ECE 661 Fall 2020 - Homework 11

Brian Helfrecht

bhelfre@purdue.edu

1 Theory

As discussed several times in past assignments, image classification is an important topic in computer vision. Typically, small image sizes can be analyzed with relative ease. However, in the modern area, a single image can contain tens of millions of pixels—analysis of which can increase processing times exponentially. Additionally, high-dimensional feature vectors can lead to issues during classification. To combat this, the input images can be represented in a lower dimensional space that contains far fewer data points than the number of pixels in the image while still retaining the image's unique information. This dimensionality reduction can be accomplished in several ways, but two—principal component analysis and linear discriminant analysis—are explored in this assignment. The target application was the classification of several faces.

Furthermore, algorithms for classifying objects in an image are a large focus in computer vision. The second half of this assignment focused on implementing the cascaded AdaBoost classifier using the Viola and Jones algorithm, which aims to achieve an extremely low false positive rate. For this task, the false positive and negative rates for images containing cars were analyzed as a function of the number of cascade stages in the classifier.

1.1 Principal Component Analysis (PCA)

The first method to reduce the dimensionality of an image is through principal component analysis (PCA). The general idea of this method is to find an orthogonal set of direction vectors upon which to project each input image vector. This can be accomplished by retaining only the largest, most influential eigenvectors of a matrix containing image features. These eigenvectors, which are often far fewer in number than the number of features in the initial set, can then be used for testing and training. The general process for obtaining feature vectors using PCA is as follows:

1. Vectorize each image. That is, convert it from an $m \times n$ array to an $mn \times 1$ vector.
2. Normalize each vectorized image x_i by dividing by its length:
$$\hat{x_i} = \frac{x_i}{||x_i||}$$
3. Compute the global mean vector $\vec{m}$ of all N images in the data set:
$$\vec{m} = \frac{1}{N}\sum_{i=1}^{N} \hat{x_i}$$
4. Subtract the global mean $\vec{m}$ from each normalized image vector $\hat{x_i}$ and concatenate all the results into the columns of a matrix X.
5. Compute the eigenvalues and eigenvectors $\Lambda_K = [\tilde{\lambda_1}|\tilde{\lambda_2}|...|\tilde{\lambda_K}]$ of $X^T X$, which is an orthogonal matrix of PCA feature vectors.

6. Sort the eigenvectors in Λ_K by ordering their corresponding eigenvalues from largest to smallest. This process will enable dimensionality reduction by taking only the K largest, most influential eigenvectors from the original vector matrix.

7. Compute the final eigenvectors λ_i of the inital image matrix X using:

$$\lambda_i = X\tilde{\lambda_i}$$

8. Normalize the final eigenvectors:

$$\hat{\lambda_i} = \frac{\lambda_i}{||\lambda_i||}$$

9. Finally, retain only the P largest eigenvectors as columns in a matrix W_P, where P is the desired dimensionality of the classifier.

10. Then, feature vectors y_i for testing and training images can be calculated as follows:

$$y_i = W_P^T(\hat{x_i} - \vec{m})$$

The training and testing feature vectors can then be used for classification. For this assignment, the Nearest-Neighbor classifier was implemented, which uses the Euclidean distance between a test vector and all training vectors to find the closest match. The class of the nearest training vector becomes the predicted class for the test vector. In this assignment, the number of nearest neighbors used for each prediction was varied between 1 and 10. However, it was found that a single nearest-neighbor classifier worked the best. This single NN classifier was used for testing of both PCA and LDA, the latter of which is described below.

1.2 Linear Discriminant Analysis (LDA)

Linear discriminant analysis (LDA) is a method similar to LDA for dimensionality reduction. However, it seeks to find P maximally discriminating direction vectors between image classes. This is possible by maximizing the ratio between within-class and between-class scatter of the feature vectors. This can be expressed mathematically by finding the eigenvectors that maximize the Fisher Discriminant function:

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

This formula can be used directly provided S_W is not singular. However, this is often not the case. As such, a modified method must be used. As with PCA, once the feature vectors have been computed, test image classes can be predicted with a NN classifier or similar. A general outline of LDA for computing feature vectors when S_W is singular is described below:

1. As with PCA, vectorize each image. That is, convert it from an $m \times n$ array to an $mn \times 1$ vector.

2. Normalize each vectorized image x_i by dividing by its length:

$$\hat{x_i} = \frac{x_i}{||x_i||}$$

3. Compute the global mean vector $\vec{m_G}$ of all N images in the data set:

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

4. Compute the class mean vector for each class using the C images in the class:

$$\vec{m_C} = \frac{1}{C}\sum_{i=1}^{C} \hat{x_i}$$

5. Compute the mean matrix M by subtracting the global mean vector from each class mean vector. The resulting vectors should appear in the columns of M:

$$M = \vec{m_C} - \vec{m_G} \text{ for each } \vec{m_C}$$

6. Compute the eigenvalues and eigenvectors of $M^T M$ through eigen decomposition, and sort the eigenvectors from largest to smallest based on their corresponding eigenvalues.

7. Compute the final eigenvectors λ_i of M using:

$$\lambda_i = M\tilde{\lambda_i}$$

8. Normalize the final eigenvectors:

$$\hat{\lambda_i} = \frac{\lambda_i}{||\lambda_i||}$$

9. Create a new matrix $\Lambda_K = [\tilde{\lambda_1}|\tilde{\lambda_2}|...|\tilde{\lambda_K}]$ containing only eigenvectors corresponding to the K eigenvalues that are not nearly zero (vectors with eigenvalues greater than 1E-6 will suffice). This is necessary because we will be computing an inverse with the eigenvalues shortly, and we cannot do this with a singular matrix.

10. Create a diagonal matrix D_B containing the K non-zero eigenvalues along the main diagonal.

11. Determine the product $Z = \Lambda D_B^{-0.5}$.

12. Compute the matrix X of within-class difference vectors with $\vec{X_i} = \hat{x_i} - \vec{m_C}$, where i represents the column of X. In other words, for each normalized image vector, subtract its corresponding class mean, and assemble the results into the columns of the matrix X.

13. Compute the eigenvalues and initial eigenvectors $\tilde{v_i}$ of the product $(Z^T X)(Z^T X)^T$. Sort the eigenvectors from largest to smallest based on the eigenvalues.

14. Compute the final eigenvectors using $v_i = Z\tilde{v_i}$ and normalize them.

15. Take the P eigenvectors with largest eigenvalues as the columns of a matrix W_P.

16. Finally, a feature vector y_i can be computed using the P eigevectors using $y_i = W_P^T(\hat{x_i} - \vec{m})$ where $\vec{m}$ is the global mean of all image vectors in the training set and $\hat{x_i}$ is the image vector for which we wish to find the corresponding feature vector.

As with PCA, when computing the feature vectors for images in the test set, the global mean of all images in the training set is used. Then, a NN classifier can predict the class of an unknown test image vector.

