Homework 10

Theory Question

1. Overfitting to the training data means that our model learns only the training data on a superficial level, and not the representations underneath that best describe our dataset. For example, in linear regression, if our classifier overfits, that means that it has not learned what is underneath but has learned some curve with passes through all the data points.

2. The problem is that if we try to sample from the latent space, this procedure is not differentiable. So to make it differentiable there is a need to change this sampling procedure. What is done is to not sample from the distribution $z \sim \mathcal{N}(\mu, \sigma^2)$ but we sample from a zero-mean unit variance normal $\epsilon \sim \mathcal{N}(0, 1)$. So now we add our mean μ and multiply it by our variance. So now, $z = \mu + \epsilon\sigma$. Now this operation is differentiable, so essentially we change the way we sample from the normal, through making use of the zero mean unit variance normal distribution.

Task 1

For this and Task 2 we will be using the FacePix dataset provided with the homework files. At the very start, we associate each image to a label represented by the first part of the image file's name. In this case we have 30 different classes (30 faces of 30 different people).

PCA

We begin by vectorizing each image, turning it from, in this case of size 128×128, to a vector of length 16384. Then we can compute a mean image given by:

$$\mathbf{m} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}_i \tag{1}$$

Where N is the number of images and $\mathbf{x}_i$ is the vectorized i$^{\text{th}}$ image. With this we can now compute the covariance matrix. The covariance matrix is given by:

$$\mathbf{C} = \frac{1}{N}\sum_{i=1}^{N}\{(\mathbf{x}_i - \mathbf{m})(\mathbf{x}_i - \mathbf{m})^T\} \tag{2}$$

After substracting the mean from the vectorized images, we normalize them each, ensuring that the norm of each image/row is equal to 1. The next step is to get the SVD decomposition of this covariance matrix, however, we run into issues since this covariance matrix is of size 16384×16384, making it computational resource intensive to be able to do this decomposition. We make use of the computational trick to aleviate this. Our new covariance matrix is now:

$$\mathbf{C} = \frac{1}{N}\sum_{i=1}^{N}\{(\mathbf{x}_i - \mathbf{m})^T(\mathbf{x}_i - \mathbf{m})\} \tag{3}$$

Which results in a $N \times N$ size covariance matrix. We perform SVD decomposition on this smaller size covariance matrix. Now, we are interested in the eigenvectors of the bigger matrix, given by $\mathbf{w}$. We know that our smaller matrix eigenvectors are related to these by:

$$\mathbf{w} = \mathbf{X}\mathbf{v} \tag{4}$$

Where $\mathbf{X}$ is the entire stack of images and $\mathbf{v}$ is the eigenvectors from the smaller covariance matrix. Now, all that is left is for us to normalize $\mathbf{W}$, and to pick the first p eigenvectors.
Now that we have the eigenvectors, we can compute the projection of our images into the eigenvectors. This projection is now our training "features".

Nearest Neighbor To be able to classify our faces, we make use of nearest neighbor algorithm. This consists of getting the distance from a data point to all the points of the training set, we then take the closest training set data point and assign our test dataset point the same label as this closest training set data point.
For the testing set, we follow the same process. First, we retain the mean image, that is calculated from the training set, and we substract it from the test data as well. Then we get the projection of all our testing set data points to the space generated by the eigenvectors $\mathbf{w}$. We use nearest neighbor to classify these testing data points.

UMAP Embeddings The training and testing features undergo dimensionality reduction with UMAP library, resulting in only 2 dimensions. For the training set, the classes are set with the ground truth labels and for the test set the classes are indicated by the predicted classes. We plot these embeddings in a 2D graph.

LDA

We begin similarly to PCA. The main difference is that we now need to divide the images into their respective classes. We now calculate the per class mean given by:

$$\mathbf{m}_i = \frac{1}{|\mathcal{C}_i|}\sum_{n=1}^{|\mathcal{C}_i|}\mathbf{x}_n \tag{5}$$

Where $\mathcal{C}_i$ is the i$^{\text{th}}$ class. We then make use of all the class means and the global mean to get the between class scatter and the within class scatter, given by:

$$\mathbf{S}_B = \frac{1}{|\mathcal{C}|}\sum_{i=1}^{|\mathcal{C}|}\{(\mathbf{m}_i - \mathbf{m})^T(\mathbf{m}_i - \mathbf{m})\} \tag{6}$$

$$\mathbf{S}_W = \frac{1}{|\mathcal{C}|}\sum_{i=1}^{|\mathcal{C}|}\frac{1}{|\mathcal{C}_i|}\sum_{n=1}^{|\mathcal{C}_i|}\{(\mathbf{x}_n - \mathbf{m}_i)^T(\mathbf{x}_n - \mathbf{m}_i)\} \tag{7}$$

Where $\mathbf{S}_B$ is the between class scatter matrix and $\mathbf{S}_W$ is the within class scatter matrix. We apply the Yu-Yang algorithm, which consists of starting with a similar step to the computational trick for calculating the between class scatter matrix. We obtain $\mathbf{S}_B$ through:

$$\mathbf{S}_B = \frac{1}{|\mathcal{C}|}\sum_{i=1}^{|\mathcal{C}|}\{(\mathbf{m}_i - \mathbf{m})(\mathbf{m}_i - \mathbf{m})^T\} \tag{8}$$

Which gives us the same number of eigenvectors as images in our training set after following the same steps to get the actual eigenvectors similar to PCA. Then we calculate a matrix $\mathbf{Z}$, given by:

$$\mathbf{Z} = \mathbf{w}\mathbf{D}^{-1/2} \tag{9}$$

Where $\mathbf{D}$ is the diagonal matrix built from the eigenvalues and $\mathbf{w}$ is the eigenvectors obtained previously. One more thing to consider in this step is that we are discarding the smallest 5 eigenvalues and their corresponding eigenvectors. To get the eigenvectors of $\mathbf{S}_w$ we make use of $\mathbf{Z}$ and the computational trick.

$$\mathbf{Z}_X = \mathbf{Z}^T\mathbf{X} \tag{10}$$

Now we can get $\mathbf{S}_w$ from this:

$$\mathbf{S}_w = \mathbf{Z}_X\mathbf{Z}_X^T \tag{11}$$

Now we get the eigenvectors of $\mathbf{S}_w$ and only retain p of them, similar to PCA. Finally, the classification step is done with the same steps as for PCA, the data points are all projected to the space of the eigenvectors, and then nearest neighbor is used to classify the test set data points. UMAP embeddings are also obtained for this.

