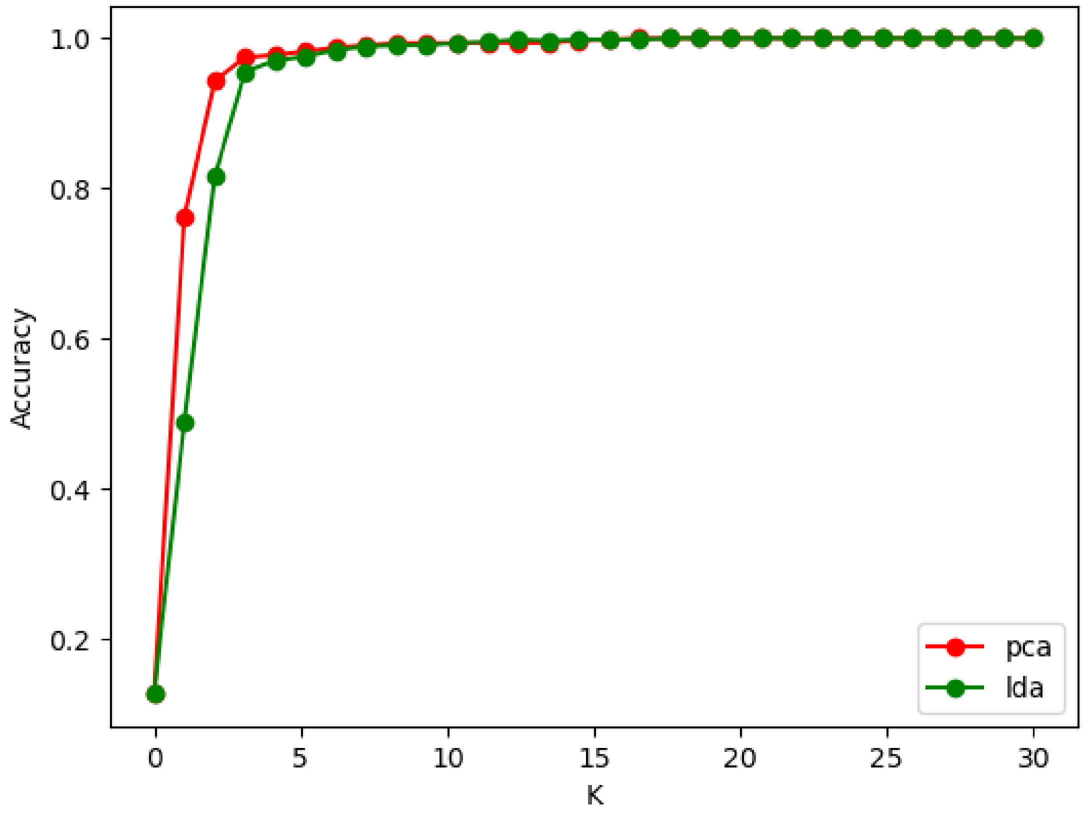


Figure 1: Accuracy: PCA vs LDA

As can be seen from the accuracy plot above, PCA and LDA work very well as the dimensional space increases. With $K = 20$, the accuracy for both the methods is close to 100%. It can be seen that PCA achieves higher accuracy when K is under 4.

Task 2: Face Recognition with Autoencoders

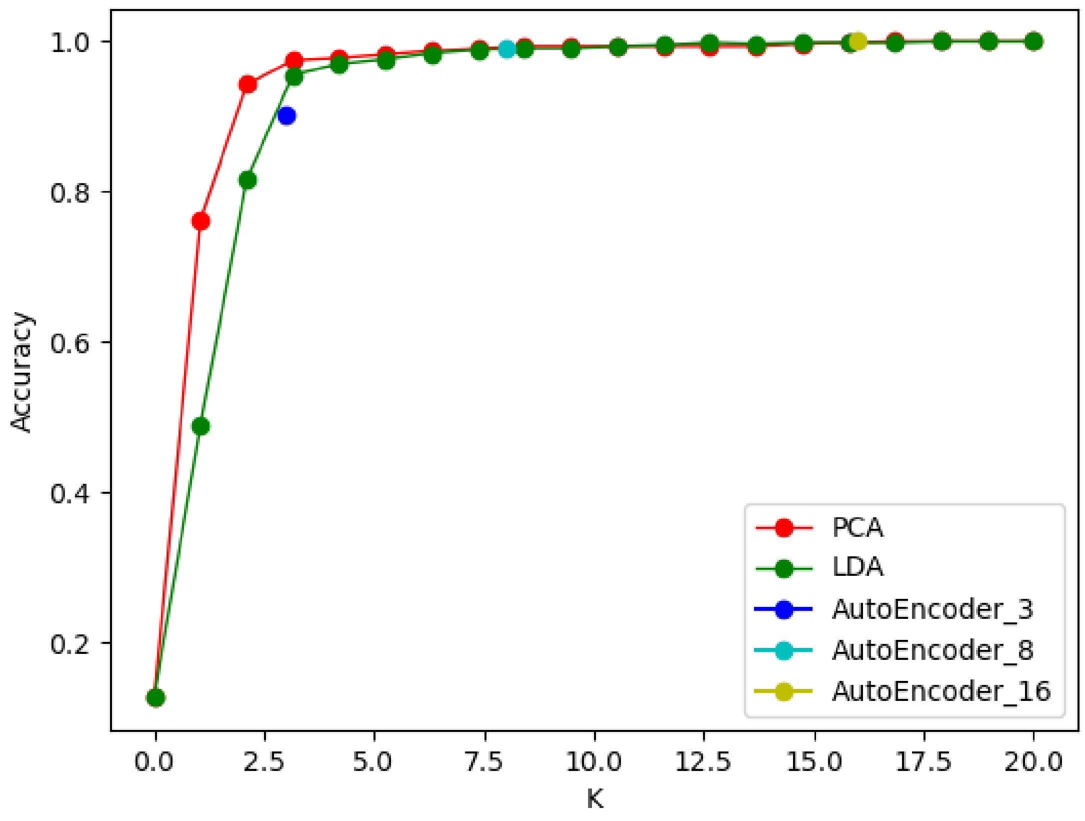


Figure 2: Accuracy: PCA vs LDA vs Autoencoder

As can be seen from the accuracy plot above, all three methods achieve close to 100% accuracy. At lower dimensional space of around 3 4, PCA works the best, followed by LDA and then Autoencoders. With $K = 20$, the accuracy for all three methods is close to 100%.

