ECE 661: Homework 11

Christina Eberhardt (eberharc@purdue.edu)

Part 1: Face Recognition

1.1: Theoretical Background and Implementation

We want to use the image vectors instead of the images themselves. Therefore, when loading the images, we can already flatten them to obtain their vectors. We normalize all vectors to unit length to enable better comparison between them.

Afterwards we use PCA and LDA to create a low-dimensional representation of the data. Details see Section 1.1.1 and 1.1.2.

The next step is to project all the training images into the k-dimensional subspace with the following formula:

$$\boldsymbol{y} = \boldsymbol{W}_k^T(\boldsymbol{x} - \boldsymbol{m})$$

We train a nearest neighbor (NN) classifier with this projected data and the given labels.

Then we project the test images into the subspace and assign the label of the nearest neighbor. We compare with the ground truth labels and calculate the accuracy with the following formula:

$$Accuracy = \frac{Number\ of\ correctly\ classified\ images}{Number\ of\ test\ images}$$

We can repeat this for multiple values of k and compare the results.
Pick $1 \leq k \leq 19$.

1.1.1: PCA

PCA requires the eigendecomposition of the covariance matrix of the image vectors $\boldsymbol{x}_i$, $1 \leq i \leq N$. For the given dataset N = 630. Each image vector is of length 49152(color) or 16384(grayscale).

$$\boldsymbol{Cw} = \lambda \boldsymbol{w}$$

We only keep the k eigenvectors that belong to the k largest eigenvalues.

We cannot calculate the eigenvectors of **C** directly without risking numerical instability due to its huge size of 49152x49152 (or 16384x16384).
Instead we can look at

$$\boldsymbol{X} = [\boldsymbol{x}_1 - \mathbf{m} \quad | \quad \boldsymbol{x}_2 - \boldsymbol{m} \quad | \quad \cdots \quad | \quad \boldsymbol{x}_N - \boldsymbol{m}]$$

Where **m** is the mean image of the normalized training images, making the vectors zero-mean.

We then have that

$$\boldsymbol{C} = \boldsymbol{X}\boldsymbol{X}^T$$

$$\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{w} = \boldsymbol{C}\boldsymbol{w} = \lambda \boldsymbol{w}$$

Now $\boldsymbol{X}\boldsymbol{X}^T$ is still of same dimensionality as **C**, so we do not profit from this change. When we look at

$$\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{u} = \lambda \boldsymbol{u}$$

the dimensionality is significantly lower with NxN = 630x630.

Then we can obtain the eigenvectors of **C** via the following correspondence:

$$\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{u} = (\boldsymbol{X}\boldsymbol{X}^T)\boldsymbol{X}\boldsymbol{u} = \boldsymbol{C}\boldsymbol{X}\boldsymbol{u} = \lambda \boldsymbol{X}\boldsymbol{u}$$

And we get

$$\boldsymbol{w} = \boldsymbol{X}\boldsymbol{u}$$

These eigenvectors are not of unit length and need to be normalized. We then keep the k eigenvectors that correspond to the largest eigenvalues.

1.1.2: LDA

For LDA we consider the between- class scatter S_B and the within- class scatter S_W. Let $\boldsymbol{m}_i$ be the class mean image, $|\mathcal{C}|$ the number of all classes and $|\mathcal{C}_i|$ the number of images in class i. Let x_k^i be the kth image vector of the ith class.

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

$$\boldsymbol{S}_W = \frac{1}{|\mathcal{C}|}\sum_{i=1}^{|\mathcal{C}|}\frac{1}{|\mathcal{C}_i|}\sum_{i=1}^{|\mathcal{C}_i|}\left(x_k^i - \boldsymbol{m}_i\right)\left(x_k^i - \boldsymbol{m}_i\right)^T$$

We want to find the **w** that maximize the Fisher Discriminant Function:

$$J(\boldsymbol{w}) = \frac{\boldsymbol{w}^T\boldsymbol{S}_B\boldsymbol{w}}{\boldsymbol{w}^T\boldsymbol{S}_W\boldsymbol{w}}$$

This is the case for

$$\boldsymbol{S}_B\boldsymbol{w} = \boldsymbol{\lambda}\boldsymbol{S}_W\boldsymbol{w}$$

If $\boldsymbol{S}_W$ is nonsingular we can solve

$$\boldsymbol{S}_W{}^{-1}\boldsymbol{S}_B\boldsymbol{w} = \boldsymbol{\lambda}$$

Usually this is not the case. Therefore, look at the Yu and Yang algorithm.

As a first step calculate the eigendecomposition of $\boldsymbol{S}_B$.

$$\boldsymbol{S}_B = \boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}^T$$

Discard of those eigenvectors corresponding to eigenvalues in $\boldsymbol{\Lambda}$ that are close to zero. Call the matrix of the k largest eigenvectors that we keep **Y**.

$$\boldsymbol{Y}^T\boldsymbol{S}_B\boldsymbol{Y} = \boldsymbol{D}_B$$

Where $\boldsymbol{D}_B$ is the upper left kxk sub-matrix of $\boldsymbol{\Lambda}$ (due to the descending order of the eigenvalues).

In the next step construct a matrix **Z** that unitizes $\boldsymbol{S}_B$.

$$\boldsymbol{Z} = \boldsymbol{Y}\boldsymbol{D}_B{}^{-1/2}$$

Use eigendecomposition to diagonalize $\boldsymbol{Z}^T\boldsymbol{S}_W\boldsymbol{Z}$**.** This yields a matrix **U** of eigenvectors s.t.

$$\boldsymbol{Z}^T\boldsymbol{S}_W\boldsymbol{Z} = \boldsymbol{U}\boldsymbol{D}_W\boldsymbol{U}^T$$

We discard of the largest eigenvalues of $\boldsymbol{S}_W$ and drop their eigenvectors.

Denote the matrix of eigenvectors that we keep by $\widehat{\boldsymbol{U}}$. We can then obtain the LDA eigenvectors that maximize the Fisher Discriminant Function by

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

We cannot calculate the two eigendecompositions directly as the matrices are too big. Hence, we use the same trick we used for PCA to solve the problem.

1.2: Observations

For colour images we observe that PCA reaches 100% accuracy for the first time at k=13 and LDA for k=7. For small k PCA performs significantly better than LDA. Starting at k=5 LDA performs better than PCA until PCA reaches 100% accuracy as well.

