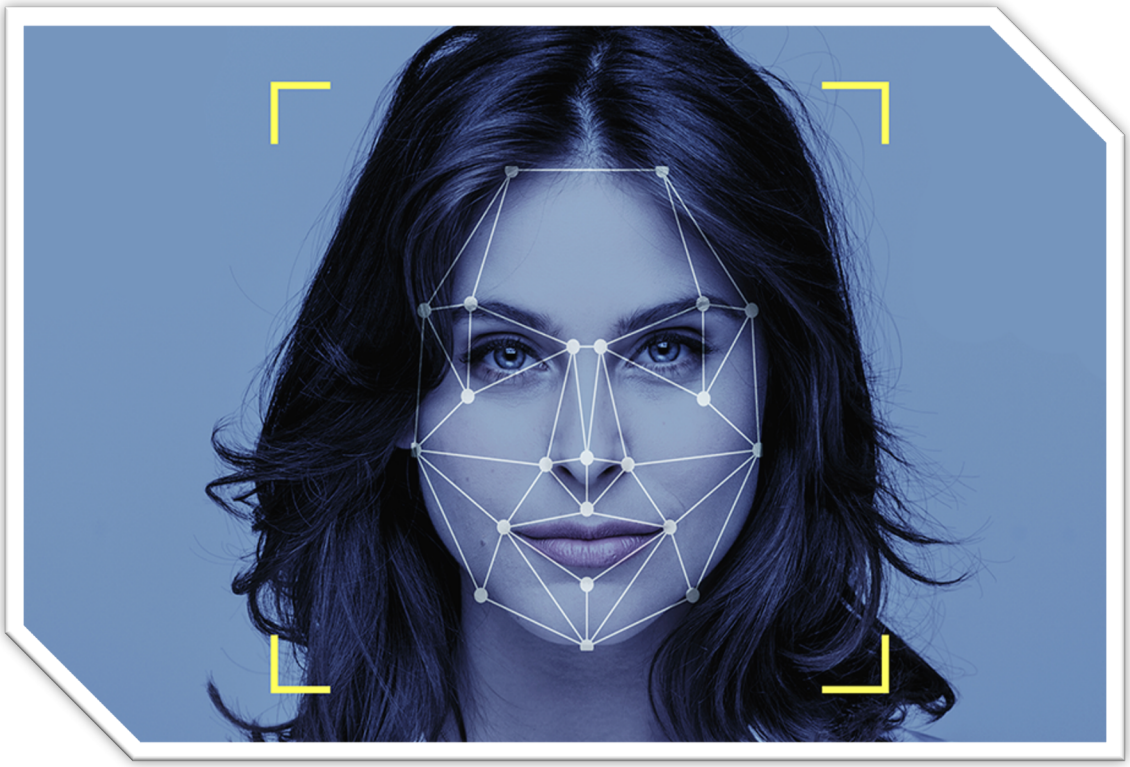
Pattern Recognition

Assignment 1

Face Recognition



Prepared by:

Name	ID
Youssef Samuel Nachaat Labib	6978
Youssef Amr Ismail Othman	6913
Arsany Mousa Fathy Rezk	6927

Contents

1.Introduction

1.1 Problem statement

It is required to perform face recognition using PCA and LDA as an algorithm for dimensionality reduction, then for classification K-NN algorithm is used.

Two classification problems are required:

- Classification of a test face as one of 40 possible classes.
- Face vs non-face classification problem. Given a test image, is this image a face or a non-face?

1.2 Dataset description

There are 10 different images of 40 distinct people. The images were taken at different times, varying lighting slightly, facial expressions (open/closed eyes, smiling/non-smiling) and facial details (glasses/no-glasses). All the images are taken against a dark homogeneous background and the subjects are in up-right, frontal position (with tolerance for some side movement).

The data matrix is a numpy array of size (400 x 10304), where 400 is the total number of faces because there 40 classes and and 10 images per class. The number of features for each image is 10304.

In case of faces vs non faces classification problem another dataset is used of size (800 x 10304) containing this time 800 images. First, 400 non face then another 400 for faces.