Task 3: Car detection with Cascaded Adaboost

AdaBoost stands for Adaptive Boosting and is made of cascading weak classifiers in a sequence in which each weak classifier is the best choice for a classifier at that point for rectifying the errors made by the previous classifier.

3.1 Background and Methodology

Brief Summary of the process:

- A feature matrix is created and then convolved with the image vectors to produce feature vectors used in the algorithm.
- For $t = 1, 2, \ldots, T$ where t is a cascade stage and T is the total number of cascade stages defined by the user, a new AdaBoost classifier is added and in each Adaboost classifier the following is done:
    1. Weights are initialized. If the positive classes are p in number and negative classes are n in number
    $$w_i = \begin{cases} \frac{1}{2p} & \text{if } x = 1 \\ \frac{1}{2n} & \text{if } x = 0 \end{cases}$$
    2. for $n = 1, \ldots, N$, where N is desired number of weak classifiers:
        - The weights are normalized $w_i = \frac{w_{ji}}{\sum_j w_{ji}}$
        - The best weak classifier is computed based on the weighted error of the set of training samples. Each weak classifier looks at a single feature (f). Each weak classifier has a threshold (θ) and a polarity (p) to determine the classification of a training sample. θ is set to the value of the feature at which the error is minimum. $p = 1$, if there are more positive samples with feature values less than the threshold, else $p = -1$.:
        $$h(x, f, p, \theta) = \begin{cases} 1 & \text{if } pf(x) < p(\theta), \\ 0 & \text{otherwise} \end{cases}$$
        - Iterate through all the weak classifiers to get the average weighted error of each one on the training dataset, and choose the classifier with the lowest error as the best weak classifier.
        - Update the weights with the error of the best weak classifier calculated previously:
        $$\epsilon = min_{f,p,\theta} \sum_{i}^{N} w_i |h(x, f, p, \theta) - y_i|$$
        $$\beta = \frac{\epsilon}{1 - \epsilon}$$
        $$\mathrm{w}_i = w_i \beta^{1-\epsilon_i}$$

3. All the best weak classifiers are combined to form a strong classifier

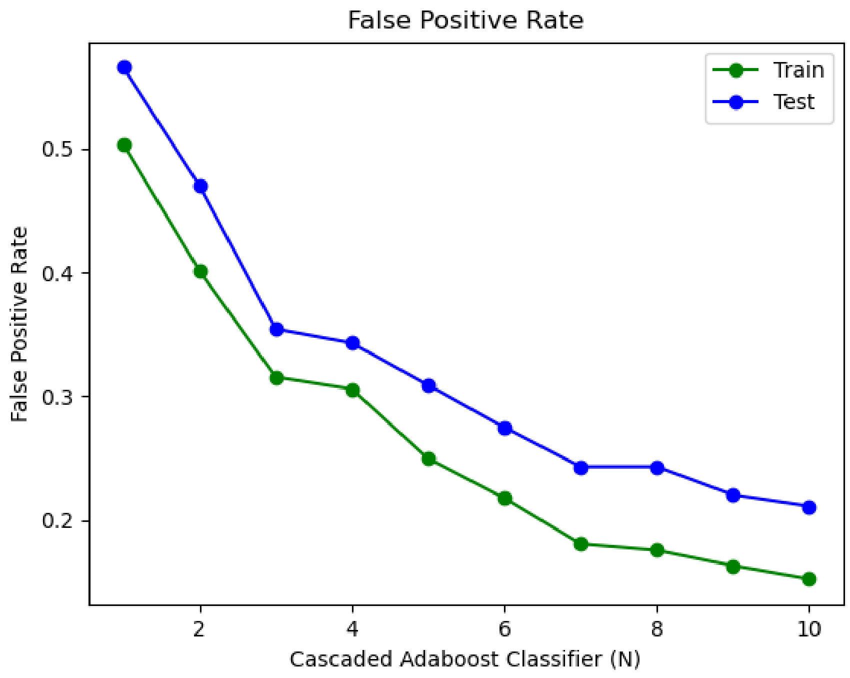
$$C(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t, \\ 0 & \text{otherwise} \end{cases}$$

$$\alpha_t = ln\left(\frac{1}{\beta_t}\right)$$

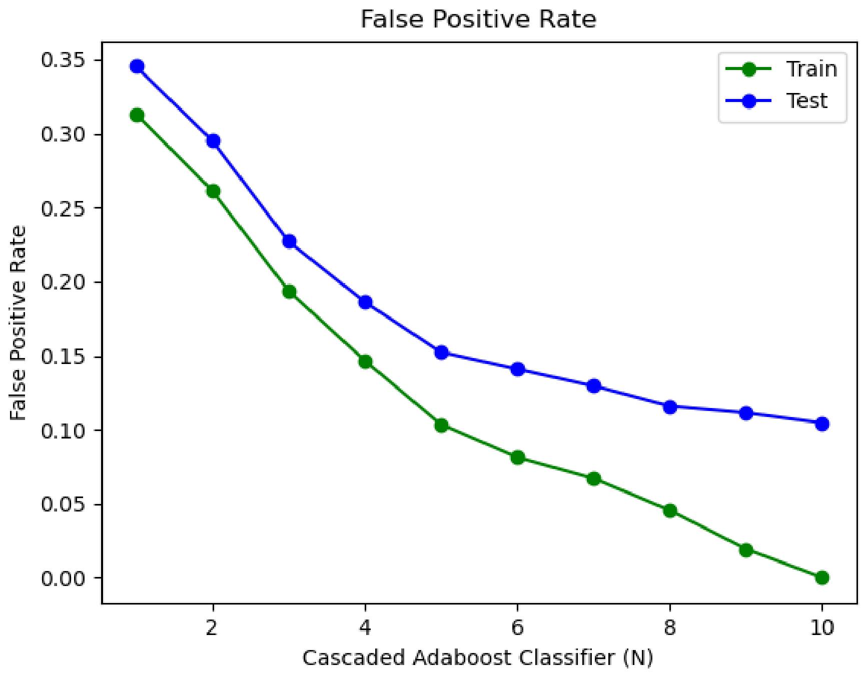
3.2: Observations

- As can be seen from the plots below, false positive rate decreases at each cascade stage.
- False negative rate and true positive rate changes more slowly than false positive rate at each cascade stage. This shows that a power like x^n decreases with n much more slowly when x is close to 1 compared to the case when x is closer to 0.
- To see how different number of weak classifiers per cascade affect performance, the number of cascade stages was fixed at 10, and the number of weak classifier per cascade was varied. Two sets of results were collected, when number of weak classifier is 15 vs when it is 25.
- As can be seen from the plots below, the false positive rate for 25 classifiers per cascade decreases to 0% for the training data and close to 13%for the test data, compared to around 20% for 15 classifiers per cascade.
- On the other hand, the true positive rate decreases to 93% for the test data for 15 classifiers per cascade and stays constant for each cascade stage after 9. However, the true positive rate for 25 classifiers per cascade decreases even further to 86% by stage 10. This shows that if more classifiers are added, it lowers the false positive rate but also lowers the true positive rate.
- Therefore, the optimal number of classifiers for 10 cascades should be less than 25 and greater than 15. Increasing the cascade stages can also increase the true positive rate upto a point before it becomes constant.