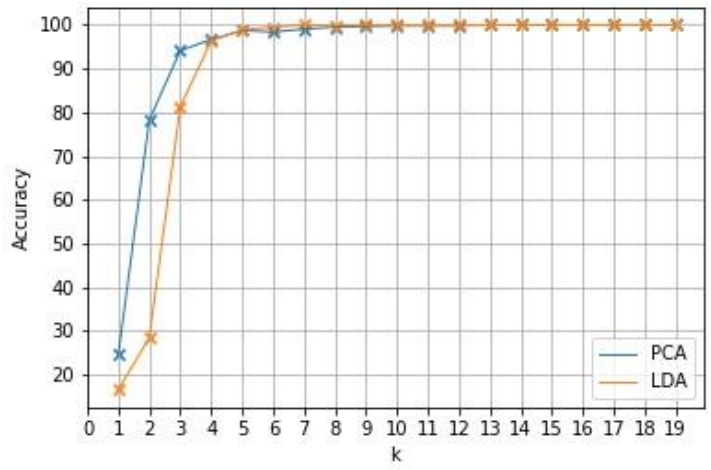

Figure 1: Accuracies for PCA and LDA for different values of k on colour images

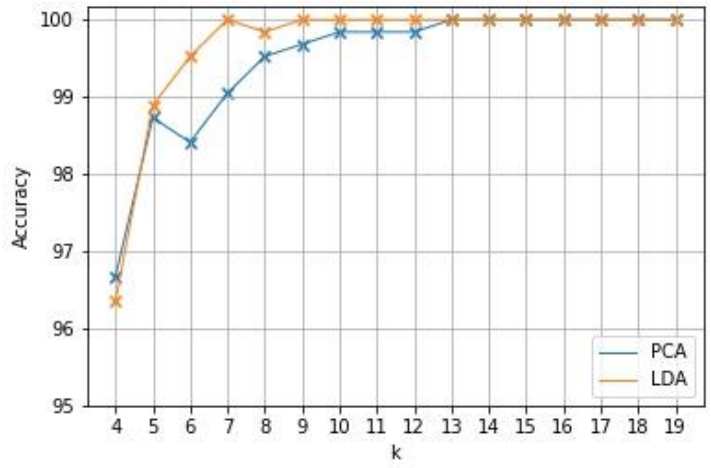

Figure 2: Accuracies for PCA and LDA for different values of k on colour images zoomed in on accuracy >95%

For grayscale images we observe that PCA reaches 100% accuracy for the first time at k=13 and LDA for k=7. For small k PCA performs significantly better than LDA. Starting at k=6 LDA performs better than PCA until PCA reaches 100% accuracy as well.

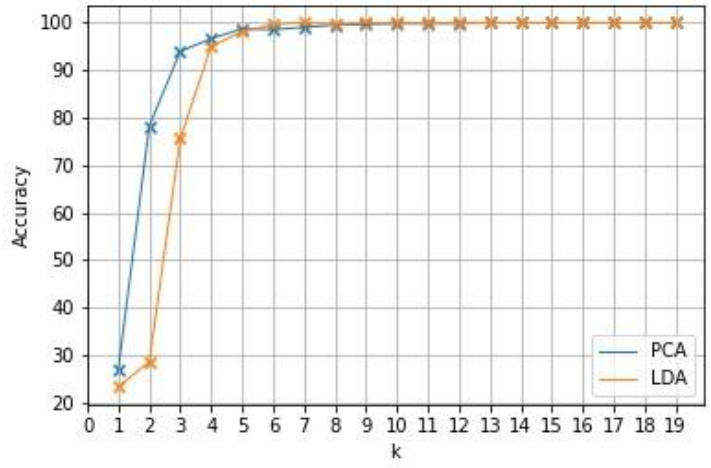

Figure 3:Accuracies for PCA and LDA for different values of k on grayscale images

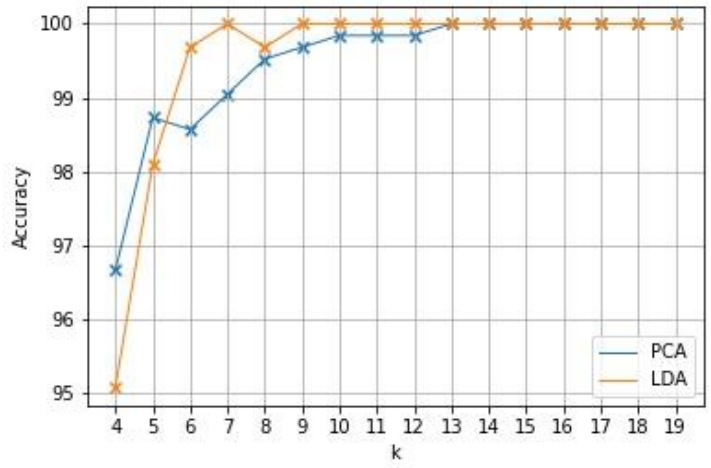

Figure 4: Accuracies for PCA and LDA for different values of k on grayscale images zoomed in on accuracy >95%

Overall, we can observe that PCA performs better on very small k but LDA reaches 100% accuracy for smaller k already than PCA. The overall difference in performance between colour and grayscale images is very small.

Part 2: Object Detection with Cascaded AdaBoost Classification

2.1: Theoretical Background and Implementation

We have labeled training data $(\boldsymbol{x}_i, y_i)$ where $1 \leq i \leq m$ and m is the number of training samples. The $\boldsymbol{x}_i$ are the training data and y_i the class labels. We have only two class labels "negative" and "positive". Represent them by 0 for "negative" and +1 for "positive". To improve the performance, we use a classifier cascade with S stages. For each stage we determine an AdaBoost classifier of its own.

Then for each stage we repeat the following:

For each iteration t of the algorithm in each stage we have a distribution $D_t(\boldsymbol{x}_i)$. The initial distribution of the first stage has a uniform probability distribution over the training samples. Denote it as $D_0(\boldsymbol{x}_i)$.

For each iteration of the AdaBoost algorithm we choose a weak classifier. Denote it as h_t. With this classifier we determine the predicted label $h_t(\boldsymbol{x}_i)$ and compare it to y_i. We repeat this for all m training elements and determine the classification error rate. Because of the large number of features this is done in a vectorized form. Let the misclassification rate for h_t be denoted by ϵ_t. Based on ϵ_t define a trust value α_t to describe how much we trust the classifier.

Use the set of weak classifiers and take their weighted sum with weights based on their α_t. Denote this final classifier as H.

For each iteration t, $1 \leq t \leq T$ of the AdaBoost algorithm we carry out 5 steps.