2.Generating the dataset

2.1 Download the datasets

```
import opendatasets as od
import pandas
#download given dataset for the original problem
od.download(
    "https://www.kaggle.com/datasets/kasikrit/att-database-of-faces")
#download dataset for face-nonface
od.download(
    "https://www.kaggle.com/datasets/arsanymousafathy/facenonfacedata")
```

The **opendatasets** package is used to download datasets from Kaggle. So, first the dataset of faces is downloaded. Then the dataset that we built is downloaded, which contains some faces and non-faces images.

2.2 Dataset of faces

```
###################################
data_size = 400
n_perclass = 10
number_of_classes = 40
n_eig_used = 39
reg_param = 0
confusion_labels=[i for i in range(1, 41)]
###################################
folder_path = '/content/att-database-of-faces'
subdirs = [f.path for f in os.scandir(folder_path) if f.is_dir()]
data_matrix=np.empty([400, 10304])
label_vector=np.empty([400,1])
i = 0
for subdir, dirs, files in os.walk(folder_path):
  for filename in files:
    if filename != 'README':
      id = get_id(subdir)
      image_path=''+subdir+'/'+filename
      img = Image.open(image_path)
      label_vector[i] = id
      data_matrix[i] = np.array(img).flatten()
      i=i+1
```

- **data_size** is set to 400, which corresponds to the total number of images in the dataset
- **n_perclass** is set to 10, which corresponds to the number of images per person in the dataset
- **number_of_classes** is set to 40, which corresponds to the number of different people in the dataset
- **n_eig_used** is set to 39, which will be used later in the LDA algorithm.
- **confusion_labels** is a list containing the integer values 1 through 40, which will be used later to create a confusion matrix
- **folder_path** is set to the path of the folder containing the image data
- **subdirs** is created using **os.scandir()** to list all subdirectories in the **folder_path**
- **data_matrix** is created as a 400x10304 numpy array (400 images with 10304 pixels per image)
- **label_vector** is created as a 400x1 numpy array to hold the labels (person IDs) for each image
- **i** is set to 0 to start iterating through each image in the dataset
- **os.walk()** is used to iterate through each subdirectory and file in the **folder_path**
- If the filename is not 'README', then the ID for the person in the image is extracted using **get_id(subdir)**
- The image is opened using **Image.open(image_path)**

- The label for the image is stored in **label_vector** at index **i**
- The pixel values for the image are stored in **data_matrix** at index **i** as a flattened numpy array
- **i** is incremented by 1 to move on to the next image

In summary, this code loads in image data for face recognition by iterating through each image in each folder, extracting the ID for the person in the image, and storing the pixel values and label in numpy arrays.

2.3 Dataset of faces vs non faces

```
data_size = 800
n_perclass = 400
number_of_classes = 2
n_eig_used = 3
confusion_labels=[1,2]
reg_param = 1
data_matrix=np.empty([data_size , 10304])
# Set the directory containing the images you want to resize and convert
directory1 = '/content/facenonfacedata/face-vs-non-face/non-faces'
directory2 = '/content/facenonfacedata/face-vs-non-face/faces'
subdirs1 = [f.path for f in os.scandir(directory1) if f.is_dir()]
subdirs2 = [f.path for f in os.scandir(directory2) if f.is_dir()]
label_vector=np.empty([data_size,1])
# Set the target size for the resized images
target_size = (92, 112)
i=0
label_vector[0:400] = 1 #non faces
label_vector[400:data_size] = 2 #faces
for subdir1, dirs, files in os.walk(directory1):
  for filename in files:
      image_path=''+subdir1+'/'+filename
      img_non_face = cv2.imread(image_path)
      # Resize the image
      resized_img = cv2.resize(img_non_face, target_size)
      # Convert the image to grayscale
      gray_img = cv2.cvtColor(resized_img, cv2.COLOR_BGR2GRAY)
      image_vector = gray_img.flatten()
      data_matrix[i] = image_vector
      i=i+1
for subdir2, dirs, files in os.walk(directory2):
  for filename in files:
      image_path=''+subdir2+'/'+filename
      img = Image.open(image_path)
      data_matrix[i] = np.array(img).flatten()
      i=i+1
```

The dataset consists of 800 images, with 400 non-face images and 400 face images.