3.3: Results

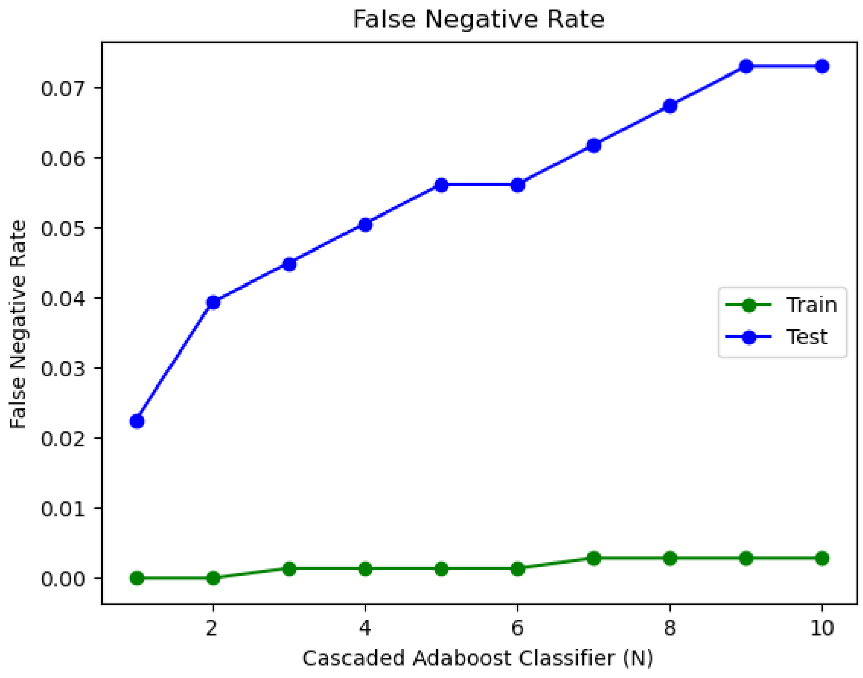


(a) 15 weak classifiers per cascade

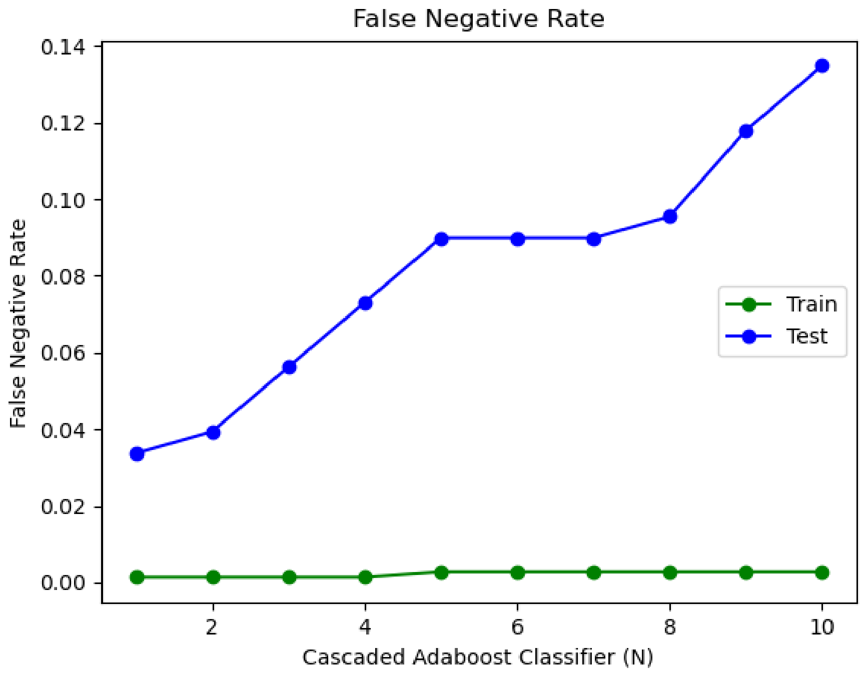


(b) 25 weak classifiers per cascade

Figure 3: False Positive Rate

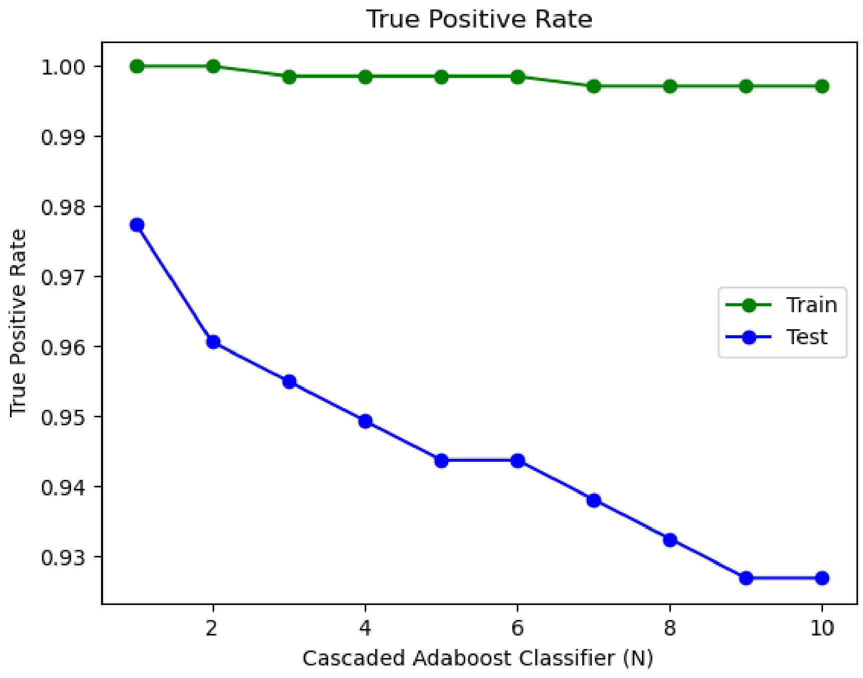


(a) 15 weak classifiers per cascade

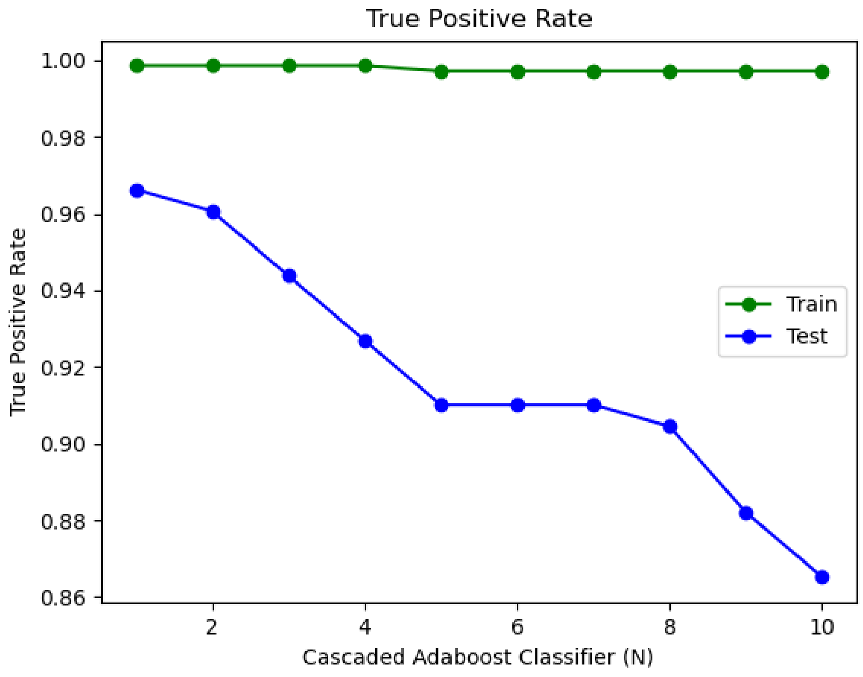


(b) 25 weak classifiers per cascade

Figure 4: False Positive Rate

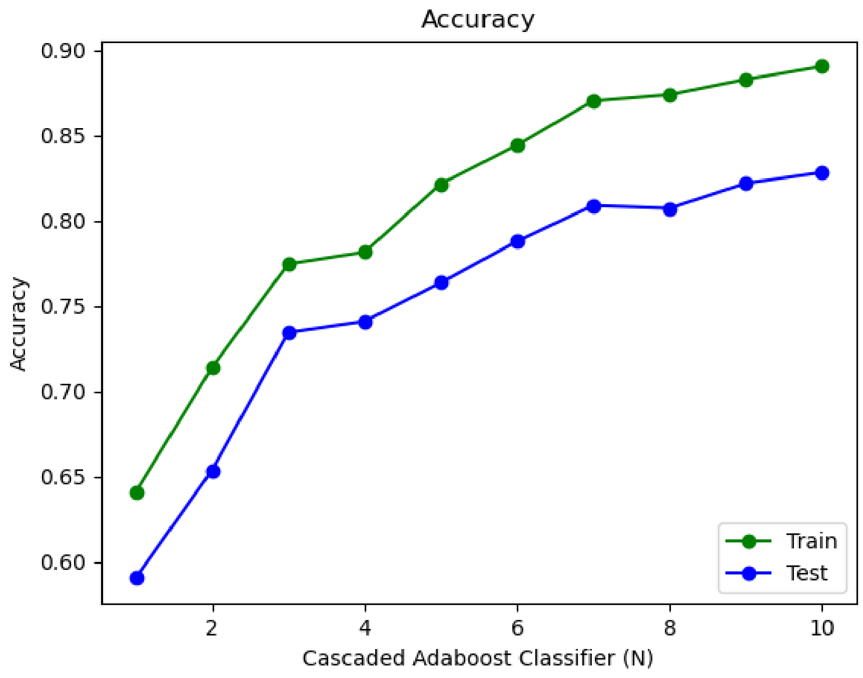


(a) 15 weak classifiers per cascade

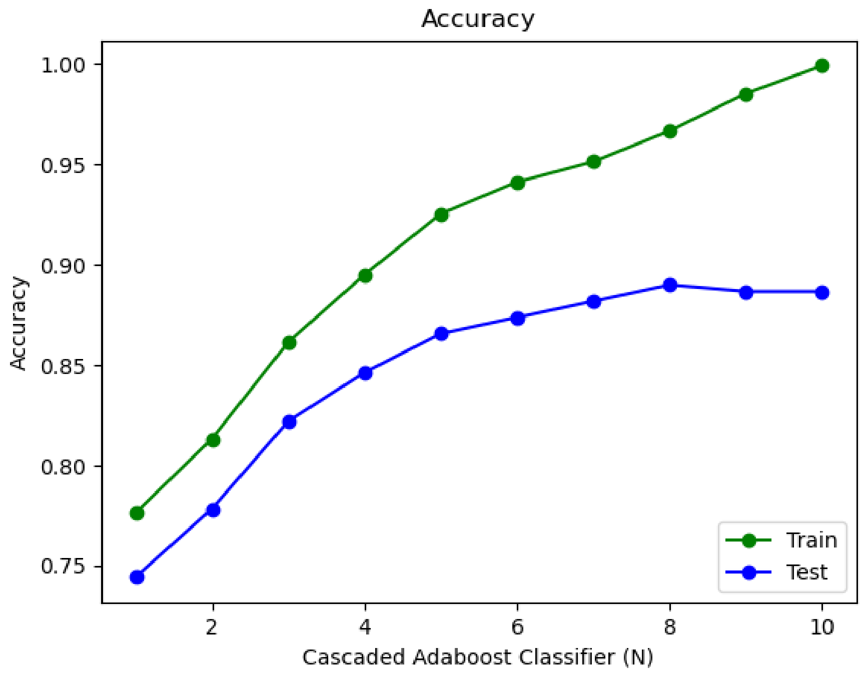


(b) 25 weak classifiers per cascade

Figure 5: False Negative Rate



(a) 15 weak classifiers per cascade



(b) 25 weak classifiers per cascade

Figure 6: Accuracy

Source Code

Main.py

```python
import cv2 as cv
import numpy as np
from PCA import PCA
from LDA import LDA
import os
import matplotlib.pyplot as plt
from NearestNeighbor import KNearestNeighbor
from adaboost import CascadedAdaboost

###============================Functions============================###
def loadData(data_path, dim, uglabels = None):
    imgList = [img for img in os.listdir(data_path) if img.endswith(".png"
                                                    )]
    num_imgs = len(imgList)
    X = np.zeros((dim[0]*dim[1], num_imgs), dtype=float)
    labels = np.zeros((num_imgs, 1), dtype=float)
    for i,imgName in enumerate(imgList):
        img = cv.imread(os.path.join(data_path, imgName), 0)
        X[:,i] = img.flatten()
        if uglabels == None:
            labels[i] = int(imgName.split('_')[0])
        if uglabels != None:
            labels[i] = uglabels
    # Normalize data
    X_mean = np.mean(X)
    X_std = np.std(X)
    X = X - X_mean
    X = X / X_std
    return X, labels

def Dim_Red(train_x, train_l, test_x, test_l, algos, maxK):
    accuracies = []
    for K in range(1,maxK+1):
        print(f'K: {K}')
        if 'pca' in algos:
            pca = PCA()
            pca.train(train_x, train_l, K)
            pca_acc = pca.test(test_x, test_l)
            print(f'PCA Accuracy: {pca_acc}')
            accuracies.append(pca_acc)
        if 'lda' in algos:
            lda = LDA()
            lda.train1(train_x, train_l, K)
            lda_acc = lda.test(test_x, test_l)
            print(f'LDA Accuracy: {lda_acc}')