1. Choose a subset of the training samples based on $D_t(\boldsymbol{x}_i)$. Construct a weak classifier h_t. Make sure that h_t specifically targets training samples that were misclassified by h_{t-1}. We do so by using the weight **w** that is initialized the way that 50% of the weight are on the positive samples and 50% on the negative samples.
2. Apply h_t.to all training data.
3. Estimate $\epsilon_t = \frac{1}{2}\sum_{i=1}^{m} D_t(\boldsymbol{x}_i) \cdot |\, h_t(\boldsymbol{x}_i) - y_i|$.
4. Calculate the trust factor $\alpha_t = \frac{1}{2}\ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$
5. Update the probability distribution $D_{t+1}(\boldsymbol{x}_i) = \frac{D_t(\boldsymbol{x}_i)\exp\left(-\alpha_t y_i h_t(\boldsymbol{x}_i)\right)}{Z_t}$.
   Where $Z_t = \sum_{i=1}^{m} D_t(\boldsymbol{x}_i)\exp\left(-\alpha_t y_i h_t(\boldsymbol{x}_i)\right)$

After T iterations we construct H:

$$H(x) = sign\left(\sum_{t=1}^{T} \alpha_t \, h_t(x)\right)$$

We use this H as a feature once again and repeat for the outer loop.

2.1.1: Obtaining the features

We use 4 types of weak features and place their windows at every possible position in the image to create a large number of features. With all of the types and the image size of 40x20 we obtain 192270 features. We use Haar filters of different kinds. We sum up the values of all the regions of ones and subtract the sum of all the regions of zeros. For a rectangle ABCD we can calculate its value by obtaining the integral image and using the corner points of the rectangle the following way:

A B
D C

$$value(ABCD) = C - B - D + A$$

Type 1 features:

Use Haar filters that are symmetric along the y-axis. On the left we have only zeros, on the right only ones. Then we have the following structure:

A B C
0 1
F E D

Then we get the following formula for the feature:

$$D - 2E - C + 2B + F - A$$

Type 2 features:

Use Haar filters that are symmetric along the x-axis. On the top we have only zeros, on the bottom only ones. Then we have the following structure:

A B
0
F C
1
E D

Then we get the following formula for the feature:

$$D - E - 2C + B + 2F - A$$

Type 3 features:

This feature is an extension of the Type 1 feature. It consists of a block of zeros followed by a block of ones followed by a block of zeros horizontally. The three blocks are of equal size.

A		B		C		D
	0		1		0	
H		G		F		E

Then we get the following formula for the feature:

$$2F + 2B - 2C - 2G - A + H - E + D$$

Type 4 features:

This feature is an extension of the Type 1 and Type 2 features. It consists of a big square consisting of a block of ones and zeros horizontally and vertically each.

A		B		C
	0		1	
D		E		F
	1		0	
G		H		I

Then we get the following formula for the feature:

$$2H - 4E - G + 2D + 2F - C + 2B + A - I$$

2.1.2: A few implementation details of the Cascaded AdaBoost implementation

In the outer loop we create a strong classifier for each stage. In the inner loop we find the best weak feature. Use the vectorized approach to avoid a third nested loop.

We find the error $\epsilon = \min\left(S^+ + (T^- - S^-), S^- + (T^+ - S^+)\right)$ where T^+ is the total sum of the sample weights of the positive samples and S^+ is the sum of the weights of the positive samples that are below the current sample.

T^- is the total sum of the sample weights of the negative samples and S^+ is the sum of the weights of the negative samples that are below the current sample.

By minimizing this error we choose the best sample for thresholding. We determine the polarity based on if the first or second term is the better one.

We check the false positive rate of the classifier. If it is below 50% we are done. If not we continue for another iteration until the features together have an accuracy of at least 50%. We choose the threshold for feature detection to be the smallest value of the positive samples. Then all the samples of the positive class cause a true positive rate of 1.

Once we have a strong classifier, we change the sample space: we do not consider samples from the negative class that have been classified correctly anymore.

Next, we check the performance of all stages so far together. If there are no negative samples left (i.e. FPR = 0) we terminate the loop. Otherwise we go on to another stage. We can then use the obtained classifier to classify the testing images by evaluating the strong classifiers.

In the testing phase we use the condition

$$polarity \cdot f(x) < polarity \cdot \theta$$

On the weak classifiers. If this is true we assign the sample to class “positive”, otherwise to class “negative”. Then we take the result of the classifier and use the combination of all the strong classifier results with their respective alphas. We threshold the resulting value to obtain the final prediction.

2.2: Observations

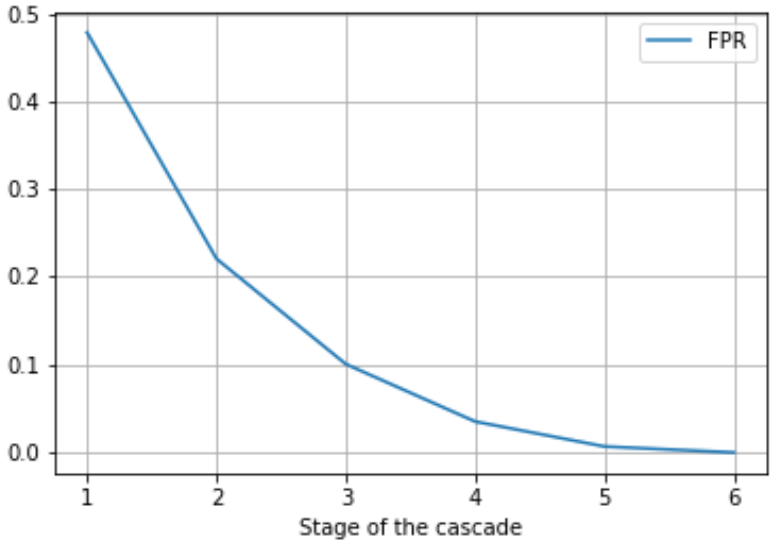

Figure 5: FPR during training for the different stages

Due to forcing the true positive rate (TPR) to be 1 constantly during training we can ignore it – there is no change. Further stages are added until all negative samples are classified correctly and hence the false positive rate FPR = 0.

6 stages are needed to reach this result. Each stage consists of multiple features. Namely the number of features per stage are the following:

Stage	Number of features
1	5
2	13
3	18
4	24
5	16
6	9

We can observe exponential decay in the FPR. This is because each stage has an individual FPR of at most 50%. Hence, we observe the training behaviour that we would expect. It is noticeable that those numbers are smaller than in prior semesters. This can be explained with the use of the more expressive type 3 and 4 features.