1.3 The cascaded AdaBoost classifier

For the second task in this assignment, a cascaded AdaBoost classifier employing the Viola and Jones algorithm was implemented for object detection. This algorithm prioritizes a low false-positive rate. At a high level, the algorithm works by creating several stages, or cascades, of weak classifiers that, when linked together, form a very strong classifier. Each weak classifier need not be extremely robust—in fact, so long as it classifies at least half of the images properly, it can be used. If not, a "polarity" parameter for the classifier can be inverted such that the classifier does correctly classify at least half the images. In total, several simple weak classifiers applied in succession can construct a highly accurate overall classifier. Details on the implementation of such a classification algorithm are enumerated below.

1.3.1 Feature extraction

The weak classifiers making up the cascaded AdaBoost classifier work most effectively on a large feature set. That is, classification will work best if each image can decomposed into a set of thousands, or even tens of thousands of features. To extract these features, we can use Haar filters of differing sizes and orientations. For simplicity, we can use n-dimensional Haar vectors oriented either horizontally or vertically. The horizontal filters will be of size $1 \times 2, 1 \times 4, ..., 1 \times N$ where N is the width of the image in pixels. Similarly, the vertical filters will be of sizes $2 \times 1, 4 \times 1, ..., M \times 1$, where M is the height of the image in pixels. These vectors are filled half with -1 elements and half with 1. See below for examples of 1×4 and 4×1 filters.

$$\text{Example } 1 \times 4 \text{ Haar vector filter: } \begin{pmatrix} -1 & -1 & 1 & 1 \end{pmatrix}$$

$$\text{Example } 4 \times 1 \text{ Haar vector filter: } \begin{pmatrix} -1 \\ -1 \\ 1 \\ 1 \end{pmatrix}$$

Each of these filters is then convolved with the input image, and the output at each pixel becomes an element in the feature vector associated with that image. For this assignment, the implementation simply subtracted the sum of the negative pixel filter region from the sum of the positive pixel filter region for efficiency, rather than using a convolution function. In total, there were 11,940 elements in each feature vector (per image). This amount of processing can take quite some time, especially when there are hundreds or thousands of images to process. Consequently, the features vectors associated with each image were saved to a file so that feature extraction did not have to be performed each time the program was run.

1.3.2 Strong and weak classifier construction

The cascaded AdaBoost classifier uses an iterative method that continually updates its parameters to achieve an extremely low false positive rate. Several weak classifiers can constitute a strong classifier, which is used as part of one cascade in the classifier. Each weak classifier can be thought of as a function $weak(input, feature, threshold, polarity)$ that classifies the input based on its feature value when compared to the threshold. The polarity of the classifier determines whether the classifier assigns the input to the first or second of two classes if the feature value is greater than the threshold value. The steps to create a strong (and weak) classifier are outlined below:

1. Before beginning, create a matrix containing the feature vectors for all images representing both positive and negative labels. Additionally, a vector of labels identifying the true label for each row (or column) in the feature matrix is needed for training. From here forward, images containing the object of interest will be referred to as "positive images", and those not depicting the object of interest will be called "negative images".

2. Establish initial weights for the M positive and N negative images. The weights should be initialized to $\frac{1}{2M}$ and $\frac{1}{2N}$, respectively. These weights are initially set such that the total weight of the positive and negative images is 1.

3. Now, we can construct the weak classifiers as part of a single cascade. To do this, the following steps are taken:

   (a) Normalize the weights associated with the positive and negative images by dividing the weight associated with each image by the sum of the weights associated with all images.

   (b) Iterate over each feature associated with all images. That is, acquire a vector of each feature element from all images. For each feature, perform steps (c)-(e).

   (c) Sort the features and their corresponding image labels and weights in ascending order, based on the feature element values. We do this as an efficient precursor to computing the threshold for separating the set into two classes with $> 50\%$ accuracy.

(d) Compute the classification error if any of the feature elements was used as the threshold, for both polarities (referenced from ECE 661 Fall 2018 report 1):

$$\text{Error if polarity 1 is chosen: } e_1 = S^+ + T^- - S^-$$

$$\text{Error if polarity -1 is chosen: } e_{-1} = S^- + T^+ - S^+$$

- S^+: Sum of weights of positive images whose feature value is less than the current threshold.
- S^-: Sum of weights of negative images whose feature value is less than the current threshold.
- T^+: Sum of weights of positive images.
- T^-: Sum of weights of negative images.

The classification error is then calculated as $min(e_1, e_{-1})$. This error is a vector that represents the error that would result if each feature value was used as a threshold for the corresponding feature set in all images.

(e) Note the minimum error value ϵ that results. If it is less than the current minimum error, note the current feature vector index, the threshold (feature value) that resulted in the minimum error, and the polarity that produced the minimum error.

(f) Repeat steps (c)-(e) until all feature vectors have been tested. The feature index, threshold, and polarity parameters associated with the overall minimum error constitute a single weak classifier in the cascade.

4. After each weak classifier has been found, we have to update the parameters of the algorithm to find the next weak classifier in the cascade. We first compute the confidence parameters from the weak classifier:

$$\beta = \frac{\epsilon}{1-\epsilon} \text{ and } \alpha = ln(\frac{1}{\beta})$$

Keep note of the value of α for the weak classifier as it will be needed during testing.

5. Update the image weights w_i according to the following formula, where δ represents the classification disparity. That is, $\delta = 1$ if the classification of a particular image was incorrect, and $\delta = 0$ if it was correct.

$$w_{i,new} = w_{i,old} * \beta^{1-\delta}$$

6. Now, compute the parameters for determining the final classifications using the weak classifiers created so far. These weak classifiers constitute one strong classifier.

(a) First, we multiply α by the classifications assigned by the weak classifier. Each iteration, we will update this list by adding the new product to the sum of the products from the previous iterations. We will call this list c_α.

(b) Also, keep a running sum α_{tot} of the α values from all weak classifiers found so far. If desired, each α value can be multiplied by a constant to adjust the extent to which it affects the threshold on the next iteration.

7. Determine the final classifications assigned by the strong classifier so far. These are found by applying the threshold value α_{tot} to the list c_α. From this list, the false negative and positive rates can be found, as well as their true counterparts.

8. Ideally, the strong classifier should achieve 100% accuracy on positive images, such that no positive image is misclassified (i.e. the false positive rate is 0). However, this may not be possible from the current number of weak classifiers found so far. If we do achieve the desired performance rates, we can stop searching for more weak classifiers for the current cascade. If not, we continue finding more weak classifiers until we do, or until a maximum number of classifiers are found.

9. Once all weak classifiers for the current cascade have been found (either by reaching the termination conditions or by accumulating a set maximum number of classifiers), the data set needs to be revised such that only the incorrectly classified images remain. However, since we desire to achieve a very low false positive rate, we must also include all positive images, regardless of whether they were classified correctly or incorrectly by the current cascade. More simply, we remove all the correctly classified negative images from the data set, and repeat the steps above to create another cascade.