            accuracies.append(lda_acc)
    if len(algos) == 2:
        accuracies = np.reshape(accuracies, (maxK, 2))
    elif len(algos) == 1:
        accuracies = np.reshape(accuracies, (maxK, 1))
    return np.asarray(accuracies)

def plotAccuracies(accuracies):
    maxK = accuracies.shape[0]
    xdata = np.linspace(0,maxK, num=maxK)
    acc_plt = plt.figure()
    line1, = plt.plot(xdata, accuracies[:,0], '-ro', label = 'PCA',
                                          linewidth=1)
    line2, = plt.plot(xdata, accuracies[:,1], '-go', label = 'LDA',
                                          linewidth=1)
    return acc_plt, line1, line2

def plotTask3res(train_res, test_res, name, loc):
    plt.figure()
    line1, = plt.plot(np.arange(1, len(train_res)+1), train_res, '-go',
                                          label = 'Train')
    line2, = plt.plot(np.arange(1, len(test_res)+1), test_res, '-bo',
                                          label = 'Test')
    plt.legend(handles=[line1, line2], loc=loc)
    plt.xlabel('Cascaded Adaboost Classifier (N)')
    plt.ylabel(name)
    plt.title(name)
    plt.show()

###========================Main Function==========================###
## Load data
print("Task 1 and Task 2")
print("Loading data...")
train_x, train_l = loadData('HW10/FaceRecognition/train/', (128,128))
print(f'Training Data: {train_x.shape}, Training Labels: {train_l.shape}')

test_x, test_l = loadData('HW10/FaceRecognition/test/', (128,128))
print(f'Test Data: {test_x.shape}, Test Labels: {test_l.shape}')

# ### Task 1: Face Recognition with PCA, LDA
print("Starting PCA and LDA")
algos = ['pca', 'lda']
accuracies = Dim_Red(train_x, train_l, test_x, test_l, algos, maxK = 20)
plot, line1, line2 = plotAccuracies(accuracies)

### Task 2: Face Recognition with Autoencoders
print("Starting Autoencoders")
from autoencoder import autoencoderOutput

P = [3, 8, 16]
acc_autoencoder = np.zeros(len(P))
for i,p in enumerate(P):
    print(f'P:{p}')
    X_train, y_train, X_test, y_test = autoencoderOutput(p)

    # Fit k-Nearest neighbors and predict
    NNmodel = KNearestNeighbor(k_neighbors = 1)
    NNmodel.fit(X_train, np.reshape(y_train, (y_train.shape[0], 1)))
    predicted_labels = NNmodel.kneighbors(X_test)