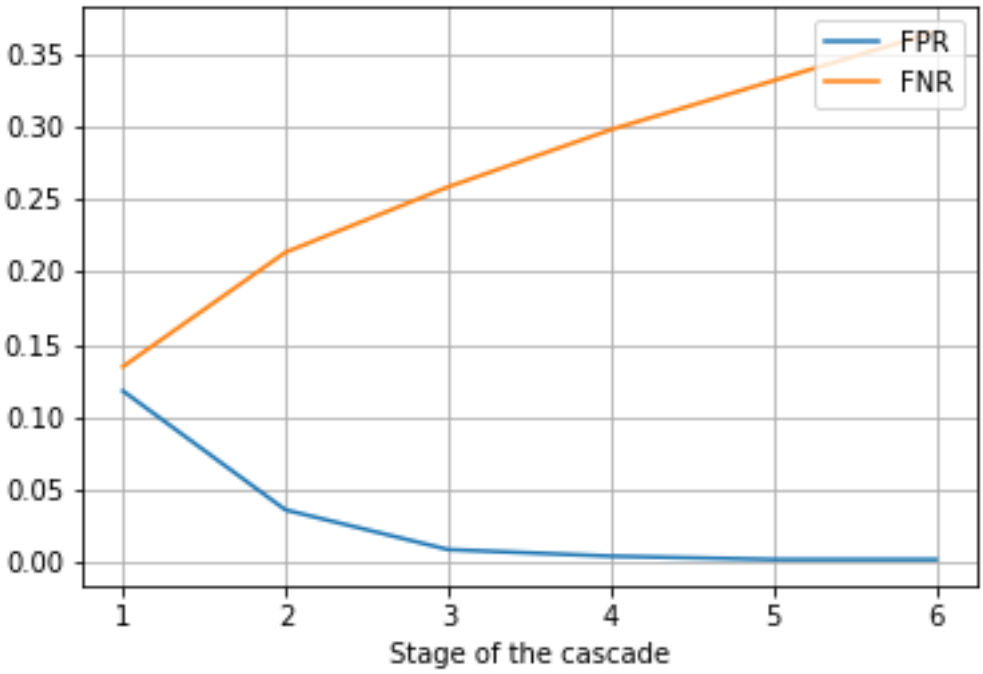

Figure 6:FPR and FNR during testing for the different stages

For the testing data we observe that the increased number of stages actually improves the performance concerning the FPR. We can also see, that this improvement comes at a cost: the false negative rate (FNR) increases and hence the true positive rate decreases. This means that some objects of the positive class get misclassified as negative. The final FPR is 0.00227 and final FNR 0.36517.

This behaviour can be explained with the multiplicative behaviour of the combined rates between the stages.

For example, $(0.5)^6 \approx 0.015625$ and $(0.95)^6 \approx 0.735$. This means that even if we observe the smallest errors concerning the TPR then those errors still get penalized harshly over multiple stages.

Part 3: Code

3.1: Code for Task 1

```
import numpy as np

import cv2

from sklearn.neighbors import KNeighborsClassifier

import matplotlib.pyplot as plt



trainpath = "./ECE661_2020_hw11_DB1/train/"

testpath = "./ECE661_2020_hw11_DB1/test/"


def get_images(train_or_test, grayscale):
    """
    This method takes in the string "train" or "test" and returns the list of normalized image vectors with the list of corresponding labels.
    If "train" is selected the mean normalized image vector is returned as well.
    """
    images = []
    labels = []
    class_means = []
    for label in range(1,31):
        if label < 10:
            label = "0%s"%(label)
        images_of_class = []
        for image_no in range(1,22):
            labels.append(label)
            if image_no < 10:
                image_no = "0%s"%(image_no)
            if grayscale == True:
                image = cv2.imread("./ECE661_2020_hw11_DB1/%s/%s_%s.png"%(train_or_test,label,image_no),cv2.IMREAD_GRAYSCALE)
            else:
                image = cv2.imread("./ECE661_2020_hw11_DB1/%s/%s_%s.png"%(train_or_test,label,image_no))
            image = np.ndarray.flatten(image)
            image_norm = image/np.linalg.norm(image)
            images.append(image_norm)
            images_of_class.append(image_norm)
        class_means.append(np.mean(images_of_class, axis = 0))
    if train_or_test == "train":
        return images, labels, class_means
```

```
    return images, labels


def subtract_mean_image(train_images, test_images):
    """
    Subtracts the mean image from the array of image vectors.
    """
    X = np.array(train_images).astype(np.float64)
    X_test = np.array(test_images).astype(np.float64)
    mean_image = np.mean(X, axis = 0)
    X = np.subtract(X,mean_image)
    X_test = np.subtract(X_test,mean_image)
    return X, X_test, mean_image


def get_PCA_eigenvectors(X, k):
    """
    Apply PCA to obtain the k best eigenvectors of the matrix X.
    """
    # Note: X is stored in transposed form to theory
    xtransx = np.dot(X,np.transpose(X))
    u,d,vt = np.linalg.svd(xtransx)
    v = np.transpose(vt)
    w = np.dot(np.transpose(X),v)
    w = w/np.linalg.norm(w, axis = 0)
    k_eigenvectors = w[:,:k]

    return k_eigenvectors


def get_acc_NN_for_given_w(w, X, X_test, train_labels, test_labels):
    """
    Project the training and testing data into the subspace formed by w. Returns the accuracy on the testing data with NN.
    """
    y = np.dot(np.transpose(w), np.transpose(X))
    y_test = np.dot(np.transpose(w), np.transpose(X_test))

    classifier = KNeighborsClassifier(n_neighbors=1)
    classifier.fit(np.transpose(y), train_labels)

    y_hat = classifier.predict(np.transpose(y_test))
    matching_labels = np.where(y_hat==test_labels)
```

```
    correct_label_count = len(matching_labels[0])
    acc = correct_label_count/len(test_images)
    return acc

def get_LDA_eigenvectors(X, k, number_of_classes = 30, number_of_images_per_class=21):
    """
    Apply LDA to obtain the k best eigenvectors of the matrix X.
    """
    mean_i_minus_mean = []
    x_minus_m_i = []
    for i in range(number_of_classes):
        mean_i_minus_mean.append(class_means[i]-mean_image)
        for j in range(number_of_images_per_class):
            x_minus_m_i.append(train_images[j+i*number_of_images_per_class]-class_means[i])

    mean_i_minus_mean = np.array(mean_i_minus_mean)
    u, d, vt = np.linalg.svd(np.dot(mean_i_minus_mean, np.transpose(mean_i_minus_mean)))
    Y = np.dot(np.transpose(mean_i_minus_mean),u)
    d = 1/np.sqrt(d)
    D = np.diag(d)
    Z = np.dot(Y,D)

    x_minus_m_i = np.array(x_minus_m_i)
    X = np.dot(np.transpose(Z),np.transpose(x_minus_m_i))
    u, d, vt = np.linalg.svd(np.dot(X, np.transpose(X)))

    w_transpose = np.dot(vt[np.shape(vt)[0]-k:],np.transpose(Z))
    w = np.transpose(w_transpose)
    return w

def draw_accuracy_plots(accuracies_PCA, accuracies_LDA, grayscale):
    """
    This method takes in the list of accuracies for the values of k and plots them into plots together.
    We obtain 2 plots:
        - one plot showing all accuracies
        - one plot zooming in to the area above 95% accuracy
    """
    accuracies_PCA_percent = [element*100 for element in accuracies_PCA]
    accuracies_LDA_percent = [element*100 for element in accuracies_LDA]
```

```
    plt.clf()
    plt.plot(range(1,20),accuracies_PCA_percent, label = "PCA", linewidth = 1)
    plt.plot(range(1,20),accuracies_LDA_percent, label = "LDA",linewidth = 1)
    plt.scatter(range(1,20),accuracies_PCA_percent, marker = "x")
    plt.scatter(range(1,20),accuracies_LDA_percent,marker ="x")
    plt.xticks(ticks=range(0,20))
    plt.yticks(ticks=range(20,101,10))
    plt.legend(loc="lower right")
    plt.grid()
    if grayscale == True:
        plt.savefig("Grayscale Accuracies of PCA and LDA.jpg")
    else:
        plt.savefig("Accuracies of PCA and LDA.jpg")
    #plt.show()


    plt.clf()
    plt.plot(range(4,20),accuracies_PCA_percent[3:], label = "PCA", linewidth = 1)
    plt.plot(range(4,20),accuracies_LDA_percent[3:], label = "LDA",linewidth = 1)
    plt.scatter(range(4,20),accuracies_PCA_percent[3:], marker = "x")
    plt.scatter(range(4,20),accuracies_LDA_percent[3:],marker ="x")
    plt.xticks(ticks=range(4,20))
    plt.yticks(ticks=range(95,101,1))
    plt.legend(loc="lower right")
    plt.grid()
    if grayscale == True:
        plt.savefig("Grayscale Accuracies of PCA and LDA detail.jpg")
    else:
        plt.savefig("Accuracies of PCA and LDA detail.jpg")



for grayscale in [True, False]:
    train_images, train_labels, class_means = get_images("train", grayscale)
    test_images, test_labels = get_images("test", grayscale)

    accuracies_PCA = []
    accuracies_LDA = []

    for k in range(1,20):
        X, X_test, mean_image = subtract_mean_image(train_images, test_images)
```

```
        w_PCA = get_PCA_eigenvectors(X,k)
        acc_PCA = get_acc_NN_for_given_w(w_PCA, X, X_test, train_labels, test_labels)
        accuracies_PCA.append(acc_PCA)

        w_LDA = get_LDA_eigenvectors(X,k)
        acc_LDA = get_acc_NN_for_given_w(w_LDA, X, X_test, train_labels, test_labels)
        accuracies_LDA.append(acc_LDA)

    draw_accuracy_plots(accuracies_PCA, accuracies_LDA, grayscale)
```