The code loads images from two directories, "non-faces" and "faces," resizes them to a target size of 92x112 pixels, converts them to grayscale, and then flattens them into a 1D vector of length 10304 (92*112). These vectors are then stored in a numpy array "data_matrix."

The label vector "label_vector" is also created with a size of 800x1. The first 400 entries are set to 1, indicating non-faces, and the remaining 400 entries are set to 2, indicating faces.

3.User defined functions

3.1 'get_id'

```
[ ]  def get_id(s):
         id = s[-2:]
         if id[0] == 's':
           id = id[-1:]
         return id
```

The **get_id** is used to get the id of a specific image given as input the file name of the image. As an example, suppose the file name is: 's20', the function just take the 20 as id and returns it.

3.2 'split_even_odd'

```
[ ]  def split_even_odd(data_matrix, label_vector):
       testing_set=data_matrix[::2] #slicing notation: [start:stop:step]
       training_set=data_matrix[1::2]
       testing_labels=label_vector[::2]
       training_labels=label_vector[1::2]
       return training_set, testing_set, training_labels, testing_labels
```

The **split_even_odd** function is used to split a dataset into training and testing sets based on even and odd indices.

Variables:

- **data_matrix**: A numpy array representing the input data, where each row represents a sample, and each column represents a feature.
- **label_vector**: A numpy array containing the corresponding labels for the samples in **data_matrix**.

The function first uses slicing notation to extract even-indexed rows from **data_matrix** as the testing set, and odd-indexed rows as the training set. It then does the same for **label_vector**.

Finally, the function returns the training and testing data and labels.

3.3 'split'

```
[ ]  def split(data_matrix, label_vector, train_portion, n_classes, n_perclass):
       test_portion = 1-train_portion
       data_size = data_matrix.shape[0]
       feature_size = data_matrix.shape[1]
       training_data_size = int(train_portion * data_size)
       test_data_size = int(test_portion * data_size)
       n_trainperclass = int(train_portion * n_perclass)
       n_testperclass = int(test_portion * n_perclass)
       testing_set = np.empty([test_data_size, feature_size])
       training_set = np.empty([training_data_size, feature_size])
       testing_labels=np.empty([test_data_size,1])
       training_labels=np.empty([training_data_size,1])
       for i in range(n_classes):
         training_set[i*n_trainperclass:(i*n_trainperclass)+n_trainperclass, :] = data_matrix[i*n_perclass:(i*n_perclass)+n_trainperclass, :]
         testing_set[i*n_testperclass:(i*n_testperclass)+n_testperclass, :] = data_matrix[(i*n_perclass)+n_trainperclass:(i*n_perclass)+n_perclass, :]
         testing_labels[i*n_testperclass:(i*n_testperclass)+n_testperclass, :] = label_vector[(i*n_perclass)+n_trainperclass:(i*n_perclass)+n_perclass, :]
         training_labels[i*n_trainperclass:(i*n_trainperclass)+n_trainperclass] = label_vector[i*n_perclass:(i*n_perclass)+n_trainperclass]
       return training_set, testing_set, training_labels, testing_labels
```

The **split** function is used to split a dataset into training and testing sets while ensuring that there are equal numbers of samples from each class in both sets.

Variables:

- **data_matrix**: The input data, where each row represents a sample, and each column represents a feature.
- **label_vector**: The corresponding labels for the samples in **data_matrix**.
- **train_portion**: A float representing the percentage of data to be used for training, where the remaining percentage is used for testing.
- **n_classes**: An integer representing the number of classes in the dataset.
- **n_perclass**: An integer representing the number of samples per class in the dataset.

The function first calculates the size of the training and testing data based on the **train_portion** parameter. It then calculates the number of samples to be used for training and testing per class based on the **train_portion** and **n_perclass** parameters.

The function then initializes numpy arrays for the training and testing data and labels, and loops through each class to add the corresponding samples and labels to the sets. Finally, the function returns the training and testing data and labels.