    acc_autoencoder[i] = np.sum((predicted_labels - np.reshape(y_train, (
                                        y_train.shape[0], 1))) == 0) /
                                        y_test.size

# Plot the accuracies on the same plot
line_ae_3, = plt.plot(3, acc_autoencoder[0], '-bo', label = 'AutoEncoder_3
                                    ')
line_ae_8, = plt.plot(8, acc_autoencoder[1], '-co', label = 'AutoEncoder_8
                                    ')
line_ae_16, = plt.plot(16, acc_autoencoder[2], '-yo', label = '
                                    AutoEncoder_16')

plt.legend(handles=[line1, line2, line_ae_3, line_ae_8, line_ae_16], loc=4
                                    )
plt.xlabel('K')
plt.ylabel('Accuracy')
plt.show()

### Task 3: Adaboost Classifier
# Load data
print("Starting Adaboost")
print("Loading data")
train_x_pos, train_l_pos = loadData('HW10/CarDetection/train/positive', (
                                    40,20), 1)
train_x_neg, train_l_neg = loadData('HW10/CarDetection/train/negative', (
                                    40,20), 0)
train_x = np.hstack((train_x_pos, train_x_neg))
train_l = np.hstack((train_l_pos.T, train_l_neg.T))
print(f'Training Data: {train_x.shape}, Training Labels: {train_l.shape}')

test_x_pos, test_l_pos = loadData('HW10/CarDetection/test/positive', (40,
                                    20), 1)
test_x_neg, test_l_neg = loadData('HW10/CarDetection/test/negative', (40,
                                    20), 0)
test_x = np.hstack((test_x_pos, test_x_neg))
test_l = np.hstack((test_l_pos.T, test_l_neg.T))
print(f'Test Data: {test_x.shape}, Test Labels: {test_l.shape}')

num_cascades = 10
num_classifier_per_cascade = 25
carClassifier = CascadedAdaboost(train_x, train_l, test_x, test_l)
FP_train, FN_train, TP_train, Ac_train, FP_test, FN_test, TP_test, Ac_test
                                     = carClassifier.train(num_cascades,
                                    num_classifier_per_cascade)

# Plot results
plotTask3res(FP_train, FP_test, 'False Positive Rate', 1)
plotTask3res(FN_train, FN_test, 'False Negative Rate', 7)
plotTask3res(TP_train, TP_test, 'True Positive Rate', 7)
plotTask3res(Ac_train, Ac_test, 'Accuracy', 4)
```