3.2: Code for Task 2

```
"""
Parts of this implementation are strongly based on the vectorized implementation of cascaded AdaBoost by Wan-Eih Huang.
"""



import numpy as np
import cv2
import os
from skimage.transform.integral import integral_image
import matplotlib.pyplot as plt
import pickle



def get_images(train_or_test, positive_or_negative):
    """
    This method takes in the string "train" or "test" and returns the list of images in grayscale with the list of corresponding labels.

    Due to the cars being able to having any color we know that color is not a valuable feature in this application.
    Each image is of size 40x20 (width x height)
    """
    path = "./ECE661_2020_hw11_DB2/%s/%s/"%(train_or_test, positive_or_negative)
    required_images = os.listdir(path)
    images = []
    for extension in required_images:
```

```
        full_path = os.path.join(path,extension)
        image = cv2.imread(full_path, cv2.IMREAD_GRAYSCALE )
        images.append(image)
    return images


def get_type1_kernels(img_width, img_height):
    """
    Returns all kernels of type 1.

    All images are of same shape, therefore it is sufficient to determine the general shape of all kernels once. The maximum size of kernel we can have is the image size.
    """
    list_of_kernels_type1 = []
    for w in range(1, int(img_width/2)+1):
        for h in range(1, int(img_height)+1):
            zeros = np.zeros((h,w))
            ones = np.ones((h,w))
            kernel = np.append(zeros, ones, axis = 1)
            list_of_kernels_type1.append(kernel)
    return list_of_kernels_type1


def get_type2_kernels(img_width, img_height):
    """
    Returns all kernels of type 2.

    Type 2 kernels are the transpose variant of Type 1 kernels.
    All images are of same shape, therefore it is sufficient to determine the general shape of all kernels once. The maximum size of kernel we can have is the image size.
    """
    list_of_kernels_type2 = []
    for h in range(1, int(img_height/2)+1):
        for w in range(1, int(img_width)+1):
            zeros = np.zeros((w,h))
            ones = np.ones((w,h))
            kernel = np.append(zeros, ones, axis = 1)
            kernel2 = np.transpose(kernel)
            list_of_kernels_type2.append(kernel2)
    return list_of_kernels_type2
```

```
def get_type3_kernels(img_width, img_height):
    """
    Returns all kernels of type 3.

    The kernels of type 3 are a horizontal block of zeros followed by a horizontal block of ones followed by a block of zeros.
    All images are of same shape, therefore it is sufficient to determine the general shape of all kernels once. The maximum size of kernel we can have is the image size.
    """
    list_of_kernels_type1 = []
    for w in range(1, int(img_width/3)+1):
        for h in range(1, int(img_height)+1):
            zeros = np.zeros((h,w))
            ones = np.ones((h,w))
            kernel = np.append(np.append(zeros, ones, axis = 1), zeros, axis=1)
            list_of_kernels_type1.append(kernel)
    return list_of_kernels_type1

def get_type4_kernels(img_width, img_height):
    """
    Returns all kernels of type 4.

    They consist of 4 squares of ones and zeros respectively.
    All images are of same shape, therefore it is sufficient to determine the general shape of all kernels once. The maximum size of kernel we can have is the image size.
    """
    list_of_kernels_type4 = []
    s = min(img_width, img_height)
    for s in range(1, int(s/2)+1):
        zeros = np.zeros((s,s))
        ones = np.ones((s,s))
        kernelupper = np.append(zeros, ones, axis = 1)
        kernellower = np.append(ones, zeros, axis = 1)
        kernel = np.append(kernelupper, kernellower, axis = 0)
        list_of_kernels_type4.append(kernel)
    return list_of_kernels_type4

def get_integral_image(image):
    """
    Returns the integral image of an input image.
```

```
    """
    integral_img = integral_image(image)
    return integral_img

def get_all_integral_images():
    """
    return the integral images for all images in their respective lists.
    """
    images_train_pos = get_images("train", "positive")
    images_train_neg = get_images("train", "negative")
    images_test_pos = get_images("test", "positive")
    images_test_neg = get_images("test", "negative")
    integral_images_train_pos = []
    integral_images_train_neg = []
    integral_images_test_pos = []
    integral_images_test_neg = []

    for image in images_train_pos:
        integral_images_train_pos.append(get_integral_image(image))

    for image in images_train_neg:
        integral_images_train_neg.append(get_integral_image(image))

    for image in images_test_pos:
        integral_images_test_pos.append(get_integral_image(image))

    for image in images_test_neg:
        integral_images_test_neg.append(get_integral_image(image))
    return integral_images_train_pos, integral_images_train_neg, integral_images_test_pos, integral_images_test_neg

def evaluate_Haar_filters_type1(integral_img, list_of_kernels_type1, height, width):
    """
    Create the type1 features at every possible position in the image.
    """
    feature_list = []
    for kernel in list_of_kernels_type1:
        kernel_height, kernel_width = np.shape(kernel)
        for y in range(height - kernel_height):
            for x in range(width - kernel_width):
```

```
            coord_x_0 = x
            coord_x_1 = x + int(kernel_width/2)
            coord_x_2 = x + kernel_width

            coord_y_0 = y
            coord_y_1 = y + kernel_height

            A = integral_img[coord_y_0, coord_x_0]
            B = integral_img[coord_y_0, coord_x_1]
            C = integral_img[coord_y_0, coord_x_2]
            D = integral_img[coord_y_1, coord_x_2]
            E = integral_img[coord_y_1, coord_x_1]
            F = integral_img[coord_y_1, coord_x_0]