10. Cascade creation terminates when either the cumulative false positive and negative rates fall below a set threshold, or no negative images remain. Each cascade is then a strong classifier that will be applied during the testing phase. As with feature extraction, the final cascade set was saved to a file so that it could simply be loaded when it came time to test the test set.

1.3.3 Testing with a cascaded AdaBoost classifier

Testing with an AdaBoost classifier is a fairly simple process. Each weak classifier in each cascade is applied to the each image in the test set. More specifically, the weak classifier denoting the specific feature value to threshold for each image is applied with the polarity found during testing that produced the most accurate classification. The classifications found from the output of each weak classifier are then adjusted with the α values found during training and the α-sum threshold is re-applied to obtain the final classifications. The images that are classified as positive (whether their true label is positive or not) are kept in the data set for the next iteration, while the negatively classified samples are removed. Once again, the false positive and negative rates are monitored after each full cascade is applied, but this time it is to characterize the performance of the classifier, rather than to provide a termination condition. The testing procedure stops once no negative images remain in the data set.

2 Results

2.1 Task 1: Face recognition with PCA and LDA

Below are my results for Task 1, which involved classifying several different faces using Principal Component Analysis and Linear Discriminant Analysis methods.

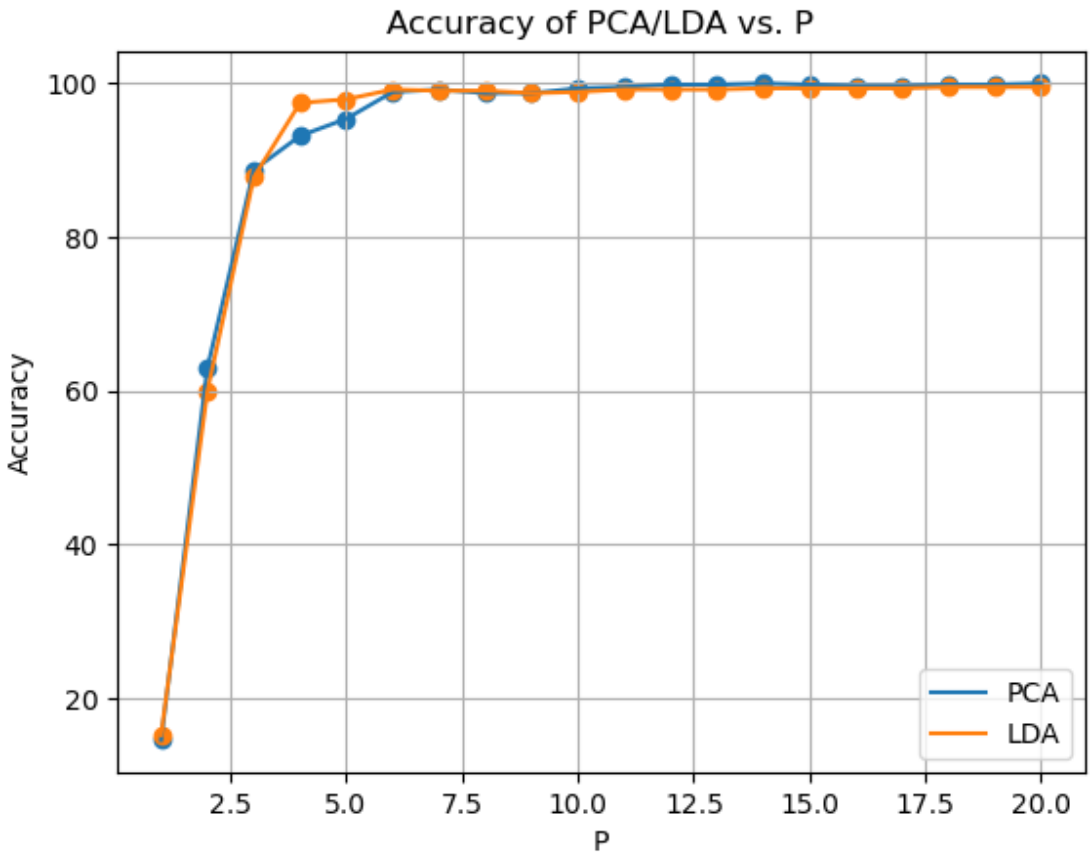

Figure 1: Results of PCA and LDA accuracy vs. classifier dimensionality P.

The plot above shows quite similar results for the performance of PCA and LDA. However, we note that LDA converges slightly faster to 99% accuracy than PCA. LDA remained above 99% accuracy after P hit 6, while PCA only remained above 99% accuracy after P hit 10. This matches the expected results: in general, LDA will converge faster than PCA. Near-perfect accuracy (99%) was selected over perfect (100%) accuracy as the convergence criterion to eliminate the effect of potential rounding errors in calculations. Additionally, both methods seemed to jump between 99% and 100% accuracy.

2.2 Task 2: AdaBoost classification

For this task, the Viola and Jones algorithm was implemented to identify whether an image contained a car or not. This algorithm uses a cascaded version of the AdaBoost classifier described in the theory section. A few notes:

- Several iterations were performed with limits on the number of weak classifiers per cascade. This is because my full implementation (with up to 25 weak classifiers per cascade) only required 4 stages, and thus didn't show the false negative and false positive trends as well. I tested the system with up to 1, 5, and 25 weak classifiers per cascade. Limiting the number of weak classifiers per cascade also dramatically improved training time.
- The maximum number of cascades allowed for the classifier was 10.
- From the plots below, we can see that during training, as the number of cascade stages increases, the false positive and negative rates both decrease. This is expected because the algorithm attempts to

minimize the false positive rate, giving it priority over the false negative rate. In all cases, both the false positive and negative rates tended towards zero after all cascade stages.

- The results of applying the cascaded classifier to the images in the test set show that the false positive rate is correctly minimized, with priority over the false negative rate. Both rates, after running all images through the complete classifier, show fairly low false positive and negative rates.
- There is a large spike in the false positive/negative rates for each test data set after cascade 2. I am not entirely sure what causes this, but it could be due to the algorithms tradeoff of minimizing the false positive rate at the expense of the false negative rate. Once that stabilizes, the false negative rate begins to drop as well. It could also be a result of a code flaw.