3.4 'split_nonfaces'

```
[ ]  def split_nonfaces(data_matrix, label_vector):
       training_set, testing_set_d, training_labels, testing_labels_d = split_even_odd(data_matrix, label_vector)
       testing_set=np.empty([100,10304])
       testing_labels=np.empty([100,1])
       testing_set[0:50,:]=testing_set_d[0:50,:]
       testing_labels[0:50,:]=testing_labels_d[0:50,:]
       testing_set[50:100,:]=testing_set_d[-50:,:]
       testing_labels[50:100,:]=testing_labels_d[-50:,:]
       return training_set, testing_set, training_labels, testing_labels
```

This function splits a dataset into a training set and a testing set for a face recognition task. However, unlike the **split** function used in the **pca** function, this function splits the dataset in an unusual way. It first splits the dataset using the **split_even_odd** function, which simply separates the dataset into training and testing sets by alternating between even and odd indices. Then, it

further splits the testing set into two parts: the first 50 samples and the last 50 samples. These two parts are combined with the first 50 samples and last 50 samples of the original testing set, respectively, to form a new testing set of size 100.

The function returns the training set, new testing set, training labels, and new testing labels. This is done because in the face vs non face classification we need to maintain the size of the testing set constant while changing the size of the training set, to see how the accuracy of the program increases when the training data increases while fixing the size of the testing.

3.5 'knn'

```
[ ]  def knn (projected_testing_set, projected_training_set, number_of_classes, training_labels, euc_dist_matrix, k):
       test_data_size = projected_testing_set.shape[0]
       classify_vectorknn=np.empty([test_data_size,1])
       for j in range(euc_dist_matrix.shape[0]):
         voting = np.zeros((number_of_classes,2))
         sorted_indices = np.argsort(euc_dist_matrix[j])     # Get indices that would sort the array in ascending order
         least_k_indices = sorted_indices[:k]
         for v in range(k):
           id_index = least_k_indices[v]
           id = training_labels[id_index] - 1
           id = int(id)
           voting[id,0] = voting[id,0] + 1
           voting[id,1] = voting[id,1] + (euc_dist_matrix[j,id_index])
         max_value = np.amax(voting[:,0], axis=0)
         indices = np.where(voting[:,0] == max_value)
         indices = np.array(indices)
         indices=indices[0,:]
         min = indices[0]
         for ind in range(len(indices)):
           x1 = int(voting[indices[ind],1])
           x2 = int(voting[min,1])
           if x1 < x2:
             min = indices[ind]
         winner = min + 1
         classify_vectorknn[j] = winner
       return classify_vectorknn
```

The **knn** function is used to classify samples in a testing set using the k-nearest neighbors algorithm.

Variables:

- **projected_testing_set**: The testing set data that has been projected onto a lower-dimensional space.
- **projected_training_set**: The training set data that has been projected onto the same lower-dimensional space as **projected_testing_set**.
- **number_of_classes**: An integer representing the number of classes in the dataset.
- **training_labels**: The corresponding class labels for the samples in **projected_training_set**.
- **euc_dist_matrix**: A numpy array representing the pairwise Euclidean distances between each sample in the testing set and each sample in the training set.

- **k**: An integer representing the number of nearest neighbors to consider for classification.

The function loops through each sample in **projected_testing_set** and performs the following steps:

1. Finds the **k** nearest neighbors in **projected_training_set** to the current sample in **projected_testing_set**.
2. Computes the weighted votes of each class based on the distances to the **k** nearest neighbors.
3. Determines the class with the highest weighted vote.
4. Assigns the class label of the testing sample to the winning class.
5. Returns an array containing the predicted class labels for each sample in the testing set.

Note: **training_labels** are assumed to start from 1, so **training_labels[id_index] - 1** is used to convert the label to a zero-based index.