PCA.py

```
import cv2
import numpy as np
from NearestNeighbor import KNearestNeighbor

class PCA(object):
    def __init__(self):
        self.m = None # mean
        self.K = 0  # K largest eigenvectors
        self.Wk = None # PCA Feature set (eigenvectors of covariance
                                          matrix)
        self.NNmodel = None # Nearest Neighbor

    def train(self, train_x, train_l, K):
        print("Performing PCA")
        self.K = K              # K largest eigenvectors
        X = train_x
        self.m = np.mean(X, axis=1)     # Global mean
        X = X - self.m.reshape(-1,1)    # Zero mean img vectors
        # Using the trick to compute eigenvectors faster
        u,s,v = np.linalg.svd(np.dot(X.T,X))
        # w = Xu
        w = np.dot(X, v.T)
        w = w / np.linalg.norm(w, axis = 0)

        # Taking the first K eigenvectors
        self.Wk = w[:,:self.K]
        # Project training data into the eigenspace
        Y = np.dot(self.Wk.T, (X))
        # k-Nearest Neigbors model of the data
        self.NNmodel = KNearestNeighbor()
        self.NNmodel.fit(Y.T, train_l)

    def test(self, test_x, test_l):
        # Assign training vectors to X
        X = test_x
        # Project testing data into eigenspace
        Y = np.dot(self.Wk.T, X - self.m.reshape(-1,1))
        # Predict labels with k-Nearest Neighbors
        predicted_labels = self.NNmodel.kneighbors(Y.T)
        # Accuracy
        acc = np.sum((predicted_labels - test_l) == 0) / np.float(test_l.
                                          size)
        return acc
```

LDA.py

```
import cv2
import numpy as np
from NearestNeighbor import KNearestNeighbor

```

```
class LDA(object):
    def __init__(self):
        self.m = None # mean
        self.K = 0  # K largest eigenvectors
        self.Wk = None # PCA Feature set (eigenvectors of covariance
                                                          matrix)
        self.NNmodel = None # Nearest Neighbor

    def train1(self, train_x, train_l, K):
        print("Performing LDA")
        # Get labels
        class_labels, counts = np.unique(train_l, return_counts=True)
        C = len(class_labels) # num of labels
        self.K = C-1 if K > C-1 else K  # At most eigenvectors C-1
        X = train_x
        # Construct S_W (within class scatter) matrix
        self.m = np.reshape(np.mean(X, axis=1), (X.shape[0], 1))    #
                                                          Global mean
        mc = np.zeros((X.shape[0], C))                              #
                                                          class mean
        mi_m = np.zeros((X.shape[0],X.shape[1]))                    #
                                                          Between class mean
        for i,c in enumerate(class_labels):
            cols = (train_l==c).flatten()
            X_c = X[:, cols]
            m_c = np.mean(X_c, axis=1)
            mc[:,i] = m_c
            mi_m[:,cols] = X_c - np.reshape(m_c, (X.shape[0], 1))

        # Construct S_B matrix  (between class scatter)
        xMat = mc - self.m
        # Diagolize S_B with eigendecomposition
        _,S,V = np.linalg.svd(np.dot(xMat.T,xMat))
        eigMat = np.eye(C)*S
        DB = np.sqrt(np.linalg.inv(eigMat))
        # Construct matrix Z = Y(DB)^-1/2
        Y = np.dot(xMat, V.T)
        Z = np.dot(Y, DB)
        # eigendecomposition of Z^T * SW * Z
        x_z = np.dot(np.dot(Z.T, mi_m), np.transpose(np.dot(Z.T, mi_m)))
        _,S,U = np.linalg.svd(x_z)
        # LDA eigenvectors that maximize the Fisher discriminant function
        w_z = np.dot(Z, U.T)
        w_z = w_z / np.linalg.norm(w_z, axis = 0)
        # Take the K largest
        self.Wk = w_z[:,:self.K]

        # Project training data into the eigenspace
        Y = np.dot(self.Wk.T, (X - self.m.reshape(-1,1)))
        # k-Nearest Neigbors model of the data
        self.NNmodel = KNearestNeighbor()
        self.NNmodel.fit(Y.T, train_l)
    def test(self, test_x, test_l):
        # Assign training vectors to X
        X = test_x
        # Project testing data into eigenspace
        Y = np.dot(self.Wk.T, X - self.m.reshape(-1,1))
        # Predict labels with k-Nearest Neighbors
        predicted_labels = self.NNmodel.kneighbors(Y.T)
        # Accuracy
        acc = np.sum((predicted_labels - test_l) == 0) / np.float(test_l.
                                            size)
        return acc
```

KNearestNeighbor.py

```
import numpy as np

def distEuclidean(a, b):
    dist = np.linalg.norm(a-b)
    return dist

class KNearestNeighbor:
    def __init__(self, k_neighbors = 1):
        self.k_neighbors = k_neighbors
        self.distance = distEuclidean
        self.X = None

    def fit(self, x, labels):
        # combine instances and their labels
        self.X = np.hstack((x,labels))

    def kneighbors(self, Xp):
        predicted_labels = np.zeros((len(Xp), 1))
        for i,xp in enumerate(Xp):
            neighbors = []
            # find distances of instances
            dists = {self.distance(xp, x[:-1]): x for x in self.X}
            for key in sorted(dists.keys())[:self.k_neighbors]:
                # Add the labels of the k nearest neighbors
                label = int(dists[key][-1])
                neighbors.append(label)
            neighbors = np.array(neighbors).flatten()
            # get the most common label
            counts = np.bincount(neighbors)
            label = np.argmax(counts)
            # label = max(set(neighbors), key = neighbors.count)
            predicted_labels[i] = label
        return predicted_labels
```

Autoencoder.py

```
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
from NearestNeighbor import KNearestNeighbor


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


##################################
# Change these
# p = 8#[3, 8, 16]

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

def autoencoderOutput(p):
    training = False
    TRAIN_DATA_PATH = 'HW10/FaceRecognition/train/'
    EVAL_DATA_PATH = 'HW10/FaceRecognition/test/'
    LOAD_PATH = f'HW10/weights/model_{p}.pt'
    OUT_PATH = 'HW10/output/'

    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)))

    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)

    return X_train, y_train, X_test, y_test
```