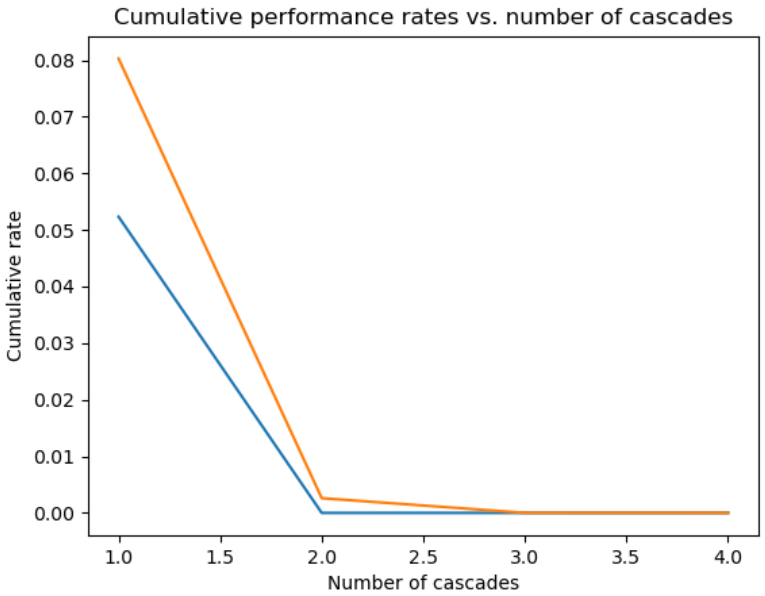

Figure 2: False positive (blue) and negative (orange) rates after training with up to 25 weak classifiers per cascade. The actual number of weak classifiers per cascade were [25, 25, 25, 15].

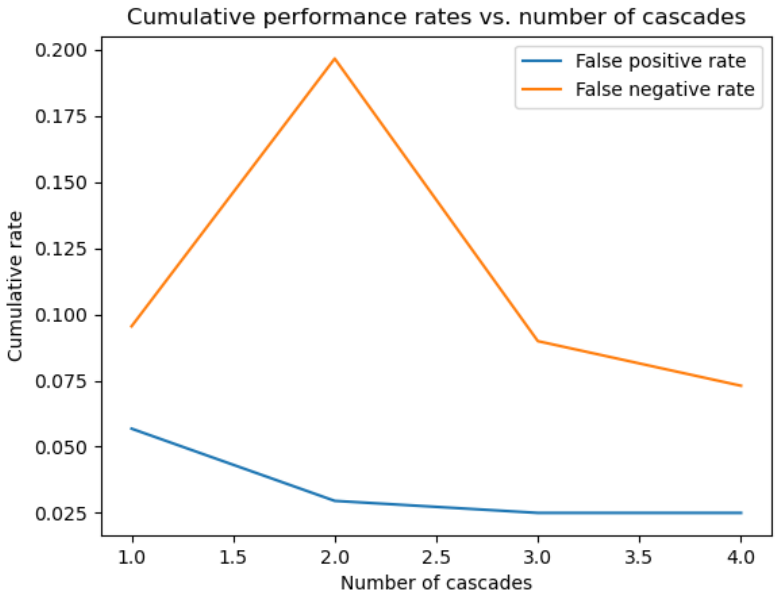

Figure 3: False positive and negative rates for the testing data set, when up to 25 weak classifiers per cascade were used.

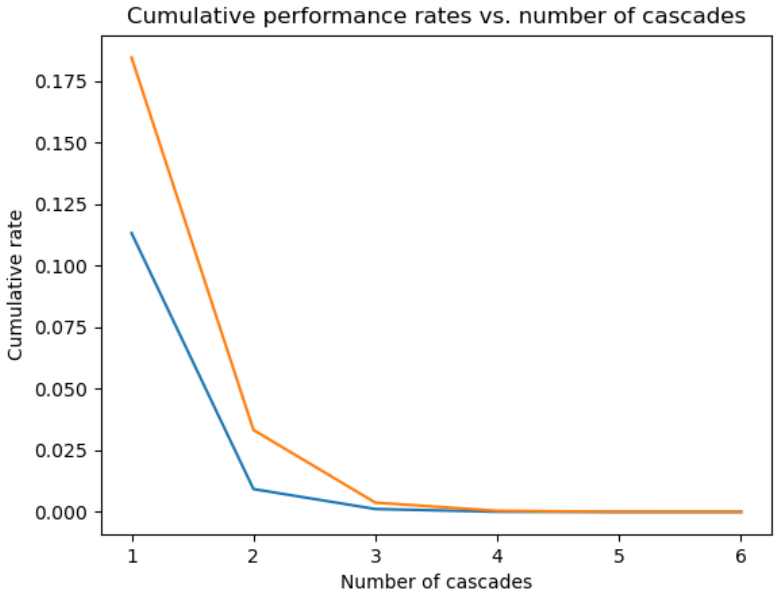

Figure 4: False positive (blue) and negative (orange) rates after training with up to 5 weak classifiers per cascade. The actual number of weak classifiers per cascade were [5, 5, 5, 5, 5, 3].

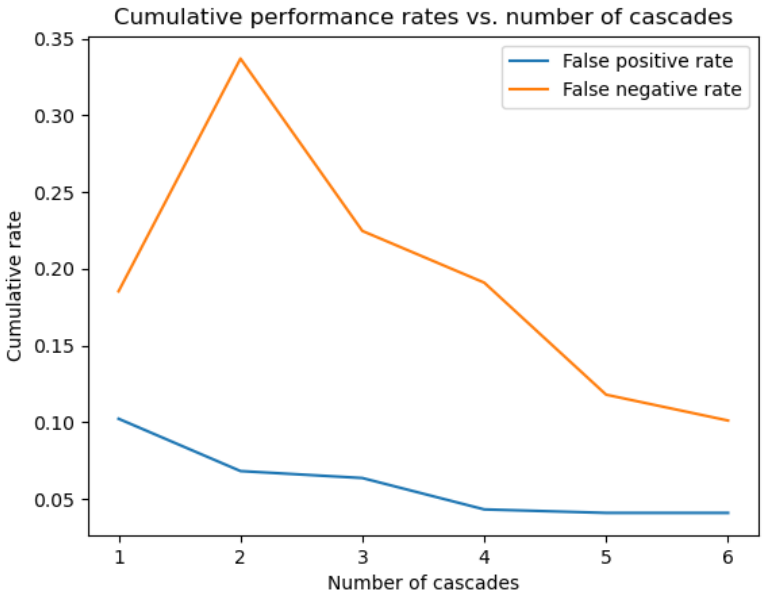

Figure 5: False positive and negative rates for the testing data set, when up to 5 weak classifiers per cascade were used.

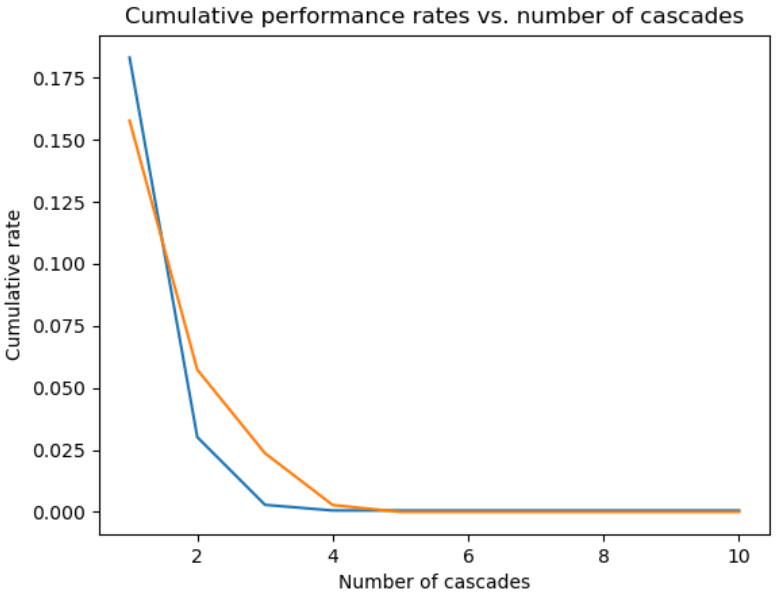

Figure 6: False positive (blue) and negative (orange) rates after training with only 1 weak classifier per cascade.