3.6 'pca'

```
def pca(D,label_vector, alpha, train_portion, n_classes, n_perclass, k_knn, normal_split):
  # Splitting the dataset
  if normal_split ==False :
    training_set, testing_set, training_labels, testing_labels = split_nonfaces(D, label_vector)
  elif train_portion == 0.5:
    training_set, testing_set, training_labels, testing_labels = split_even_odd(D, label_vector)
  else:
    training_set, testing_set, training_labels, testing_labels = split(D, label_vector, train_portion, n_classes, n_perclass)
  training_data_size = training_labels.shape[0]
  print(training_data_size)
  test_data_size = testing_labels.shape[0]
  print(test_data_size)
  n_trainperclass = int(train_portion * n_perclass)
  #Mean Vector
  mean_vector = np.mean(training_set, axis=0)
  #Centering the data
  Z_training = training_set - mean_vector
  Z_training_transpose = np.transpose(Z_training)
  #Covariance Matrix
  cov_mat = (Z_training_transpose@Z_training)/Z_training.shape[0] #the symbol @ stands for matrix multiplication.
  #Eigen Values and Eigen Vectors
  eig_values, eig_vectors = np.linalg.eigh(cov_mat)
  indices = eig_values.argsort()[::-1]
  eig_values_vector = eig_values[indices]
  eig_values = np.diag(eig_values_vector)
  eig_vectors = eig_vectors[:,indices]
```

```
    if isinstance(k_knn, (int, float)):
      k_knn=[k_knn]
    n = len(alpha) #number of different values of alpha
    r_alpha = [0 for _ in range(n)]
    sum = 0
    for i in range(eig_values_vector.shape[0]):
      sum = sum + (eig_values_vector[i]/np.sum(eig_values_vector))
      for j in range(n):
        if sum >= alpha[j]:
          if r_alpha[j] == 0:
            r_alpha[j] = i
        else:
          break
      if r_alpha[n-1] != 0:
        break
    accuracy = np.empty([n, len(k_knn)])
    #Projection
    for a in range(n):
      P = eig_vectors[:,0:r_alpha[a]]
      projected_training_matrix = np.dot(Z_training, P)
      Z_testing = testing_set - mean_vector
      projected_testing_matrix = np.dot(Z_testing, P)
      euc_dist = cdist(projected_testing_matrix, projected_training_matrix)
      for kk in range(len(k_knn)):
        classify_vector = knn(projected_testing_matrix, projected_training_matrix, n_classes, training_labels, euc_dist, k_knn[kk])
        y = np.count_nonzero(testing_labels-classify_vector)
        cm = confusion_matrix(testing_labels, classify_vector, labels=confusion_labels)
        if normal_split == True:
          images_display(test_data_size,testing_set,training_labels,classify_vector,training_set)
```

```
        else:
          for o in range(test_data_size):
            array = np.reshape(testing_set[o,:], (112,92))
            data = im.fromarray(array)
            plt.imshow(data)
            plt.axis('off')
            plt.show()
            if classify_vector[o] == 1:
              print("This is a Non-face image")
            else:
              print("This is a face image")
      accuracy[a, kk] = 1-y/test_data_size
    return accuracy, cm
```

The function performs Principal Component Analysis (PCA) on the input dataset to reduce the dimensionality of the feature space and then applies K-Nearest Neighbor (KNN) classification to identify the class of each test data point. The function also provides an option to display the results using confusion matrices and/or images.

Inputs:

- **D:** The dataset matrix
- **label_vector:** The label vector corresponding to each data point in the dataset matrix
- **alpha:** The percentage of variance to retain after PCA, can be a single value or a list of values
- **train_portion:** The portion of the dataset to be used for training, if normal_split=False
- **n_classes:** The number of classes in the dataset
- **n_perclass:** The number of samples per class in the dataset
- **k_knn:** The number of nearest neighbors to consider for classification, can be a single value or a list of values

- **normal_split:** A boolean variable indicating whether to use the normal dataset split or not

Outputs:

- **accuracy:** The accuracy of the classification for each value of alpha and k_knn
- **cm:** The confusion matrix of the classification results

Steps:

1. Splitting the input dataset into training and testing sets based on the provided parameters.
2. Computing the mean vector and centering the training set around it.
3. Calculating the covariance matrix of the centered training set.
4. Computing the eigenvalues and eigenvectors of the covariance matrix.
5. Sorting the eigenvalues and eigenvectors in descending order of eigenvalues.
6. Selecting the top r eigenvectors based on a threshold alpha value provided as input.
7. Projecting the training and testing sets onto the selected eigenvectors.
8. Computing the Euclidean distance between each testing set image and every training set image in the projected space.
9. Classifying each testing set image by performing k-NN classification using the k nearest training set images in the projected space.
10. Evaluating the classification accuracy of the model for each combination of alpha and k values provided as input.
11. Displaying the confusion matrix of the final classification results and, if requested, displaying some sample images with their true and predicted labels.
12. Returning the computed accuracy and confusion matrix as output.