Task 2: Autoencoder

We make the equivalence of the autoencoder weights at epoch e with the p chosen eigenvectors from PCA and LDA so we can graph both of them in one single graph. With the given autoencoder weights, we get the latent space variables for each of images in the training and testing set, and it is this latent space variables that we call our features. We then use the previously mentioned nearest neighbor to classify get the closest train feature and classify it with that label. We repeat this process for all 3 weights given: 3, 8 and 16 epochs. We also do dimensionality reduction to get the UMAP embeddings and plot them.

Task 3: Adaboost Cascade

Feature extraction

To begin with, we extract horizontal and vertical features with a 1D Haar filter. These filters look like: $\begin{pmatrix}-1 & -1 & -1 & 1 & 1 & 1\end{pmatrix}$ for horizontal and $\begin{pmatrix}-1 & -1 & -1 & 1 & 1 & 1\end{pmatrix}^T$ for vertical features. We convolve the image with several of these filters, with sizes ranging from 2 to the width/height of the image. We append every result of the convolution to a vector, we treat these as our features. We repeat this process for every image in both the training and testing set.

Weak Classifiers

Now that we have all the features for our training set and a given set of weights, we proceed to loop through all the features. We now sort them, and find the error:

$$\epsilon_1 = S^+ + T^- - S^- \tag{12}$$

However, we need to consider the inverse polarity for our thresholding, so we consider another error:

$$\epsilon_2 = S^- + T^+ - S^+ \tag{13}$$

In both, S is the cumulative sum of the weights for the positive or negative samples, and T is the total sum of the positive or negative weights. We now find the minimum of each error and find whether this minimum belongs to ϵ_1 or ϵ_2, as this will determine the polarity for our thresholding. We find the feature where the minimum error is in and store it. We compare this minimum error value to a minimum value found across all features to only keep the best weak classifier. Once this best weak classifier is found, we save the polarity, threshold value, which feature is involved, and the predictions on the training set.

Cascade/Strong classifier

We initialize the weights, making sure that positive and negative labels have collective weights of 1/2. All the positive labels will then have the same weight, so they are normalized depending on how many there are, same for negative labels. Now we take the entire training feature set and obtain a weak classifier. We then take the predictions from the weak classifier and the error from this weak classifier to update our weights for the next weak classifier.

$$\begin{aligned} \beta_t &= \frac{\epsilon_t}{1-\epsilon_t} \\ \alpha_t &= \frac{1}{2}\ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right) \end{aligned} \tag{14}$$

We use this to update the weights for our next weak classifier:

$$\begin{aligned} D_{t+1}(x_i) &= \frac{D_t(x_i)\{\beta_t(h(x_i)-y_i)\}}{Z_t} \\ Z_t &= \sum_{i=1}^{m} D_t(x_i)\{\beta_t(h(x_i)-y_i)\} \end{aligned} \tag{15}$$

As for the trust factor, α_t, we will use it for the final predictions. We repeat this process for as many classifiers we want our cascade to have. Our final classifier for the cascade will be given by:

$$H(x_i) = \sum_{t=1}^{N} \alpha_t h_t(x_i) \geq \sum_{t=1}^{N} \frac{1}{2}\alpha_t \tag{16}$$

Training

For training, we get our first strong classifier, then depending on the final predictions of this classifier, we revise the training set. We keep all the positive samples, but for the negative samples, we only keep those that have been misclassified. We revise both the features and the labels for the next cascade. We repeat this for as many cascades as desired.

Inference

For inference, we take the features from the testing set and put them through the first cascade. We obtain the final predictions, and in a similar manner to when we were training, we need to revise the testing set. Our revision consists of keeping only the positive predicted features. After this, we put it through the next cascade and so on until the final cascade.

Results

Task 1 & 2: PCA, LDA and Autoencoder

As we can see in Fig. 1, LDA outperforms PCA in every single value of p tested. It also achieved 100% classification accuracy, while PCA did not. We also see that PCA tends to not always have an increase in accuracy as the value of p increases, as we can see for $p = 7$, $p = 13$. In contrast, LDA does increase in as the value of p increases. Now, as for the autoencoder, it starts off about as good as LDA, and better than PCA, however it quickly loses ground to LDA, as at $p = 8$ it is worse than LDA, but better than PCA. It also reaches 100% accuracy, which PCA was not able to do. As for the UMAP embeddings, we can see how as we increase p these tend to be drawn together, in low p values these are very spread out and as a result our nearest neighbor misclassifies them. This is common to all 3: PCA, LDA and autoencoder. As we can see, the best performing classifier is LDA, and for its $p = 15$ embeddings we can see that these are closely bunched together, compared to PCA and autoencoder.