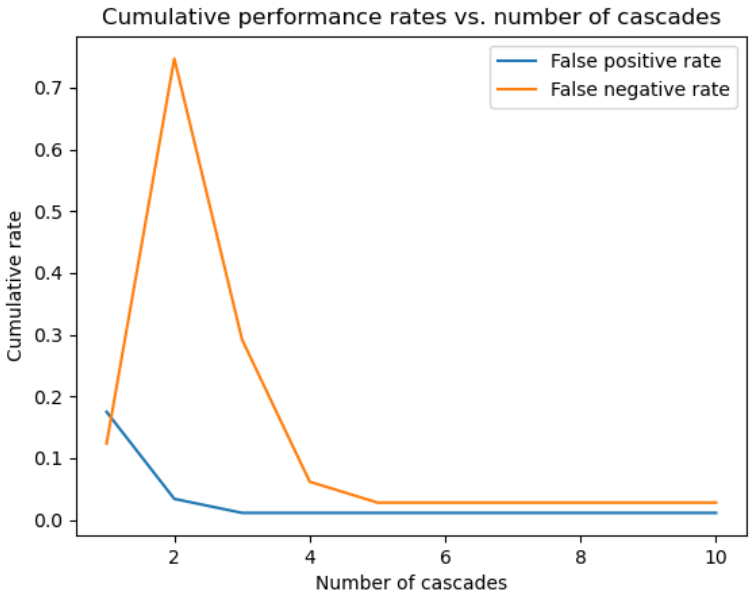

Figure 7: False positive and negative rates for the testing data set, when only one weak classifier was used per cascade.

3 Source Code

3.1 Task 1: Face recognition with PCA and LDA

```
## ============= FILE INFORMATION ============= ##
#
# Name:         Brian Helfrecht
# Email:        bhelfre@purdue.edu
# Course:       ECE 661
# Assignment:   Homework 11, Task 1
# Due date:     December 2, 2020
#
## ============= PACKAGE/FILE IMPORTS ============= ##
import numpy as np
import cv2 as cv
import math
import os
import time
import matplotlib.pyplot as plt
## ============= FUNCTION DEFINITIONS ============= ##
def vectorizeImg(img):
    #Ensure we use a grayscale image
    if (len(img.shape) > 2):
        img = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
    imgWidth, imgHeight = img.shape[1], img.shape[0]

    #Vectorize the image into a vector and normalize it
    imgVec = np.ravel(img)
    return imgVec / np.linalg.norm(imgVec)

def calcPCAData(imgVecs, N, IMG_SZ, P):
    #Compute the mean of the image set decompose X^TX
    meanImgVecs = np.reshape(np.mean(imgVecs, 1), (IMG_SZ, 1))
    xMat = imgVecs - meanImgVecs
    matToDecompose = np.dot(np.transpose(xMat), xMat)

    #Compute the eigenvalues and eigenvectors and sort them from largest to smallest
    eigenVals, eigenVecs = np.linalg.eig(matToDecompose)
    sortedEigVecs = [pt for _, pt in sorted(zip(eigenVals, eigenVecs), \
            key = lambda pair: pair[0], reverse = True)]

    #Compute the true eigenvectors and take the P largest
    finalEigVecs = np.zeros((IMG_SZ, N))
    for i in range(N):
        finalEigVec = np.dot(xMat, sortedEigVecs[i])
        finalEigVecs[:, i] = finalEigVec / np.linalg.norm(finalEigVec)
    return meanImgVecs, finalEigVecs[:, 0:P]

def calcLDAData(imgVecs, numLabels, IMGS_PER_LABEL, IMG_SZ, P):
    globalMean = np.reshape(np.mean(imgVecs, 1), (IMG_SZ, 1))
    classMeans = np.zeros((IMG_SZ, numLabels))
    inClassDiffVecs = np.zeros((IMG_SZ, numLabels*IMGS_PER_LABEL))

    #Populate the matrix of class means
    for i in range(numLabels):
        classMean = np.mean(imgVecs[:, i*IMGS_PER_LABEL:(i+1)*IMGS_PER_LABEL], 1)
        classMeans[:, i] = classMean
        inClassDiffVecs[:, i*IMGS_PER_LABEL:(i+1)*IMGS_PER_LABEL] = \
                imgVecs[:, i*IMGS_PER_LABEL:(i+1)*IMGS_PER_LABEL] - \
                np.reshape(classMean, (IMG_SZ, 1))
    meanMat = classMeans - globalMean
    matToDecompose = np.dot(np.transpose(meanMat), meanMat)

    #Compute the eigenvalues and eigenvectors and sort them from largest to smallest
    eigenVals, eigenVecs = np.linalg.eig(matToDecompose)
    sortedEigVals = sorted(eigenVals, reverse = True)
    sortedEigVecs = [pt for _, pt in sorted(zip(eigenVals, eigenVecs), \
            key = lambda pair: pair[0], reverse = True)]

    #Compute the true eigenvectors. Only 1 is "close to zero", by inspection.
    #This one vector makes the matrix singular, so it must be removed to form the
    #upper left submatrix of Y.
    finalEigVecs = np.zeros((IMG_SZ, numLabels-1))
    finalEigVals = np.zeros(numLabels-1)
    for i in range(numLabels-1):
        finalEigVec = np.dot(meanMat, sortedEigVecs[i])
        finalEigVecs[:, i] = finalEigVec / np.linalg.norm(finalEigVec)
        finalEigVals[i] = sortedEigVals[i]

    #Compute the eigen matrix decomposition
    eigenValMat = np.eye(numLabels-1) * finalEigVals
    DB = np.sqrt(np.linalg.inv(eigenValMat))
    Z = np.dot(finalEigVecs, DB)
    newEigVecDecompMat = np.dot(np.dot(np.transpose(Z), inClassDiffVecs), \
            np.transpose(np.dot(np.transpose(Z), inClassDiffVecs)))

    #Compute the eigenvectors of the new matrix
    eigenVals, eigenVecs = np.linalg.eig(newEigVecDecompMat)
    sortedEigVecs = [pt for _, pt in sorted(zip(eigenVals, eigenVecs), \