            value = D-2*E-C+2*B+F-A
            feature_list.append(value)
    return feature_list

def evaluate_Haar_filters_type2(integral_img, list_of_kernels_type2, height, width):
    """
    Create the type2 features at every possible position in the image.
    """
    feature_list = []
    for kernel in list_of_kernels_type2:
        kernel_height, kernel_width = np.shape(kernel)
        for y in range(height - kernel_height):
            for x in range(width - kernel_width):
                coord_y_0 = y
                coord_y_1 = y + int(kernel_height/2)
                coord_y_2 = y + kernel_height

                coord_x_0 = x
                coord_x_1 = x + kernel_width

                A = integral_img[coord_y_0, coord_x_0]
                B = integral_img[coord_y_0, coord_x_1]
                C = integral_img[coord_y_1, coord_x_1]
                D = integral_img[coord_y_2, coord_x_1]
                E = integral_img[coord_y_2, coord_x_0]
```

```
            F = integral_img[coord_y_1, coord_x_0]

            value = D-E-2*C+2*F-A +B
            feature_list.append(value)
    return feature_list

def evaluate_Haar_filters_type3(integral_img, list_of_kernels_type3, height, width):
    """
    Create the type3 features at every possible position in the image.
    """
    feature_list = []
    for kernel in list_of_kernels_type3:
        kernel_height, kernel_width = np.shape(kernel)
        for y in range(height - kernel_height):
            for x in range(width - kernel_width):
                coord_x_0 = x
                coord_x_1 = x + int(kernel_width/3)
                coord_x_2 = x + int(2*kernel_width/3)
                coord_x_3 = x + kernel_width

                coord_y_0 = y
                coord_y_1 = y + kernel_height

                A = integral_img[coord_y_0, coord_x_0]
                B = integral_img[coord_y_0, coord_x_1]
                C = integral_img[coord_y_0, coord_x_2]
                D = integral_img[coord_y_0, coord_x_3]
                E = integral_img[coord_y_1, coord_x_3]
                F = integral_img[coord_y_1, coord_x_2]
                G = integral_img[coord_y_1, coord_x_1]
                H = integral_img[coord_y_1, coord_x_0]

                value = 2*F + 2*B -2*C -2*G -A + H - E + D
                feature_list.append(value)
    return feature_list

def evaluate_Haar_filters_type4(integral_img, list_of_kernels_type4, height, width):
    """
    Create the type4 features at every possible position in the image.
```

```
    """
    feature_list = []
    for kernel in list_of_kernels_type4:
        kernel_height, kernel_width = np.shape(kernel)
        for y in range(height - kernel_height):
            for x in range(width - kernel_width):
                coord_x_0 = x
                coord_x_1 = x + int(kernel_width/2)
                coord_x_2 = x + kernel_width

                coord_y_0 = y
                coord_y_1 = y + int(kernel_height/2)
                coord_y_2 = y + kernel_height

                A = integral_img[coord_y_0, coord_x_0]
                B = integral_img[coord_y_0, coord_x_1]
                C = integral_img[coord_y_0, coord_x_2]
                D = integral_img[coord_y_1, coord_x_0]
                E = integral_img[coord_y_1, coord_x_1]
                F = integral_img[coord_y_1, coord_x_2]
                G = integral_img[coord_y_2, coord_x_0]
                H = integral_img[coord_y_2, coord_x_1]
                I = integral_img[coord_y_2, coord_x_2]

                value = 2*H -4*E-G+2*D+2*F-C+2*B+A-I
                feature_list.append(value)
    return feature_list


def get_features_for_all_images(kernel_list, list_of_kernels_type1, list_of_kernels_type2, list_of_kernels_type3, list_of_kernels_type4,
height, width):
    """
    Returns the features for all images of a certain image list.
    """
    list_features_per_image = []
    for integral_img in kernel_list:
        feature_list_type1 = evaluate_Haar_filters_type1(integral_img, list_of_kernels_type1,  height, width)
        feature_list_type2 = evaluate_Haar_filters_type2(integral_img, list_of_kernels_type2,  height, width)
        feature_list_type3 = evaluate_Haar_filters_type3(integral_img, list_of_kernels_type3,  height, width)
        feature_list_type4 = evaluate_Haar_filters_type4(integral_img, list_of_kernels_type4,  height, width)
```

```
        features = feature_list_type1.copy()
        features.extend(feature_list_type2)
        features.extend(feature_list_type3)
        features.extend(feature_list_type4)
        list_features_per_image.append(features)
    return list_features_per_image


def get_features():
    """
    Returns all the features for training and testing sorted after positive and negative class.
    """
    integral_images_train_pos, integral_images_train_neg, integral_images_test_pos, integral_images_test_neg = get_all_integral_images()
    height, width = np.shape(integral_images_train_pos[0])


    list_of_kernels_type1 = get_type1_kernels(width,height)
    list_of_kernels_type2 = get_type2_kernels(width,height)
    list_of_kernels_type3 = get_type3_kernels(width,height)
    list_of_kernels_type4 = get_type4_kernels(width,height)


    features_train_pos = get_features_for_all_images(integral_images_train_pos, list_of_kernels_type1, list_of_kernels_type2,
list_of_kernels_type3, list_of_kernels_type4, height, width)
    features_train_neg = get_features_for_all_images(integral_images_train_neg, list_of_kernels_type1, list_of_kernels_type2,
list_of_kernels_type3, list_of_kernels_type4, height, width)
    features_test_pos = get_features_for_all_images(integral_images_test_pos, list_of_kernels_type1, list_of_kernels_type2,
list_of_kernels_type3, list_of_kernels_type4, height, width)
    features_test_neg = get_features_for_all_images(integral_images_test_neg, list_of_kernels_type1, list_of_kernels_type2,