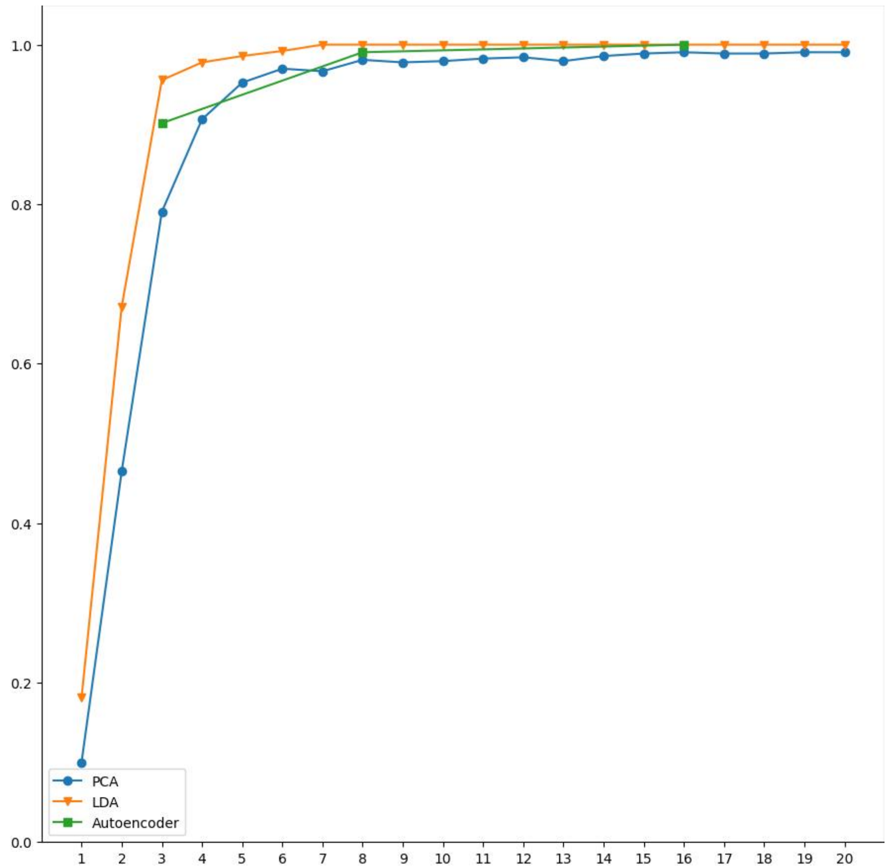
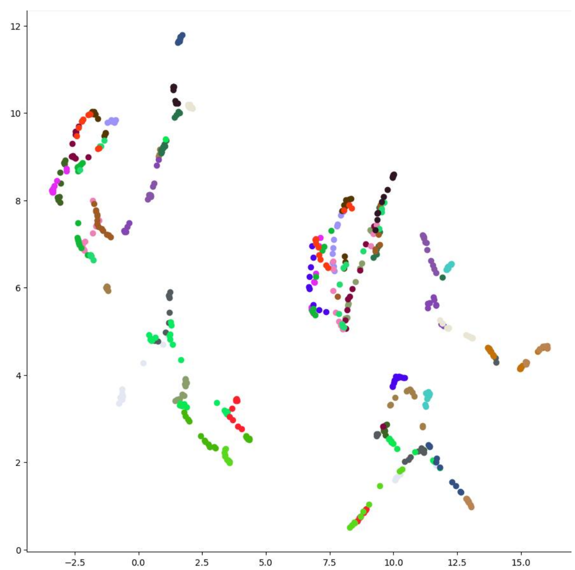
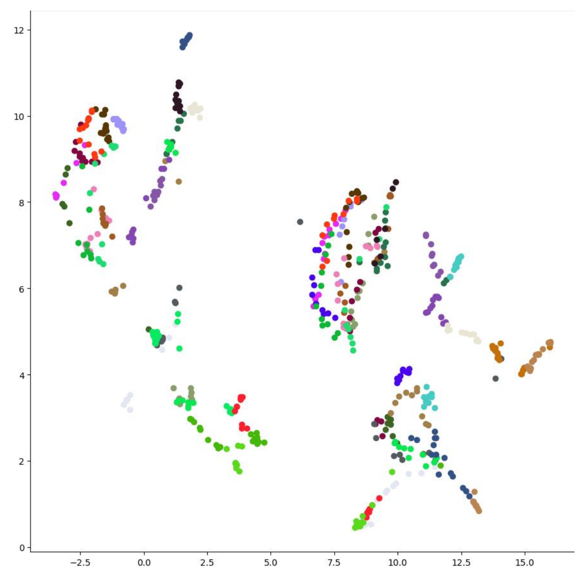


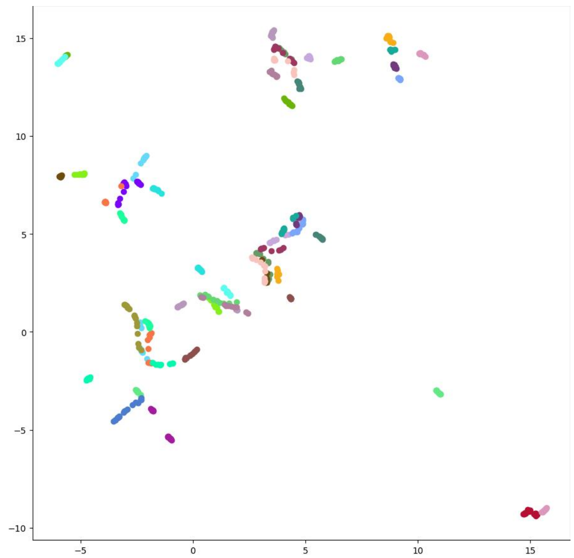
Figure 1: Accuracy as a function of p.



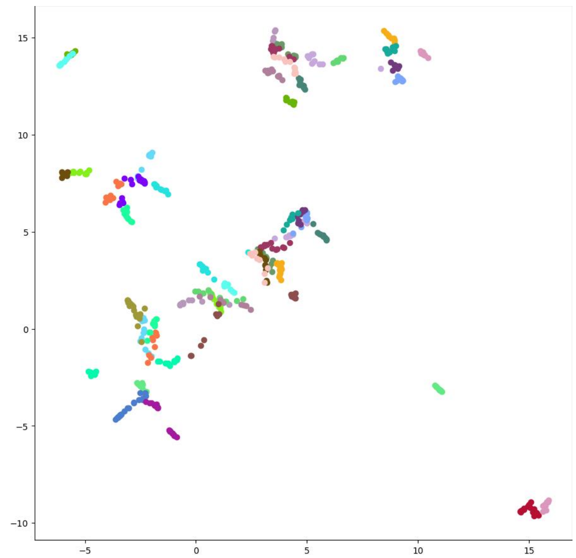
(a) PCA UMAP embeddings for $p = 5$, training set.



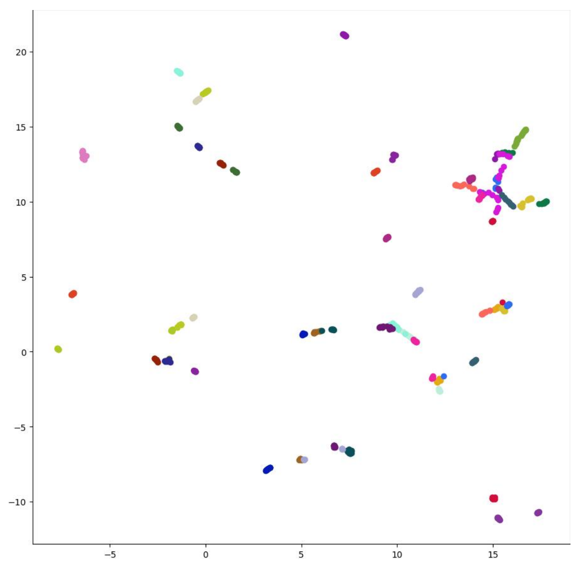
(b) PCA UMAP embeddings for $p = 5$, testing set.



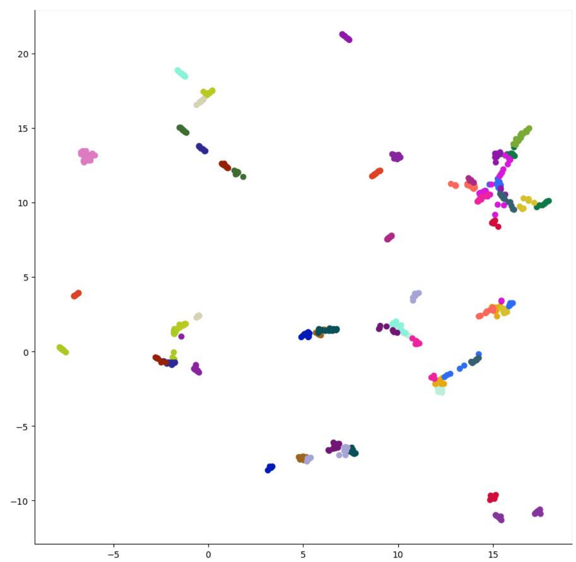
(c) PCA UMAP embeddings for $p = 15$, training set.