3.7 'lda'

```
def lda(D,label_vector, train_portion, n_classes, n_perclass, n_eig_used, k_knn):
  # Splitting the dataset
  if train_portion == 0.5:
    training_set, testing_set, training_labels, testing_labels = split_even_odd(D, label_vector)
  else:
    training_set, testing_set, training_labels, testing_labels = split(D, label_vector, train_portion, n_classes, n_perclass)
  training_data_size = training_labels.shape[0]
  test_data_size = testing_labels.shape[0]
  n_trainperclass = int(train_portion * n_perclass)
  data_size = D.shape[0]
  feature_size = D.shape[1]
  #Mean Vector
  mean_vector = np.empty([n_classes, feature_size]) #mean vector for each class.
  for i in range(n_classes):
    mean_vector[i] = np.mean(training_set[i*n_trainperclass:(i*n_trainperclass)+n_trainperclass, :], axis=0)
  overall_mean = np.mean(mean_vector,keepdims=True, axis=0) #shape = 1x10304
  #Between class scatter matrix
  centered_means=mean_vector - overall_mean
  B = n_trainperclass*(centered_means.transpose()@centered_means) #shape = 10304x10304
  #Within class scatter matrix
  Z_training = np.empty([training_data_size, feature_size])
  for i in range(n_classes):
    Z_training[i*n_trainperclass:i*n_trainperclass+n_trainperclass, :] =
    training_set[i*n_trainperclass:i*n_trainperclass+n_trainperclass, :] - mean_vector[i]
  S = Z_training.transpose()@Z_training
  S_inv = np.linalg.inv(S)
  Matrix = S_inv@B
```

```
#Eigen Values and Eigen Vectors
eig_values, eig_vectors = np.linalg.eigh(Matrix)
indices = eig_values.argsort()[::-1]
eig_values_vector = eig_values[indices]
eig_values = np.diag(eig_values_vector)
eig_vectors = eig_vectors[:,indices]
#Projection
P_LDA = eig_vectors[:,0:n_eig_used]
Z_training_LDA = training_set - overall_mean
projected_matrix_LDA = np.dot(Z_training_LDA, P_LDA)
#Centering the testing data
Z_testing_LDA = testing_set - overall_mean
projected_testing_matrix_LDA = np.dot(Z_testing_LDA, P_LDA)
euc_dist_LDA = cdist(projected_testing_matrix_LDA, projected_matrix_LDA)
#knn
if isinstance(k_knn, (int, float)):
  k_knn=[k_knn]
accuracy = np.empty([1, len(k_knn)])
for kk in range(len(k_knn)):
  classify_vector = knn(projected_testing_matrix_LDA, projected_matrix_LDA, n_classes, training_labels, euc_dist_LDA, k_knn[kk])
  y = np.count_nonzero(testing_labels-classify_vector)
  cm = confusion_matrix(testing_labels, classify_vector, labels=confusion_labels)
  accuracy[0,kk] = 1-y/test_data_size
return accuracy, cm
```

The function lda performs Linear Discriminant Analysis (LDA) on a given dataset D with corresponding label_vector. It returns the classification accuracy and confusion matrix for the testing set.

The function has the following parameters:

- **D:** an array of size (number of data samples) x (number of features), representing the dataset
- **label_vector:** an array of size (number of data samples) x 1, representing the class labels for each data sample
- **train_portion:** a float representing the proportion of the data to use for training (default is 0.5)
- **n_classes:** an integer representing the number of classes in the dataset
- **n_perclass:** an integer representing the number of data samples per class
- **n_eig_used:** an integer representing the number of eigenvalues to use in the projection (default is all)
- **k_knn:** an integer or a list of integers representing the number of nearest neighbors to consider in the K-NN algorithm (default is 1)

Steps:

1. Splitting the dataset into training and testing sets using one of three splitting methods based on the input parameters: **split_nonfaces**, **split_even_odd**, or **split**.
2. Computing the mean vector of the training set and centering the data by subtracting the mean from each data point.
3. Computing the covariance matrix of the centered training data.
4. Computing the eigenvalues and eigenvectors of the covariance matrix.
5. Sorting the eigenvalues in descending order and selecting the eigenvectors corresponding to the **r_alpha** largest eigenvalues, where **r_alpha** is determined by the input parameter **alpha**.