            key = lambda pair: pair[0], reverse = True)]

    #Compute the true eigenvectors and take the P largest
    finalEigVecs = np.zeros((IMG_SZ, numLabels - 1))
    for i in range(P):
        finalEigVec = np.dot(Z, sortedEigVecs[i])
        finalEigVecs[:, i] = finalEigVec / np.linalg.norm(finalEigVec)
    return globalMean, finalEigVecs[:, 0:P]

def predictLabel(featureMat, featureVec, numLabels, imgPerLabel, numNeighbors):
    classHits = np.zeros(numLabels, np.uint16)
    classMeans = np.zeros(numLabels) #Used to settle ties
    groupNums = np.arange(1, numLabels+1) #Needed since we trim the vectors

    #Calculate the Euclidean distance between the feature vectors
```

```
    dist = np.sqrt(np.sum((trainFeatureVecs - featureVec) ** 2, 0))

   #Find the N nearest neighbors, removing the selected entry each time
    for i in range(numNeighbors):
        minIdx = np.argmin(dist)
        nearIdx = int(minIdx / imgPerLabel)
        classHits[nearIdx] += 1 #Update the number of class occurences
        classMeans[nearIdx] += dist[nearIdx] #Update the total distance
        dist[minIdx] = float('inf') #Prevents re-selection

    #Determine the maximum hits (this could be more than 1 class!)
    maxHits = np.max(classHits)

    #Now, pick the class with the smallest average distance vector out of the
    #classes with the maximum hits
    classMeans = classMeans[classHits == maxHits] / maxHits
    groupNums = groupNums[classHits == maxHits]

    #The predicted class is the class with the most "hits" and smallest distance
    return groupNums[np.argmin(classMeans)]

## ============= MAIN CODE BEGINS BELOW ============= ##

IMG_SZ = 128 * 128
MAX_P = 20
IMGS_PER_LABEL = 21
NUM_LABELS = 30
NUM_NEIGHBORS = 1
TEST_DIR = './inputs/ECE661_2020_hw11_DB1/test/'
TRAIN_DIR = './inputs/ECE661_2020_hw11_DB1/train/'

pVals = list(np.arange(1, MAX_P+1))
pAcc  = []

''' ===== TRAINING ===== '''
#Acquire file names for the images in the desired directory
imgFiles = [img for img in os.listdir(TRAIN_DIR) if \
            os.path.isfile(os.path.join(TRAIN_DIR, img))]
N = len(imgFiles)
trainImgVecs = np.zeros((IMG_SZ, N))

#Read in each image and vectorize it
print('Vectorizing training images...')
for i in range(N):
    filename = imgFiles[i]
    img = cv.imread(TRAIN_DIR + filename)
    trainImgVecs[:, i] = vectorizeImg(img)

for P in range(1, MAX_P+1):
    #Compute feature vectors using PCA or LDA
    #meanImgVecs, pEigMat = calcPCAData(trainImgVecs, N, IMG_SZ, P)
    meanImgVecs, pEigMat = calcLDAData(trainImgVecs, NUM_LABELS, IMGS_PER_LABEL, IMG_SZ, P)
    trainFeatureVecs = np.dot(np.transpose(pEigMat), trainImgVecs - meanImgVecs)

    ''' ===== TESTING ===== '''
    #Acquire file names for the images in the desired directory
    imgFiles = [img for img in os.listdir(TEST_DIR) if \
                os.path.isfile(os.path.join(TEST_DIR, img))]
    N = len(imgFiles)
    correctPredictions = 0

    for i in range(N):
        #Read in each image and determine its true label
        filename = imgFiles[i]
        gtLabel = int(filename.strip().split('_')[0])

        #Vectorize the image and compute the feature vector
        img = cv.imread(TEST_DIR + filename)
        imgVec = np.reshape(vectorizeImg(img), (IMG_SZ, 1))
        featureVec = np.dot(np.transpose(pEigMat), imgVec - meanImgVecs)

        #Predict the class of the image
        predictedLabel = predictLabel(trainFeatureVecs, featureVec, \
                NUM_LABELS, IMGS_PER_LABEL, NUM_NEIGHBORS)
        #print('Prediction for %s: [%d, %d]' % (filename, gtLabel, predictedLabel))
        if (predictedLabel == gtLabel):
            correctPredictions += 1
    acc = correctPredictions / N * 100.0
    print('Overall accuracy for P = %d: %f' % (P, acc))
    pAcc.append(acc)

plt.plot(pVals, pAcc)
plt.ylabel('Accuracy')
plt.xlabel('P')
plt.title('Accuracy vs. P value')
plt.show()
```