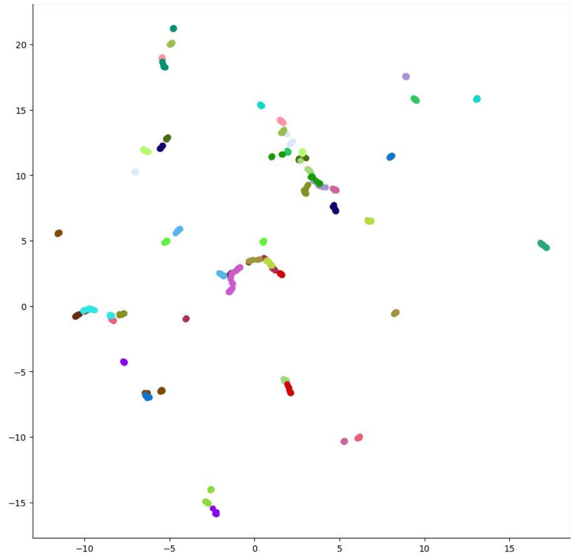
(d) PCA UMAP embeddings for $p = 15$, testing set.



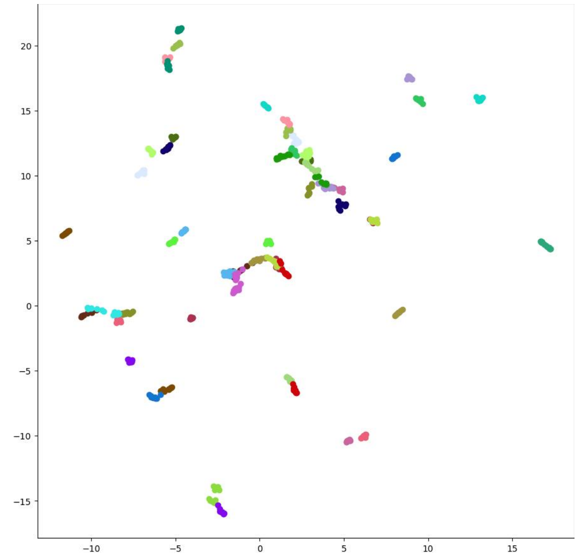
(e) LDA UMAP embeddings for $p = 5$, training set.



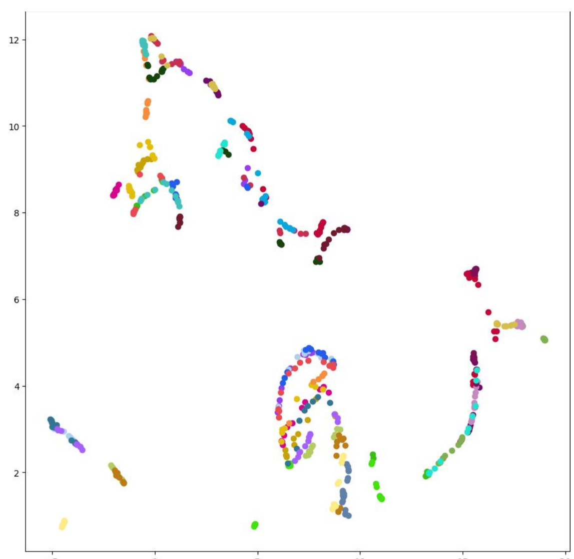
(f) LDA UMAP embeddings for $p = 5$, testing set.



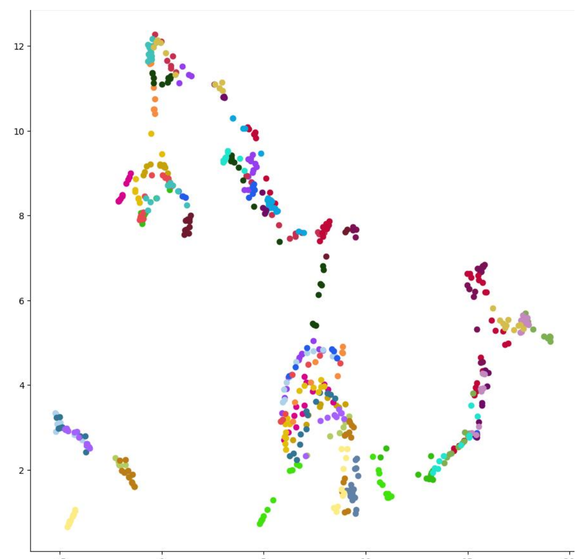
(g) LDA UMAP embeddings for $p = 15$, training set.



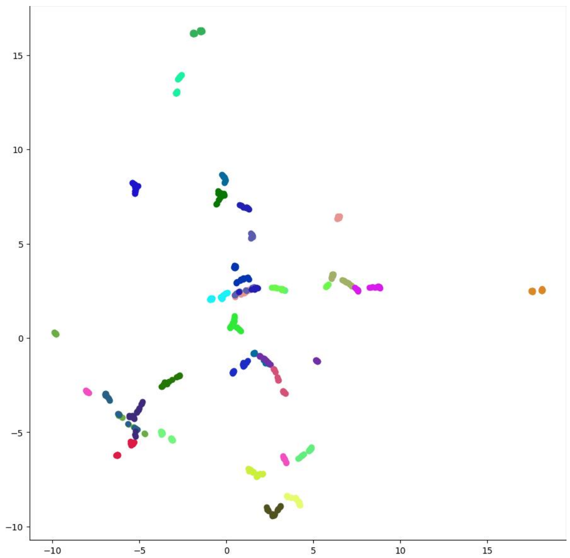
(h) LDA UMAP embeddings for $p = 15$, testing set.