list_of_kernels_type3, list_of_kernels_type4, height, width)


    return features_train_pos, features_train_neg, features_test_pos, features_test_neg


def save_features_to_file(features_train_pos, features_train_neg, features_test_pos, features_test_neg):
    """
    Code to save the features to file because calculating them takes a long time.
    """
    filename = "features_train_pos.npz"
    np.savez(filename, features_train_pos)


    filename = "features_train_neg.npz"
    np.savez(filename, features_train_neg)
```

```
    filename = "features_test_pos.npz"
    np.savez(filename, features_test_pos)

    filename = "features_test_neg.npz"
    np.savez(filename, features_test_neg)


def get_features_from_file(train_or_test):
    """
    Load the features from the file.

    train_or_test has two possible values:
        - "train": Training data is returned
        - "test" : Testing data is returned
    """
    if train_or_test == "train":
        filename = "features_train_pos.npz"
        file = np.load(filename)
        features_train_pos = file["arr_0"]

        filename = "features_train_neg.npz"
        file = np.load(filename)
        features_train_neg = file["arr_0"]

        return features_train_pos, features_train_neg

    elif train_or_test == "test":
        filename = "features_test_pos.npz"
        file = np.load(filename)
        features_test_pos = file["arr_0"]

        filename = "features_test_neg.npz"
        file = np.load(filename)
        features_test_neg = file["arr_0"]

        return features_test_pos, features_test_neg
```

```
def cascade(features, number_of_pos_samples, number_of_neg_samples, FPR_stage_max, T, cascade_data):
    """
    This function combines up to T weak classifiers.

    If the classifiers combine to a joined classifier with false positive rate below FPR_stage_max.
    """
    weights_for_pos_samples = np.ones(number_of_pos_samples)*0.5/number_of_pos_samples
    weights_for_neg_samples = np.ones(number_of_neg_samples)*0.5/number_of_neg_samples
    weights = np.append(weights_for_pos_samples, weights_for_neg_samples)

    labels_for_pos_samples = np.ones(number_of_pos_samples)
    labels_for_neg_samples = np.zeros(number_of_neg_samples)
    labels = np.append(labels_for_pos_samples, labels_for_neg_samples)
    labels = labels.astype(np.int)

    alphas = []
    list_of_ht = []
    list_of_h = []

    for t in range(T):
        print("t = %s"%(t))
        weights, alphas, list_of_h, list_of_ht = find_best_classifier(features, labels, weights, number_of_pos_samples, alphas, list_of_h,
list_of_ht)
        s = np.dot(np.asarray(list_of_h).transpose(), np.asarray(alphas))
        # smallest value of class positive --> force TPR = 100%
        internal_threshold = np.min(s[:number_of_pos_samples])

        s_pred = np.zeros(s.shape)
        s_pred[s>=internal_threshold] = 1

        fp = np.sum(s_pred[number_of_pos_samples:])/number_of_neg_samples
        tp = np.sum(s_pred[:number_of_pos_samples])/number_of_pos_samples

        print("FP: %s" %fp)
        print("TP: %s" %tp)

        if fp < FPR_stage_max:
            break
```

```
    number_of_misclassified = np.sum(s_pred[number_of_pos_samples:]).astype(int)

    updated_features = features[:number_of_pos_samples]
    for i in range(number_of_neg_samples):
        neg_id = i + number_of_pos_samples
        if s_pred[neg_id] == 1:
            neg_sample = features[neg_id]
            neg_sample = np.reshape(neg_sample, (1,-1))
            updated_features = np.append(updated_features,neg_sample, axis = 0)

    number_of_weak_classifiers = t
    cascade_classifier = [list_of_ht, updated_features, number_of_misclassified, number_of_weak_classifiers, alphas]
    return cascade_classifier



def find_best_classifier(features, labels, weights, number_of_pos_samples, alphas, list_of_h, list_of_ht):
    """
    This function finds the best weak classifier for certain features and weights.

    This code is vectorized to be able to handle the more than 192K features in a reasonable time. Therefore, it is strongly based on Wan-Eih Huang's code.
    """
    weights = weights/np.sum(weights)

    weights_unfolded = np.tile(weights, (np.shape(features)[1],1))
    labels_unfolded = np.tile(labels, (np.shape(features)[1],1))
    idx = np.argsort(features, axis = 0)
    row = np.arange(np.shape(features)[1])

    sorted_weights = weights_unfolded[row,idx]
    sorted_labels = labels_unfolded[row,idx]

    pos_T = np.sum(weights[:number_of_pos_samples])
    neg_T = np.sum(weights[number_of_pos_samples:])
    positive_weight_indicator = np.multiply(sorted_weights,sorted_labels)

    pos_S = np.cumsum(positive_weight_indicator, axis = 0)
    neg_S = np.cumsum(sorted_weights, axis = 0) - pos_S
```

```
err1 = pos_S + neg_T - neg_S
err2 = neg_S + pos_T - pos_S
error = np.stack((err1,err2), axis = 2)

min_idx = np.unravel_index(np.argmin(error), error.shape)
min_error = error[min_idx]

ft = min_idx[1]
index = idx[:,ft]
print("Feature-Index: %s" %ft)

pred_tmp = np.zeros(features.shape[0])
pred = np.zeros(features.shape[0])

if min_idx[2] == 0:
    polarity = -1
    pred_tmp[min_idx[0]+1:]=1
else:
    polarity = 1
    pred_tmp[:min_idx[0]+1]=1
pred[index] = pred_tmp

feature = features[:,ft]
feature = feature[index]

if min_idx[0] == 0:
    theta = feature[0] - 0.01
elif min_idx[0] == -1:
    theta = feature[-1] + 0.01
else:
    theta = np.mean(feature[min_idx[0]-1:min_idx[0]+1])

print("Theta = %s"%(theta))
beta = min_error/(1-min_error)

alphas.append(np.log(1/beta))
list_of_h.append(np.transpose(pred))
list_of_ht.append([ft, polarity, theta])
```

```
    weights = weights*(beta**(1-np.abs(labels-pred)))

    return weights, alphas, list_of_h, list_of_ht



#-----------------------------------------------------------------------------------------------------
# Create features and save them to file
#features_train_pos, features_train_neg, features_test_pos, features_test_neg = get_features()