3.2 Task 2: AdaBoost Classification

```
## ============= FILE INFORMATION ============= ##
#
# Name:         Brian Helfrecht
# Email:        bhelfre@purdue.edu
# Course:       ECE 661
# Assignment:   Homework 11, Task 2
# Due date:     December 2, 2020
#
## ============= PACKAGE/FILE IMPORTS ============= ##
import numpy as np
import cv2 as cv
import math
import os
import matplotlib.pyplot as plt
## ============= FUNCTION DEFINITIONS ============= ##
def extractFeatureVec(img):
    #Ensure we use a grayscale image
    if (len(img.shape) > 2):
        img = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
    imgWidth, imgHeight = img.shape[1], img.shape[0]

    #Create the feature vector object. Each element in the feature vector
    #is the convolution result of a Haar filter at a particular pixel location.
    featureVec = []
    filterWidths = np.arange(2, imgWidth, 2)
    filterHeights = np.arange(2, imgHeight, 2)

    #Compute horizontal features
    for filterWidth in filterWidths:
        #Apply the effective convolution across columns, then down rows
        for y in range(imgHeight):
            for x in range(imgWidth - filterWidth + 1):
                #Compute the feature element. This will be the sum of all pixels
                #in the "+1" filter region minus the sum of all pixels in
                #the "-1" filter region. (x, y) denotes the top left corner of
                #the filter. The filter has structure: [-1 | 1]
                negSum = np.sum(img[y:y+1, x:int(x+filterWidth/2)]).astype(np.int32)
                posSum = np.sum(img[y:y+1, int(x+filterWidth/2):x+filterWidth]).astype(np.int32)
                featureVec.append(posSum - negSum)

    #Compute vertical features
    for filterHeight in filterHeights:
        #Apply the effective convolution down rows, then across columns
        for x in range(imgWidth):
            for y in range(imgHeight - filterHeight + 1):
                #Compute the feature element, similar to the horizontal case
                negSum = np.sum(img[y:int(y+filterHeight/2), x:x+1]).astype(np.int32)
                posSum = np.sum(img[int(y+filterHeight/2):y+filterHeight, x:x+1]).astype(np.int32)
                featureVec.append(posSum - negSum)

    return featureVec

def createWeakClassifier(features, labels, normWeights):

    bestClassifier = None
    bestClassifierErr = float('inf')

    #Loop over all features associated with each image
    for f in range(features.shape[1]):
        #Sort the features, labels, and weights in according to feature values
        featureVecRow = features[:, f]
        featureVecCol = featureVecRow.reshape((features.shape[0], 1))
        featureData = np.concatenate((featureVecCol, labels, normWeights), 1)
        sortedData = [pt for _, pt in \
                sorted(zip(featureVecRow, featureData), key = lambda pair: pair[0])]
        sortedData = np.reshape(sortedData, (features.shape[0], 3))
        sortedFeatures = sortedData[:, 0]
        sortedLabels = sortedData[:, 1]
        sortedWeights = sortedData[:, 2]

        #Now, find parameters for calculating the error.
        totPosWeight = np.sum(normWeights[labels == 1])
        totNegWeight = np.sum(normWeights[labels == 0])
        sortedPosWeights = sortedWeights[np.where(sortedLabels == 1)[0]]
        sortedNegWeights = sortedWeights[np.where(sortedLabels == 0)[0]]

        #Making the weight vectors the same size
        finalPosWeights = np.zeros((features.shape[0], 1))
        finalNegWeights = np.zeros((features.shape[0], 1))
        finalPosWeights[np.where(sortedLabels == 1)[0], 0] = sortedPosWeights
        finalNegWeights[np.where(sortedLabels == 0)[0], 0] = sortedNegWeights

        #Computing the error
        cumPosWeights = np.cumsum(finalPosWeights)
        cumNegWeights = np.cumsum(finalNegWeights)
        errPol1 = cumPosWeights + totNegWeight - cumNegWeights #Error associated with polarity 1
        errPol2 = cumNegWeights + totPosWeight - cumPosWeights #Error associated with polarity -1
        errPol1 = np.reshape(errPol1, (len(errPol1), 1))
        errPol2 = np.reshape(errPol2, (len(errPol2), 1))
        errorVecs = np.concatenate((errPol1, errPol2), 1)
        minIdx = np.unravel_index(errorVecs.argmin(), errorVecs.shape)
        minErr = np.min(errorVecs)

        #Update the weak classifier with the new feature vector if it classifies
        #better than the currently selected one
        if (minErr < bestClassifierErr):
            bestClassifierErr = minErr
            threshold = sortedFeatures[minIdx[0]]
            featureIdx = f

            #Compute the polarity we should use for the weak classifier based
            #on which error vector the minimum came from, and
            #assign the threshold and feature index for the best classifier
            if (minIdx[1] == 0): #Error from the first vector
```

```
                polarity = 1
                classifications = featureVecCol >= threshold
            else: #Error from the second vector
                polarity = -1
                classifications = featureVecCol < threshold

            bestClassifier = [featureIdx, threshold, polarity, \
                    minErr, classifications]

    return bestClassifier

def createCascade(features, labels, T, weightScalar):
    #Establish weights
    numPos = np.sum(labels == 1)
    numNeg = np.sum(labels == 0)
    posWeight = 1 / (2 * numPos)
    negWeight = 1 / (2 * numNeg)
    weights = np.ones((numPos + numNeg, 1))

    #Normalize the weights for the first time
    weights[:numPos] = weights[:numPos] * posWeight
    weights[numPos:] = weights[numPos:] * negWeight
    normWeights = weights

    weakClassifiers = []
    sumAlphaClassifiers = np.zeros((features.shape[0], 1))
    aThresh = 0

    for t in range(T):
        #Find the weak classifiers
        print('Finding weak classifier ' + str(t) + '...')
        normWeights = normWeights / np.sum(normWeights)
        weakClass = createWeakClassifier(features, labels, normWeights)
        classFeat, classThresh, classPol, classErr, \
                classifications = weakClass #Unpack the weak classifier params

        #Compute confidence paramters
        Bt = classErr / abs(1 - classErr + 1e-6)
        at = math.log(1 / abs(Bt + 1e-6))
        weakClass.append(at)
        print(weakClass[:4])
        weakClassifiers.append(weakClass)

        #Update the weights for the next iteration
        classDisparity = classifications != labels
        normWeights = normWeights * (Bt ** (1 - classDisparity))

        #Now, compute the parameters for the strong cascade, and classify the samples
        sumAlphaClassifiers = sumAlphaClassifiers + (at * classifications)
        aThresh = aThresh + (at * WEIGHT_SCALAR)
        strongClassifier = sumAlphaClassifiers >= aThresh

        #Update the performance rates
        falsePosRate = np.sum(strongClassifier[numPos:] == 1) / numNeg
        falseNegRate = 1 - (np.sum(strongClassifier[:numPos] == 1) / numPos)

        print('Percent positive images incorrectly classified:', falseNegRate * 100)
        print('Percent negative images incorrectly classified:', falsePosRate * 100)

        #Stop searching for weak classifiers if performance criteria reached
        if (falsePosRate <= 0.5 and falseNegRate <= 0):
            break

    #Revise the data set to now contain only the incorrectly classified images.
    #Recall that our cascade was designed to always achieve 100% positive detection,
    #so we only need to include the negative features that were incorrectly classified
    #in addition to all positive features.
    revisedFeatures = features[:numPos, :]
    negFeatures = features[numPos:, :]
    negFeatureMask = classifications[numPos:]
    negFeaturesMasked = negFeatures[np.where(negFeatureMask == 1), :][0]
    negFeaturesMasked = np.reshape(negFeaturesMasked, \