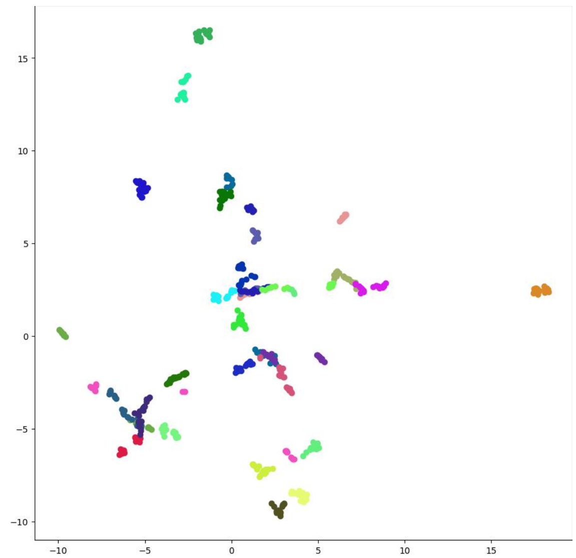
(i) Autoencoder UMAP embeddings for $p = 3$, training set.



(j) Autoencoder UMAP embeddings for $p = 3$, testing set.



(k) Autoencoder UMAP embeddings for $p = 16$, training set.



(l) Autoencoder UMAP embeddings for $p = 16$, testing set.

Task 3: Adaboost Cascade

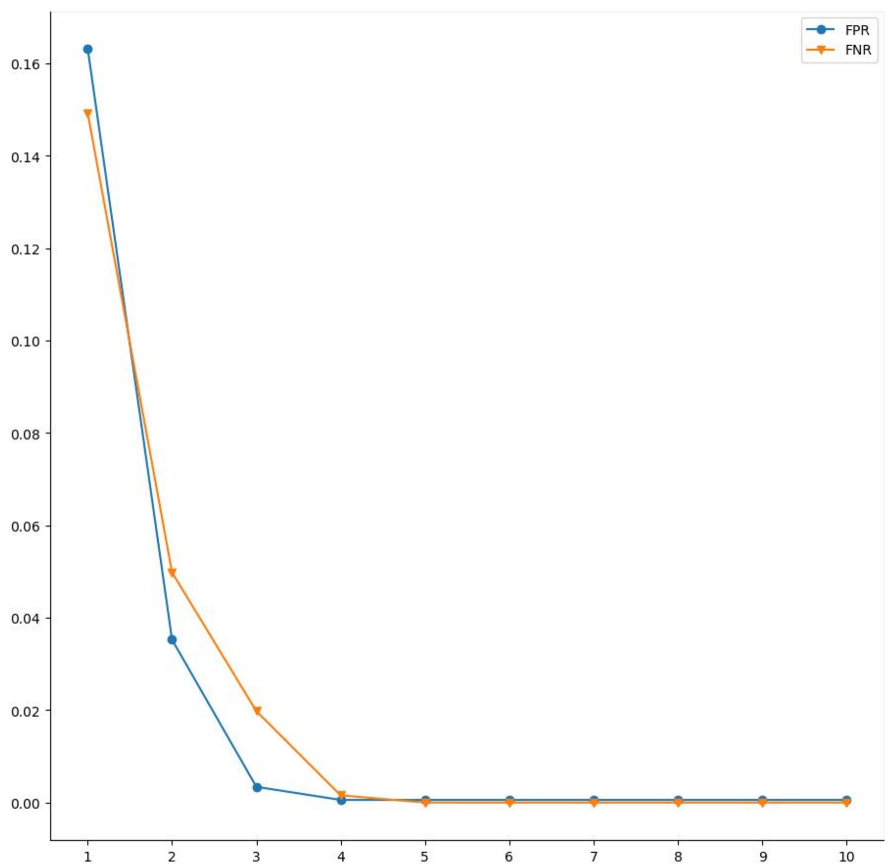


Figure 2: FPR and FNR during training, as a function of the stages.

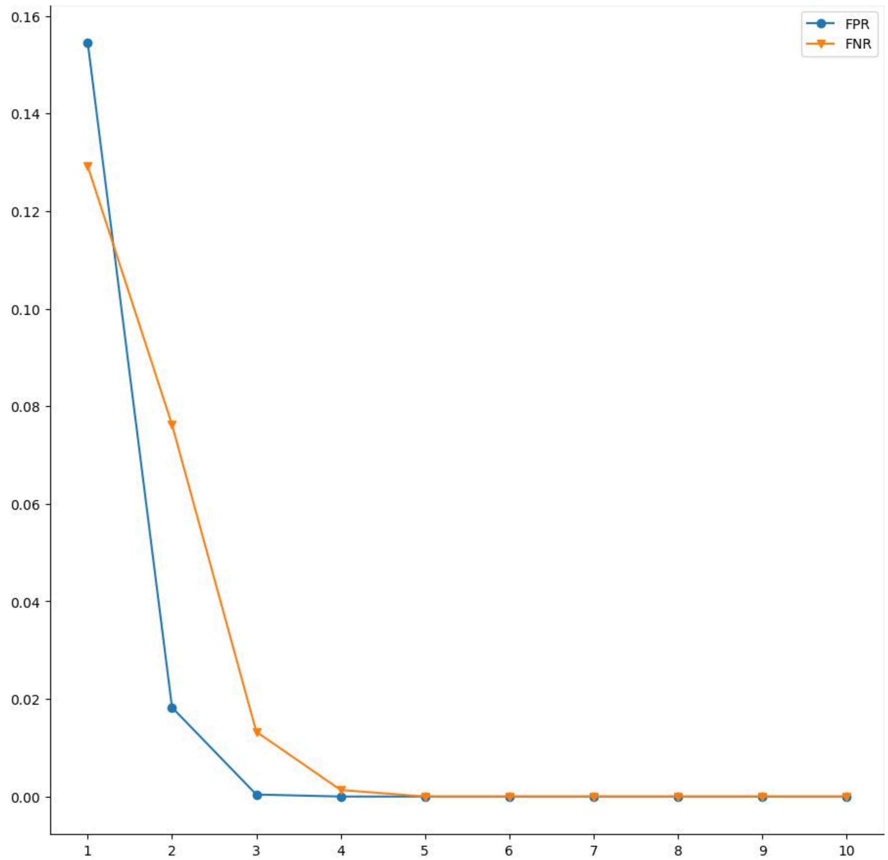


Figure 3: FPR and FNR during training, as a function of the stages.