Adaboost.py

```
import cv2 as cv
import numpy as np

IMG_H = 20
IMG_W = 40
NUM_FEATURES = 47232

def get_ftMat():
    '''
        Calculate the feature matrix to compute the feature vectors
    '''
    # ftMat was initiallized with a very larger number (50000, 20*80)
                                                first
    # and NUM_features was then updated
    ftMat = np.zeros((NUM_FEATURES, IMG_H*IMG_W), dtype=np.int8)
    ftNum = 0
    offset = 2
    # Horizontal filter
    filter_w, filter_h = 2, 1
    for i in range(1, IMG_H, filter_h):
        for j in range(1, IMG_W, filter_w):
            for y in range(offset, IMG_H - filter_h*i + 1 - offset):
                for x in range(offset, IMG_W - filter_w*j + 1 - offset):
                    ftMat[ftNum, y*IMG_H + x] = 1.0
                    ftMat[ftNum, y*IMG_H + filter_w*j//2] = -2.0
                    ftMat[ftNum, y*IMG_H + filter_w*j] = 1.0
                    ftMat[ftNum, (y+filter_w*i)*IMG_H+x] = -1.0
                    ftMat[ftNum, (y+filter_w*i)*IMG_H+x+filter_w*j//2] = 2
                                                                     .0
                    ftMat[ftNum, (y+filter_w*i)*IMG_H+x+filter_w*j] = -1.0
                    ftNum += 1
    # Vertical filter
    filter_w, filter_h = 1, 2
    for i in range(1, IMG_H, filter_h):
        for j in range(1, IMG_W, filter_w):
            for y in range(offset, IMG_H - filter_h*i + 1 - offset):
                for x in range(offset, IMG_W - filter_w*j + 1 - offset):
                    ftMat[ftNum, y*IMG_H + x] = -1.0
                    ftMat[ftNum, y*IMG_H + filter_w*j] = 1.0
                    ftMat[ftNum, (y+filter_h*i//2)*IMG_H+x] = 2.0
                    ftMat[ftNum, (y+filter_h*i//2)*IMG_H+x+filter_w*j] = -
                                                                     2.0
                    ftMat[ftNum, (y+filter_h*i)*IMG_H+x//2] = -1.0
                    ftMat[ftNum, (y+filter_h*i)*IMG_H+x+filter_w*j] = 1.0
                    ftNum += 1
    # print(ftNum)
    return ftMat

class CascadedAdaboost(object):
    '''
        Inspired from
        1. https://github.com/lifangda01/ECE661-ComputerVision/blob/master
                                                            /HW11/
        2. https://medium.com/@rohan.chaudhury.rc/adaboost-classifier-for-
                                                            face-
                detection-using-viola-jones-algorithm-30246527db11
    '''
    def __init__(self, train_X, train_l, test_X, test_l):
        self.train_X = train_X
        self.train_l = train_l
        self.test_X = test_X
        self.test_l = test_l
        self.train_posNum =  0
        self.train_negNum =  0
        self.ftMat = get_ftMat()
        self.cascaded_classifiers = []
        self.train_fvecs = np.dot(self.ftMat, self.train_X)    # Feature
                                                            Vector (47232 x 2468)


    def train(self, num_cascades, num_classifier):
        # Sort data
        self.train_posNum = int(np.sum(self.train_l))                      # Num
                                                        of neg training data
        self.train_negNum = self.train_l.size - self.train_posNum          # Num
                                                        of pos training data
        self.train_num = self.train_l.size                                 # Num
                                                        of total training data

        # Seperate pos/neg data feature vectors
        all_pos_fvecs = self.train_fvecs[:,(self.train_l==1).flatten()]
        all_neg_fvecs = self.train_fvecs[:,(self.train_l==0).flatten()]
        current_pos_fvecs = all_pos_fvecs
        current_neg_fvecs = all_neg_fvecs

        FP_train = []       # False Pos Rate
        FN_train = []       # False Neg Rate
        TP_train = []       # True Pos rate
        Ac_train = []       # Accuracy
        FP_test = []
        FN_test = []
        TP_test = []
        Ac_test = []

        # Cascade adaboost classifiers
        for i in range(num_cascades):
            print(f'Training {i+1} Adaboost classifier in the Cascade')
            current_adaboost = self.addAdaboostClassifier()
            current_adaboost.set_training_fvecs(current_pos_fvecs,
                                                    current_neg_fvecs)
            for j in range(num_classifier):
                print(f"Adding weak classifier {j+1}")
                current_adaboost.addWeakClassifier()
            falsePos_ind, FP, FN, TP, Ac = self.classifyTrainingData()
            FP_train.append(FP)
            FN_train.append(FN)
            TP_train.append(TP)
            Ac_train.append(Ac)
            current_neg_fvecs = all_neg_fvecs[:, falsePos_ind - self.
                                                    train_posNum]
            FP, FN, TP, Ac = self.classifyTestingData()
            FP_test.append(FP)
            FN_test.append(FN)
            TP_test.append(TP)
            Ac_test.append(Ac)
        return FP_train, FN_train, TP_train, Ac_train, FP_test, FN_test,
                                                TP_test, Ac_test

    def addAdaboostClassifier(self):
        classifier = AdaboostClassifier()
        classifier.setftMat(self.ftMat)
        self.cascaded_classifiers.append(classifier)
        return classifier

    def classifyTrainingData(self):
        print("Classifying training images")
        ftVecs = self.train_fvecs
        pos_inds = np.arange(self.train_num)
        for c in self.cascaded_classifiers:
            predictions = c.classify_ftVecs(ftVecs)
            ftVecs = ftVecs[:, predictions==1]
            pos_inds = pos_inds[predictions==1]
        # Sort TP, FP, FN
        FP_inds = pos_inds[np.take(self.train_l, pos_inds)==0]
        TP_num = np.sum(np.take(self.train_l, pos_inds))
        TP = TP_num / self.train_posNum
        FP = (pos_inds.size - TP_num) / self.train_negNum
        FN =  1 - TP
        w = self.train_posNum / (self.train_posNum + self.train_negNum) #
                                                     weight
        Ac = TP * w + (1-FP)*(1-w)
        print("FP = %.4f, FN = %.4f, TP = %.4f, Acc = %.4f" % (FP, FN, TP,
                                                      Ac))
        return FP_inds, FP, FN, TP, Ac

    def classifyTestingData(self):
        print("Classifying test images")
        ftVecs = np.dot(self.ftMat, self.test_X)
        test_posNum = int(np.sum(self.test_l))          # Num of neg test
                                                     data
        test_negNum = self.test_l.size - test_posNum    # Num of pos test
                                                     data
        test_num = self.test_l.size                     # Num of total
                                                     test data
        pos_inds = np.arange(test_num)
        for c in self.cascaded_classifiers:
            predictions = c.classify_ftVecs(ftVecs)
            ftVecs = ftVecs[:, predictions==1]
            pos_inds = pos_inds[predictions==1]
        TP_num = np.sum(np.take(self.test_l, pos_inds))
        TP = TP_num / test_posNum