            (len(negFeaturesMasked), np.size(features, 1)))
    revisedFeatures = np.concatenate((revisedFeatures, negFeaturesMasked), 0)
    revisedLabels = np.concatenate((np.ones((numPos, 1), np.uint8), \
            np.zeros((len(negFeaturesMasked), 1), np.uint8)), 0)
    perfRates = [falsePosRate, falseNegRate] #Package performance values

    return revisedFeatures, revisedLabels, perfRates, weakClassifiers

def testCascade(cascade, features, weightScalar):
    #Test the cascade stage through all its weak classifiers
    sumAlphaClassifiers = np.zeros((features.shape[0], 1))
    aThresh = 0

    #Apply all weak classifiers to the feature set
    for weakClassifier in cascade:
        #Unpack the weak classifier parameters
        featureIdx = weakClassifier[0]
        threshold = weakClassifier[1]
        polarity = weakClassifier[2]
        at = weakClassifier[5]

        #Extract the feature for each image
        featureVecRow = features[:, featureIdx]
        featureVecCol = featureVecRow.reshape((features.shape[0], 1))

        if (polarity == 1):
            classifications = featureVecCol >= threshold
        elif (polarity == -1):
            classifications = featureVecCol < threshold

        #Create weighted classification vectors
        sumAlphaClassifiers = sumAlphaClassifiers + (at * classifications)
        aThresh = aThresh + (at * weightScalar)

    #Perform final clsasifications
```

```
    finalClassifications = sumAlphaClassifiers >= aThresh
    return finalClassifications

## ============= MAIN CODE BEGINS BELOW ============= ##

TRAIN_POS_DIR = './inputs/ECE661_2020_hw11_DB2/train/positive/'
TRAIN_NEG_DIR = './inputs/ECE661_2020_hw11_DB2/train/negative/'
TEST_POS_DIR  = './inputs/ECE661_2020_hw11_DB2/test/positive/'
TEST_NEG_DIR  = './inputs/ECE661_2020_hw11_DB2/test/negative/'

''' ===== FEATURE EXTRACTION ===== '''
'''
#Extract features for positive and negative images (those containing and
#not containing cars, respectively).

#Compute the feature matrix for the positive data set
#Enumerate the images in the given directory
imgFiles = [img for img in os.listdir(TEST_POS_DIR) if \
            os.path.isfile(os.path.join(TEST_POS_DIR, img))]
featureMat = []

#Extract features
for i in range(len(imgFiles)):
    filename = imgFiles[i]
    print('Processing image: ' + filename + '...')
    img = cv.imread(TEST_POS_DIR + filename)
    featureMat.append(extractFeatureVec(img))
featureMat = np.reshape(featureMat, (len(featureMat), len(featureMat[0])))
np.savetxt('./outputs/t2/test_pos_featureMat.txt', featureMat, '%d')

#Compute the feature matrix for the negative data set
#Enumerate the images in the Training-Negative directory
imgFiles = [img for img in os.listdir(TEST_NEG_DIR) if \
            os.path.isfile(os.path.join(TEST_NEG_DIR, img))]
featureMat = []

#Extract features
for i in range(len(imgFiles)):
    filename = imgFiles[i]
    print('Processing image: ' + filename + '...')
    img = cv.imread(TEST_NEG_DIR + filename)
    featureMat.append(extractFeatureVec(img))
featureMat = np.reshape(featureMat, (len(featureMat), len(featureMat[0])))
np.savetxt('./outputs/t2/test_neg_featureMat.txt', featureMat, '%d')
'''

''' ===== WEAK CLASSIFIER CONSTRUCTION ===== '''
'''
MAX_CLASSIFIERS_PER_CASCADE = 1
MAX_CASCADES = 10
WEIGHT_SCALAR = 0.5

#Read in the feature matrices
print('Loading training feature matrices...')
posTrainFeatures = np.loadtxt('./outputs/t2/train_pos_featureMat.txt', np.int16)
negTrainFeatures = np.loadtxt('./outputs/t2/train_neg_featureMat.txt', np.int16)

#Concatenate the training features and create labels (1 for positive images, 0 for negative)
trainFeatures = np.concatenate((posTrainFeatures, negTrainFeatures), 0)
posLabels = np.ones((posTrainFeatures.shape[0], 1), np.uint8)
negLabels = np.zeros((negTrainFeatures.shape[0], 1), np.uint8)
trainLabels = np.concatenate((posLabels, negLabels), 0)

cascadeStages = []
features = trainFeatures
labels = trainLabels
cumFalsePosRate = 1
cumFalseNegRate = 1
falsePosRateVec = []
falseNegRateVec = []

#Create the cascades
for cascadeNum in range(MAX_CASCADES):
    print('Creating cascade ' + str(cascadeNum) + '...')
    features, labels, perfRates, cascade = createCascade(features, labels, \
            MAX_CLASSIFIERS_PER_CASCADE, WEIGHT_SCALAR)
    cascadeStages.append(cascade)

    #Update current performance
    falsePosRate = perfRates[0]
    falseNegRate = perfRates[1]
    cumFalsePosRate = cumFalsePosRate * falsePosRate
    cumFalseNegRate = cumFalseNegRate * falseNegRate
    falsePosRateVec.append(cumFalsePosRate) #Keep track for plotting
    falseNegRateVec.append(cumFalseNegRate) #Keep track for plotting

    #Terminate early if the error rate is less 1e-6
    if (cumFalsePosRate < 1e-6 and cumFalseNegRate < 1e-6):
        print('Reached performance criteria.')
        break

    print('Negative images remaining:', np.sum(labels == 0))

    #Terminate early if no negative images remain
    if (np.sum(labels == 0) == 0):
        print('No negative images remaining.')
        break

#Plot performance rates as a function of the number of cascades
cascadeVals = (np.arange(len(falsePosRateVec)) + 1).astype(np.uint8)
plt.plot(cascadeVals, falsePosRateVec)
plt.plot(cascadeVals, falseNegRateVec)
plt.ylabel('Cumulative rate')
plt.xlabel('Number of cascades')
plt.title('Cumulative performance rates vs. number of cascades')
plt.show()
```

```
#Save the cascade to a file
np.save('./outputs/t2/cascades.npy', np.array(cascadeStages, dtype=object))
'''
''' ===== TESTING ===== '''
WEIGHT_SCALAR = 0.5

#Load in the classifier cascade and test set features
print('Loading cascade stages...')
cascades = np.load('./outputs/t2/cascades.npy', allow_pickle = True)
print('Loading test data...')
posTestFeatures = np.loadtxt('./outputs/t2/test_pos_featureMat.txt', np.int16)
negTestFeatures = np.loadtxt('./outputs/t2/test_neg_featureMat.txt', np.int16)
features = np.concatenate((posTestFeatures, negTestFeatures), 0)
posLabels = np.ones((posTestFeatures.shape[0], 1), np.uint8)
negLabels = np.zeros((negTestFeatures.shape[0], 1), np.uint8)
labels = np.concatenate((posLabels, negLabels), 0)

falsePosRateVec = []
falseNegRateVec = []
numPos = np.sum(labels == 1)
numNeg = np.sum(labels == 0)
initNumPos = numPos
initNumNeg = numNeg
cascadeNum = 0
totPosClassNeg = 0
totNegClassNeg = 0

print('Testing...')
for cascade in cascades:
    classifications = testCascade(cascade, features, WEIGHT_SCALAR)

    #Compute performance parameters. We can only index the arrays with the
    #positive images because we remove negative images each iteration.
    numPosClassPos = np.sum(classifications[:numPos] == 1)
    totPosClassNeg = totPosClassNeg + numPos - numPosClassPos
    numNegClassPos = np.sum(classifications[numPos:] == 1)
    newNumNegClassNeg = numNeg - numNegClassPos
    totNegClassNeg = totNegClassNeg + newNumNegClassNeg
    falseNegRateVec.append(totPosClassNeg / initNumPos)
    falsePosRateVec.append(1 - (totNegClassNeg / initNumNeg))

    #Only use positively classified samples for the next cascade
    features = features[np.where(classifications == 1), :][0]
    labels = labels[np.where(classifications == 1)[0]]
    numPos = np.sum(labels == 1)
    numNeg = np.sum(labels == 0)

    cascadeNum += 1

    #No more positive images to classify, so stop.
    if (numPos == 0):
        print('No more positive images.')
        break

print(falsePosRateVec)
print(falseNegRateVec)

#Plot the results
cascadeVals = (np.arange(cascadeNum) + 1).astype(np.uint8)
plt.plot(cascadeVals, falsePosRateVec, label='False positive rate')
plt.plot(cascadeVals, falseNegRateVec, label='False negative rate')
plt.ylabel('Cumulative rate')
plt.xlabel('Number of cascades')
plt.title('Cumulative performance rates vs. number of cascades')
plt.legend()
plt.show()
```