Source code

```

import cv2
import os
import torch
import numpy as np
import matplotlib.pyplot as plt
import umap.umap_ as umap
from autoencoder import *

train_path = "FaceRecognition/train"
train_list = [x for x in os.listdir(train_path) if x.endswith(".png")]
train_list = sorted(train_list)

test_path = "FaceRecognition/test"
test_list = [x for x in os.listdir(test_path) if x.endswith(".png")]
test_list = sorted(test_list)

def load_data_autoencoder(TRAIN_DATA_PATH, EVAL_DATA_PATH, p):
    # this is just to load the autoencoder, and then use it to obtain the latent variables for each image,
    # these now become our training features/testing features
    model = Autoencoder(p)
```

```
    LOAD_PATH = f'weights/model_{p}.pt'
    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

def get_img_data(file_list, img_path):
    # this is just to obtain the flattened images and their corresponding
   label,
    # we obtain the label from the name of the image file
    mat = []
    labels = []
    for file in file_list:
        img = cv2.imread(os.path.join(img_path,file),cv2.IMREAD_GRAYSCALE)
        labels.append(int(file[:2]))
        mat.append(img.reshape(-1))
    return np.array(mat), np.array(labels)

def normalize_matrix(img_data):
    # we normalize the rows
    norm_ = np.linalg.norm(img_data,axis=1)[:,None]
    n_imgdata = img_data/norm_
    return n_imgdata

def get_covariance(img_data):
    # this is to get the covariance matrix
    covariance = img_data.T @ img_data
    covariance /= (len(covariance) - 1)
```

```python
    return covariance

def get_pca(img_data, p=10):
    # substract the global mean, then normalize the data
    img_data = img_data - img_data.mean(axis=0)[None,...]
    normed_data = normalize_matrix(img_data)
    normed_dataT = normed_data.T
    # get the covariance then get the svd decomposition, proceed with the computational trick and pick only the p first eigenvectors
    cov = get_covariance(normed_dataT)
    _,_, v = np.linalg.svd(cov)
    # up next is the computational trick
    # I transpose the matrix here to reuse our normalization function,
    # also because I understand it better if it has shape (N,16000) over (16000,N)
    # this is seen in the LDA function next up
    W = (normed_dataT @ v).T
    normed_W = normalize_matrix(W)
    return normed_W[:p]

def get_lda(img_data,labels,p=10):
    class_mean = {}
    per_class = {}
    # we get the class means and put them into that dictionary
    global_mean = img_data.mean(axis=0)[None,...]
    C_mean = []
    for id_class in range(labels.min(),labels.max()+1):
        # we append the mean image per class to a matrix that has all of them
        cls_img_data = img_data[labels == id_class]
        per_class[str(id_class)] = cls_img_data.shape[0]
        class_mean[str(id_class)] = cls_img_data.mean(axis=0)[None,...]
        C_mean.append(cls_img_data.mean(axis=0))
    # we use the yu-yang algorithm and also the computational trick in here
    C_mean = np.array(C_mean)
    C_mean_m = C_mean - global_mean
    img_data = img_data - global_mean
    #print(C_mean_m.shape)
    S_b_c = C_mean_m @ C_mean_m.T
    #print(S_b_c.shape)
    _, d, v = np.linalg.svd(S_b_c)
    # get the transpose to reuse our function, and also we ignore the last 5 eigenvalues and eigenvectors, ignoring the last 5 worked well
    w = (C_mean_m.T @ v).T
    normed_w = normalize_matrix(w)[:-5]
    diagd = np.diag(d[:-5])
    z = np.linalg.inv(diagd) @ normed_w
    # again I keep the matrices of shape (N,16000) as it is easier for me to grasp
    z_x = z @ img_data.T
    new_s = z_x @ z_x.T
    _, _, v2 = np.linalg.svd(new_s)
    new_w = (z.T @ v2).T
```

```

    # print(new_w.shape)
    return new_w[:p]

def NN(xprojection_train, y_train, xprojection_test):
    # implementation of nearest neighbor, we make use of broadcasting to keep it to only one line of code
    distances = np.linalg.norm(xprojection_test[:,None,:] - xprojection_train[None,...], axis=2)
    # get the index where the minimum happens
    idx = np.argmin(distances,axis=1)
    # return the label corresponding to it
    return y_train[idx]

def get_accuracy(y_test, y_pred):
    # get the accuracy as described in the homework pdf
    correct = (y_test == y_pred).sum()
    return correct/y_test.shape[0]


def plot_umap_embeddings(x_train, y_train, x_test,y_pred, mode='pca', p=10, num_classes=30):
    # plot the umap embeddings and save the plot
    umap_t = umap.UMAP(n_components=2, random_state=0)
    x_train_emb = umap_t.fit_transform(x_train)
    x_test_emb = umap_t.transform(x_test)

    colors = np.random.random((num_classes,3))

    # train embeddings
    fig = plt.figure(figsize=(11,11))
    for id in range(1,num_classes):

        x_train_emb_c = x_train_emb[y_train == id]
        plt.scatter(x_train_emb_c[:,0],x_train_emb_c[:,1], color=colors[id])
    plt.savefig(f'imgs/umap_emb_{mode}_p{p}_train.png', bbox_inches='tight', pad_inches=0)

    plt.close()
    plt.clf()
    # test embeddings
    fig = plt.figure(figsize=(11,11))
    for id in range(1,num_classes):
        x_test_emb_c = x_test_emb[y_pred == id]
        plt.scatter(x_test_emb_c[:,0],x_test_emb_c[:,1], color=colors[id])
    plt.savefig(f'imgs/umap_emb_{mode}_p{p}_test.png', bbox_inches='tight', pad_inches=0)
    plt.close()
    plt.clf()

def classifier(x_train, y_train, x_test, y_test, mode='pca', p=10, plot_embeddings=False):
```

```
    # this is to build the classifier with the training and testing
   features as well as plot the umap embeddings
    g_mean = x_train.mean(axis=0)[None,...]
    # we set 3 modes, LDA PCA and autoencoder, to not have to remake other
    functions
    if mode!="autoencoder":
        # get the projections to the eigenvectors
        if mode=='pca':
            eigs = get_pca(x_train, p=p)
        elif mode == 'lda':
            eigs = get_lda(x_train, y_train, p=p)
        x_train = x_train - g_mean
        x_test = x_test - g_mean

        xp_train = (eigs @ x_train.T).T
        xp_test = (eigs @ x_test.T).T
    else:
        # in the case of the autoencoder we don't need to do anything, as
   we already got our training features from the autoencoder itself
        xp_train = x_train
        xp_test = x_test
    # we use nearest neighbor
    y_pred = NN(xp_train, y_train, xp_test)
    # get the accuracy
    acc = get_accuracy(y_test, y_pred)

    if plot_embeddings:
        plot_umap_embeddings(xp_train, y_train, xp_test,y_pred, mode=mode,
    p=p)
    return acc

x_train, y_train = get_img_data(train_list,train_path)
x_test, y_test = get_img_data(test_list, test_path)
# values of p we will plot
ps = np.arange(1,21,1)
# save the accuracy of both
pca_acc = []
lda_acc = []

for p in ps:
    plot_emb = False
    if p % 5 == 0:
        plot_emb = True

    # pca
    pca_a = classifier(x_train, y_train, x_test,y_test, mode='pca', p=p,
   plot_embeddings=plot_emb)
    pca_acc.append(pca_a)

    # lda
    lda_a = classifier(x_train, y_train, x_test,y_test, mode='lda', p=p,
   plot_embeddings=plot_emb)
    lda_acc.append(lda_a)