#save_features_to_file(features_train_pos, features_train_neg, features_test_pos, features_test_neg)



#-----------------------------------------------------------------------------------------------------
# Load important variables for training
#features_train_pos, features_train_neg = get_features_from_file("train")
#
##-----------------------------------------------------------------------------------------------------
## Code for training
#number_of_pos_samples = np.shape(features_train_pos)[0]
#number_of_neg_samples = np.shape(features_train_neg)[0]
#original_number_of_neg_samples = number_of_neg_samples
#number_of_samples = number_of_neg_samples + number_of_pos_samples
#number_of_features = np.shape(features_train_pos)[1]
#
#features = np.concatenate((features_train_pos,features_train_neg),axis = 0)
#
## Define crucial values for Adaboost, they have been chosen based on last year's results
#T = 50 # max number of weak classifiers in a stage
#S = 10 # max number of stages
#
## Define the minimum performance of each stage
#FPR_stage_max = 0.5 # The smaller FPR per stage the less stages we need to obtain a small overall FPR
#
#FPR_of_stages = []
#TPR_of_stages = []
#FNR_of_stages = []
#actual_number_of_stages = []
#
```

```
#classifier = []
#features_per_stage = []
#misclassified_per_stage = []
#for stage in range(S):
#    cascade_classifier = cascade(features, number_of_pos_samples, number_of_neg_samples, FPR_stage_max, T, classifier)
#    classifier.append(cascade_classifier)
#
#    number_of_neg_samples = cascade_classifier[2]
#
#    features = cascade_classifier[1]
#    features_per_stage.append(cascade_classifier[3])
#    misclassified_per_stage.append(number_of_neg_samples)
#
#    fpr = number_of_neg_samples/original_number_of_neg_samples
#    FPR_of_stages.append(fpr)
#    if number_of_neg_samples == 0:
#        break
#
## Save the results
#open_file = open("classifier_info.csv", "wb")
#pickle.dump(classifier, open_file)
#open_file.close()
#
#open_file = open("FRP_train.csv", "wb")
#pickle.dump(FPR_of_stages, open_file)
#open_file.close()
#
#open_file = open("features_per_stage.csv", "wb")
#pickle.dump( features_per_stage, open_file)
#open_file.close()
#
## Plot the FPR
#actual_number_of_stages = list(range(stage+1))
#actual_number_of_stages = [x + 1 for x in actual_number_of_stages]
#fig = plt.figure(1)
#fig.clf()
#plt.plot(actual_number_of_stages, FPR_of_stages, label = "FPR")
#plt.grid()
#plt.xticks(ticks = actual_number_of_stages)
```

```
#plt.legend(loc = "upper right")
#plt.xlabel("Stage of the cascade")
#fig.savefig("Plot of FPR train")
#plt.show()



##-------------------------------------------------------------------------------------------------------
### Load important variables for testing
features_test_pos, features_test_neg = get_features_from_file("test")

open_file = open("classifier_info.csv", "rb")
classifier = pickle.load(open_file)
open_file.close()

open_file = open("FRP_train.csv", "rb")
FPR_of_stages = pickle.load(open_file)
open_file.close()

open_file = open("features_per_stage.csv", "rb")
features_per_stage = pickle.load(open_file)
open_file.close()
#
##-------------------------------------------------------------------------------------------------------
## Code for testing
number_of_pos_samples = np.shape(features_test_pos)[0]
number_of_neg_samples = np.shape(features_test_neg)[0]

original_number_of_pos_samples = number_of_pos_samples
original_number_of_neg_samples = number_of_neg_samples

S = len(classifier)

features = np.concatenate((features_test_pos,features_test_neg),axis = 0)

FPR_of_stages = []
FNR_of_stages = []

for stage in range(S):
    print("stage = %s" %(stage))
```

```
list_of_ht = classifier[stage][0]
ft = [sublist[0].astype(int) for sublist in list_of_ht]
polarity = np.array([sublist[1] for sublist in list_of_ht])
thetas = np.array([sublist[2].astype(int) for sublist in list_of_ht])
alphas = classifier[stage][4]

stage_threshold = 0.5*np.sum(alphas)

relevant_features = features[:,ft]

temp1 = polarity*thetas
temp1 = np.tile(temp1, (number_of_pos_samples+number_of_neg_samples,1))
temp2 = polarity*relevant_features

decision_matrix = np.subtract(temp1,temp2)

inner_prediction = np.zeros(np.shape(decision_matrix))
inner_prediction[decision_matrix >= 0] = 1

temp3 = np.dot(inner_prediction, alphas)

outer_prediction = np.zeros(number_of_pos_samples+number_of_neg_samples)
outer_prediction[temp3 >= stage_threshold] = 1

fp = np.sum(outer_prediction[number_of_pos_samples:])/original_number_of_neg_samples
tp = np.sum(outer_prediction[:number_of_pos_samples])/original_number_of_pos_samples
fn = 1 - tp

FPR_of_stages.append(fp)
FNR_of_stages.append(fn)

only_pos_array = outer_prediction[:number_of_pos_samples]
pos_true_indices = np.where(only_pos_array ==1)
updated_features = features[pos_true_indices]
for i in range(number_of_neg_samples):
    neg_id = i + number_of_pos_samples
    if outer_prediction[neg_id] == 1:
        neg_sample = features[neg_id]
```

```
            neg_sample = np.reshape(neg_sample, (1,-1))
            updated_features = np.append(updated_features,neg_sample, axis = 0)
    temp = np.sum(outer_prediction[:number_of_pos_samples]).astype(int)
    number_of_neg_samples = np.sum(outer_prediction[number_of_pos_samples:]).astype(int)
    number_of_pos_samples = temp


    features = updated_features


# Plot results
actual_number_of_stages = list(range(stage+1))
actual_number_of_stages = [x + 1 for x in actual_number_of_stages]
fig = plt.figure(2)
fig.clf()
plt.plot(actual_number_of_stages, FPR_of_stages, label = "FPR")
plt.plot(actual_number_of_stages, FNR_of_stages, label = "FNR")
plt.grid()
plt.xticks(ticks = actual_number_of_stages)
plt.legend(loc = "upper right")
plt.xlabel("Stage of the cascade")
fig.savefig("Plot of FPR and FNR test")
plt.show()
```