        FP = (pos_inds.size - TP_num) / test_negNum
        FN =  1 - TP
        w = test_posNum / (test_posNum + test_negNum)
        Ac = TP * w + (1-FP)*(1-w)
        print("FP = %.4f, FN = %.4f, TP = %.4f, Acc = %.4f" % (FP, FN, TP,
                                                      Ac))
        return FP, FN, TP, Ac

class AdaboostClassifier():
    def __init__(self):
        self.train_l = None
        self.train_sortedInds = None
        self.train_ftVecs = None
        self.train_posNum = 0
        self.train_negNum = 0
        self.threshold = 1.0
        self.sample_weights = None
        self.weakClassifier_inds = np.array([], dtype=int)
        self.weakClassifier_polarities = np.array([])
        self.weakClassifier_threshs = np.array([])
        self.weakClassifier_weights = np.array([])
        self.weakClassifier_results = np.array([])
        self.weakClassifier_weighted_results = None

    def setftMat(self, ftMat):
        self.ftMat = ftMat

    def set_training_fvecs(self, pos_fvecs, neg_fvecs):
        print("Sorting current training features")
        self.train_posNum = pos_fvecs.shape[1]
        self.train_negNum = neg_fvecs.shape[1]
        self.train_l = np.hstack((np.ones(self.train_posNum), np.zeros(
                                      self.train_negNum)))
        self.train_ftVecs = np.hstack((pos_fvecs, neg_fvecs))
        self.train_sortedInds = np.argsort(self.train_ftVecs, axis=1)
        print(f'Pos sample: {self.train_posNum}, Neg sample: {self.
                                      train_negNum}')

    def addWeakClassifier(self):
        if self.weakClassifier_inds.size == 0:
            self.sample_weights = np.zeros(self.train_l.size, dtype=float)
            self.sample_weights.fill(1.0 / (2*self.train_negNum))
            self.sample_weights[self.train_l==1] = 1.0 / (2*self.
                                          train_posNum)
        else:
            self.sample_weights = self.sample_weights / np.sum(self.
                                          sample_weights)
        # Get the best classifier with the minimum error
        best_ft_ind, best_ft_pol, best_ft_thresh, best_ft_error,
                                      best_ft_results = self.
                                      getBestWeakClassifier()
        self.weakClassifier_inds = np.append(self.weakClassifier_inds,
                                      best_ft_ind)
        self.weakClassifier_polarities = np.append(self.
                                      weakClassifier_polarities,
                                      best_ft_pol)
        self.weakClassifier_threshs = np.append(self.
                                      weakClassifier_threshs,
                                      best_ft_thresh)
        # calculate confidence
        beta = best_ft_error / (1 - best_ft_error)
        # Trust Factor
        alpha = np.log(1 / np.abs(beta))
        self.weakClassifier_weights = np.append(self.
                                      weakClassifier_weights, alpha
                                      )
        e = np.abs(best_ft_results - self.train_l)
        # Update the weight
        self.sample_weights = self.sample_weights*beta**(1-e)
        # Adjust the threshold
        if self.weakClassifier_results.size == 0:
            self.weakClassifier_results = best_ft_results.reshape(-1,1)
        else:
            self.weakClassifier_results = np.hstack((self.
                                                     weakClassifier_results,
                                                     best_ft_results.reshape(-
                                                     1,1)))
        self.weakClassifier_weighted_results = np.dot(self.
                                                 weakClassifier_results, self.
                                                 weakClassifier_weights)
        self.threshold = min(self.weakClassifier_weighted_results[self.
                                                 train_l==1])

    def getBestWeakClassifier(self):
        ft_errs = np.zeros(NUM_FEATURES)
        ft_thresh = np.zeros(NUM_FEATURES)
        ft_pol = np.zeros(NUM_FEATURES)
        ft_sorted_ind = np.zeros(NUM_FEATURES, dtype=int)
        Tplus = np.sum(self.sample_weights[self.train_l==1])
        Tminus = np.sum(self.sample_weights[self.train_l==0])
        for r in range(NUM_FEATURES):
            # Get weights of sorted feature vectors
            sorted_weights = self.sample_weights[self.train_sortedInds[r,:
                                                     ]]
            sorted_l = self.train_l[self.train_sortedInds[r,:]]
            Splus = np.cumsum(sorted_l * sorted_weights)
            Sminus = np.cumsum(sorted_weights) - Splus
            Eplus = Splus + Tminus - Sminus
            Eminus = Sminus + Tplus - Splus
            # Calculate polarity
            polarities = np.zeros(self.train_posNum + self.train_negNum)
            polarities[Eplus > Eminus] = -1
            polarities[Eplus <= Eminus] = 1
            # Get errors
            errs = np.minimum(Eplus, Eminus)
            sorted_ind = np.argmin(errs)
            min_err_sample_ind = self.train_sortedInds[r,sorted_ind]
            min_err = np.min(errs)
            # Get threshold based on min err index
            threshold = self.train_ftVecs[r,min_err_sample_ind]
            polarities = polarities[sorted_ind]
            ft_errs[r] = min_err
            ft_thresh[r] = threshold
            ft_pol[r] = polarities
            ft_sorted_ind[r] = sorted_ind
        best_ft_ind = np.argmin(ft_errs)
        best_ft_pol = ft_pol[best_ft_ind]
        best_ft_thresh = ft_thresh[best_ft_ind]
        best_ft_error = ft_errs[best_ft_ind]
        best_ft_results = np.zeros(self.train_posNum + self.train_negNum)
        best_sorted_ind = ft_sorted_ind[best_ft_ind]
        if best_ft_pol == 1:
            best_ft_results[self.train_sortedInds[best_ft_ind,
                                                     best_sorted_ind:]] = 1
        else:
            best_ft_results[self.train_sortedInds[best_ft_ind, :
                                                     best_sorted_ind]] = 1
        return best_ft_ind, best_ft_pol, best_ft_thresh, best_ft_error,
                                             best_ft_results


    def classify_ftVecs(self, ftVecs):
        weakClassifiers = ftVecs[self.weakClassifier_inds,:]
        pol_vec = self.weakClassifier_polarities.reshape(-1,1)
        thresh_vec = self.weakClassifier_threshs.reshape(-1,1)
        # Predictions of weak classifiers
        weakClassifiers_preds = weakClassifiers * pol_vec > thresh_vec *
                                             pol_vec
        weakClassifiers_preds[weakClassifiers_preds==True] = 1
        weakClassifiers_preds[weakClassifiers_preds==False] = 0
        # Apply weights
        finalClassifier_result = np.dot(self.weakClassifier_weights,
                                             weakClassifiers_preds)
        final_preds = np.zeros(finalClassifier_result.size)
        final_preds[finalClassifier_result >= self.threshold] = 1
        return final_preds
```