```

```
p_ep = np.array([3,8,16])
aenc_acc = []
for p in p_ep:
    # this is for our autoencoder, we plot on every value of p since
    theres only 3
    x_train, y_train, x_test, y_test = load_data_autoencoder(train_path,
    test_path, p)
    a_acc = classifier(x_train, y_train, x_test, y_test, mode='autoencoder
    ', p=p, plot_embeddings=True)
    aenc_acc.append(a_acc)


# plot accuracies as function of p
fig = plt.figure(figsize=(11,11))
plt.plot(ps,pca_acc, marker="o",label="PCA")
plt.plot(ps,lda_acc, marker="v",label="LDA")
plt.plot(p_ep,aenc_acc, marker="s",label="Autoencoder")

plt.xlim(0,21,1);
plt.ylim(0,1.05)
plt.xticks(ps)
plt.legend()
plt.savefig(f'imgs/accs_p.png', bbox_inches='tight', pad_inches=0)
plt.close()

# some functions here have been based on the 2022's best solutions

def get_features(image):
    # image is grayscale
    h, w = image.shape
    image = (image/255).astype(float)
    sizes_vert = np.arange(2, h, 2)
    sizes_hor = np.arange(2, w, 2)
    features = []
    # vertical features
    for f_size in sizes_vert:
        # pad in vertical direction only
        image_p = cv2.copyMakeBorder(image, f_size//2, f_size//2,0,0, cv2.
    BORDER_CONSTANT, 0)
        for jdx in range(f_size//2,image_p.shape[0]-f_size//2):
            for idx in range(image_p.shape[1]):
                neg = image_p[jdx - f_size//2:jdx, idx].sum()
                pos = image_p[jdx:jdx + f_size//2 + 1, idx].sum()
                feature = pos - neg
                features.append(feature)
    # horizontal features
    for f_size in sizes_hor:
        # pad in horizontal direction only
        image_p = cv2.copyMakeBorder(image, 0, 0, f_size//2, f_size//2,
    cv2.BORDER_CONSTANT, 0)
        for jdx in range(image.shape[0]):
            for idx in range(f_size//2,image_p.shape[1]-f_size//2):
                neg = image_p[jdx, idx - f_size//2:idx].sum()
                pos = image_p[jdx, idx:idx + f_size//2 + 1].sum()
```

```
                feature = pos - neg
                features.append(feature)
    # now we have all our features
    features = np.array(features)

    return features

def get_car_data(path, num=1):
    # this is just to load our images and get the features
    file_list = [x for x in os.listdir(path) if x.endswith("png")]
    feature_list = []
    for file in file_list:
        img = cv2.imread(os.path.join(path,file), cv2.IMREAD_GRAYSCALE)
        features = get_features(img)
        feature_list.append(features)
    feature_list = np.array(feature_list)
    # we just get the labels like this, for negative we just change it to
    0
    class_label = np.ones(len(file_list)) * num
    return feature_list, class_label

train_pos_path = "CarDetection/train/positive"
train_neg_path = "CarDetection/train/negative"
test_pos_path = "CarDetection/test/positive"
test_neg_path = "CarDetection/test/negative"

# we make sure to mantain the distinction between positive and negative
x_train_pos, y_train_pos = get_car_data(train_pos_path,1)
x_train_neg, y_train_neg = get_car_data(train_neg_path,0)


x_test_pos, y_test_pos = get_car_data(test_pos_path,1)
x_test_neg, y_test_neg = get_car_data(test_neg_path,0)


def weak_classifier(features,labels,weights):
    # this is to get a weak classifier,
    # to start we set a high error, since it ensures that it will always
    change in the first iteration
    cls_error = np.inf

    for idxs in range(features.shape[1]):
        # loop through features and order them,
        # also order the labels and the weights accordingly
        feature = features[:,idxs]
        idxs_sort = np.argsort(feature)
        feature_sort = feature[idxs_sort]
        labels_sort = labels[idxs_sort,0]
        weights_sort = weights[idxs_sort,0]
        # this is to get the errors, we just set the negative weights to 0
     for the cumulative positive sum and viceversa
        weights_pos = weights_sort.astype(float)
        weights_pos[labels_sort == 0] = 0

```

```
        weights_neg = weights_sort.astype(float)
        weights_neg[labels_sort == 1] = 0

        total_pos_w = weights_pos.sum()
        total_neg_w = weights_neg.sum()

        sum_pos = np.cumsum(weights_pos)
        sum_neg = np.cumsum(weights_neg)
        error_1 = sum_pos + total_neg_w - sum_neg
        error_2 = sum_neg + total_pos_w - sum_pos

        min_err_1 = np.min(error_1).astype(float)
        min_err_2 = np.min(error_2).astype(float)
        # get the minimum of both, and the minimum between those
        min_err = np.min([min_err_1, min_err_2])

        if min_err < cls_error:
            # this is for our best classifier
            cls_error = min_err
            if min_err_1 <= min_err_2:
                polarity = 1
            else:
                polarity = 0
            idx_feature = idxs
            preds = np.zeros_like(labels_sort)
            if polarity == 1:
                threshold = feature_sort[np.argmin(error_1)]
                preds[feature >= threshold] = 1

            else:
                threshold = feature_sort[np.argmin(error_2)]
                preds[feature < threshold] = 1


    return [idx_feature,threshold, polarity,cls_error, preds]

def cascade(x_train_pos, x_train_neg, y_train_pos, y_train_neg, num_iters
    =5):
    # this is for the strong classifier
    cascade_thresholds = []
    cascade_feature_idxs = []
    cascade_polarity = []
    cascade_preds = []
    cascade_error = []
    cascade_tfs = []
    # we initialize the weights here
    weights_pos = np.ones((y_train_pos.shape[0],1)) * (1/y_train_pos.shape
    [0])
    weights_neg = np.ones((y_train_neg.shape[0],1)) * (1/y_train_neg.shape
    [0])
    # now we stack everything to use later
    weights = np.vstack((weights_pos,weights_neg))
    features = np.vstack((x_train_pos, x_train_neg))
    labels = np.vstack((y_train_pos[:,None], y_train_neg[:,None]))
```

```
    final_preds = np.zeros_like(labels)[:,0]
    # normalize the weights
    weights = weights/weights.sum()
    for idx in range(num_iters):
        # get each weak classifier and then adjust the weights according to their trust factors
        idx_feature,threshold, pol, err, preds = weak_classifier(features,labels,weights)
        # use the error from the weak classifier to update the weights
        eps = err
        beta = eps / (1 - eps + 1e-16)
        tf = ((np.log((1 - eps + 1e-16)/(eps + 1e-16))) * 0.5)
        new_weights = weights * beta**(np.abs(labels[:,0]-preds))[:,None]
        # normalize our new weights
        new_weights = new_weights/new_weights.sum()
        # sum to the final predictions, this is just to keep tabs on it
        final_preds = final_preds + tf*preds
        cascade_thresholds.append(threshold)
        cascade_feature_idxs.append(idx_feature)
        cascade_polarity.append(pol)
        cascade_preds.append(preds)
        cascade_error.append(err)
        cascade_tfs.append(tf)

        # update the weights
        weights = new_weights

    # get final cascade outputs
    cascade_tfs_ = np.array(cascade_tfs)

    ths_tf = cascade_tfs_.sum()/2

    final_cascade_preds = np.zeros_like(labels)[:,0]
    final_cascade_preds[final_preds >= ths_tf] = 1
    final_cascade_preds[final_preds < ths_tf] = -1

    # get the negative indexes and positive indexes
    idx_neg = np.argwhere(labels == 0)[:,0]
    idx_pos = np.argwhere(labels == 1)[:,0]
    # get the fpr and fnr
    fpr = (final_cascade_preds[idx_neg] == 1).sum() / y_train_neg.shape[0]
    fnr = (final_cascade_preds[idx_pos] == 0).sum() / y_train_pos.shape[0]
    # now we decide which to keep and which to discard,
    # we only keep the negative samples that have been misclassified
    neg_preds = final_cascade_preds[idx_neg]
    idx_keep = np.argwhere(neg_preds == 1)[:,0]
    # return these since we will use them for the next strong classifier
    new_y_train_neg = y_train_neg[idx_keep]
    new_x_train_neg = x_train_neg[idx_keep]

    strong_classifier = [cascade_thresholds, cascade_feature_idxs, cascade_polarity,cascade_tfs,cascade_error]

    return new_x_train_neg, new_y_train_neg, strong_classifier, fpr, fnr
```

```


# train cascade

num_casc = np.arange(1,11,1)
fpr_list = []
fnr_list = []
classifiers = []
fpr_cdx = 1
fnr_cdx = 1
new_x_train_neg = x_train_neg
new_y_train_neg = y_train_neg
for cdx in num_casc:
    # we return the new features and subsequent strong classifiers use those
    new_x_train_neg_, new_y_train_neg_, strong_class_stage,fpr,fnr = cascade(x_train_pos, new_x_train_neg, y_train_pos, new_y_train_neg, num_iters=2)
    classifiers.append(strong_class_stage)
    fpr_cdx = fpr_cdx*fpr
    fnr_cdx = fnr_cdx*fnr
    fpr_list.append(fpr_cdx)
    fnr_list.append(fnr_cdx)

    new_x_train_neg = new_x_train_neg_
    new_y_train_neg = new_y_train_neg_
    # stop training if we run out of negative samples
    if len(new_x_train_neg) == 0:
        break

classifiers = np.array(classifiers)

# plot fpr and fnr
fig = plt.figure(figsize=(11,11))
plt.plot(num_casc,fpr_list, marker="o",label="FPR")
plt.plot(num_casc,fnr_list, marker="v",label="FNR")

plt.xticks(num_casc)
plt.legend()
plt.savefig(f'imgs/cascade_train.png', bbox_inches='tight', pad_inches=0)
plt.close()

# inference
# stack features, labels
features_test = np.vstack((x_test_pos, x_test_neg))
labels_test = np.vstack((y_test_pos[:,None], y_test_neg[:,None]))
fpr_test_list = []
fnr_test_list = []
fnr_cdx = 1
fpr_cdx = 1
num_tpos = y_test_pos.shape[0]
num_tneg = y_test_neg.shape[0]

for classifier in classifiers:
```

```
    # loop through the stages of the cascade
    preds_test_cascade = np.zeros_like(labels_test)[:,0]
    # get the following from each strong classifier
    feature_idxs = classifier[1]
    thresholds = classifier[0]
    polarities = classifier[3]
    trust_factors = classifier[4]

    for cdx in range(feature_idxs.shape[0]):
        # now loop through each weak classifier and start building the strong classifier output
        preds_test_classifier = np.zeros_like(labels_test)[:,0]
        feature_idx = int(feature_idxs[cdx])
        threshold = thresholds[cdx]
        polarity = polarities[cdx]
        trust_factor = trust_factors[cdx]

        feature_test = features_test[:,feature_idx]

        if polarity == 1:
            preds_test_classifier[feature_test >= threshold] = 1
            preds_test_classifier[feature_test < threshold] = 0
        else:
            preds_test_classifier[feature_test < threshold] = 1
            preds_test_classifier[feature_test >= threshold] = 0

        preds_test_cascade = preds_test_cascade + preds_test_classifier*trust_factor
    # now we adjust the final output
    ths_tf = trust_factors.sum()/2

    preds_test_cascade[preds_test_cascade >= ths_tf] = 1
    preds_test_cascade[preds_test_cascade < ths_tf] = 0
    # get the fpr and fnr
    idx_pos = np.argwhere(labels_test == 1)[:,0]
    idx_neg = np.argwhere(labels_test == 0)[:,0]

    tot_pos = (labels_test == 1).sum()
    tot_neg = (labels_test == 0).sum()
    fpr = (preds_test_cascade[idx_neg] == 1).sum() / num_tneg
    fnr = (preds_test_cascade[idx_pos] == 0).sum() / num_tpos
    # keep the misclassified negative samples which means all the positive predictions
    idx_keep = np.argwhere(preds_test_cascade == 1)[:,0]

    features_test = features_test[idx_keep]
    labels_test = labels_test[idx_keep]

    # cumulative fpr and fnr
    fnr_cdx = fnr*fnr_cdx
    fpr_cdx = fpr*fpr_cdx
    fpr_test_list.append(fpr_cdx)
    fnr_test_list.append(fnr_cdx)

```

```
# plot testing fpr and fnr
fig = plt.figure(figsize=(11,11))
plt.plot(num_casc,fpr_test_list, marker="o",label="FPR")
plt.plot(num_casc,fnr_test_list, marker="v",label="FNR")

plt.xticks(num_casc)
plt.legend()
plt.savefig(f'imgs/cascade_test.png', bbox_inches='tight', pad_inches=0)
plt.close()
```

Listing 1: Source code