6. Projecting the centered training and testing data onto the selected eigenvectors.
7. Computing the Euclidean distance between each test data point and all training data points in the projected space.
8. Classifying each test data point using k-nearest neighbors algorithm based on the input parameter **k_knn**.
9. Computing the confusion matrix and accuracy of the classification.
10. Displaying the images and classification results if the input parameter **normal_split** is **True**.
11. Returning the accuracy and confusion matrix as output.

3.8 'kernel_pca'

```
[ ]  def kernel_pca(D, label_vector, train_portion, n_classes, n_perclass, components):
        # Splitting the dataset
       if train_portion == 0.5:
         training_set, testing_set, training_labels, testing_labels = split_even_odd(D, label_vector)
       else:
         training_set, testing_set, training_labels, testing_labels = split(D, label_vector, train_portion, n_classes, n_perclass)
       training_data_size = training_labels.shape[0]
       test_data_size = testing_labels.shape[0]
       n_trainperclass = int(train_portion * n_perclass)
       #Mean Vector
       mean_vector = np.mean(training_set, axis=0)
       #Centering the data
       Z_training = training_set - mean_vector
       Z_testing = testing_set - mean_vector
       classify_vector=np.empty([test_data_size,1])
       kpca = KernelPCA(kernel='cosine', n_components = int(components))
       Z_training_kpca = kpca.fit_transform(Z_training)
       Z_testing_kpca = kpca.transform(Z_testing)
       euc_dist_KPCA = cdist(Z_testing_kpca,Z_training_kpca)
       classify_vector = knn(Z_testing_kpca, Z_training_kpca, n_classes, training_labels, euc_dist_KPCA,1)
       y = np.count_nonzero(testing_labels-classify_vector)
       print(y)
       images_display(test_data_size,testing_set,training_labels,classify_vector,training_set)
       cm = confusion_matrix(testing_labels, classify_vector, labels=confusion_labels)
       accuracy = (1-y/test_data_size)
       return accuracy, cm
```

The function **kernel_pca** performs kernel principal component analysis (PCA) and classification on a given dataset **D** and corresponding labels **label_vector**. The function first splits the dataset into training and testing sets using the **split** or **split_even_odd** functions, depending on the specified **train_portion**. The training data is then centered by subtracting the mean vector, and kernel PCA is applied to both the training and testing data using a cosine kernel and the specified number of **components**.

Next, the Euclidean distance between each testing point and every training point is computed in the kernel PCA space. For each testing point, the label of the nearest training point in the kernel PCA space is assigned to it. Finally, the classification results are compared to the true labels, and the accuracy and confusion matrix are computed and returned.

The input arguments to the function are as follows:

- **D**: the input dataset, a numpy array of shape **(n_samples, n_features)**
- **label_vector**: the corresponding labels for the input dataset, a numpy array of shape **(n_samples, 1)**

- **train_portion**: the proportion of the dataset to be used for training, either 0.5 for a 50/50 split or a decimal value between 0 and 1 for a custom split
- **n_classes**: the number of classes in the dataset
- **n_perclass**: the number of samples per class in the dataset
- **components**: the number of components to use for kernel PCA
- **voting:** two-dimensional NumPy array that is used to keep track of the number of votes and the sum of distances for each class. It has **number_of_classes** rows and 2 columns. In each iteration of the loop, the function examines the k nearest neighbors for the test data point and adds one vote and the corresponding distance to the respective class in the **voting** array. By the end of the loop, the **voting** array contains the number of votes and the sum of distances for each class among the k nearest neighbors.

Tie breaking strategy: When checking for the label of a test image, we look at the voting vector to see which label has the maximum voting. In case that 2 or more labels have the same value, and this value is the largest, some tie breaking strategy is used. The strategy is to choose the class with the smallest sum of distances to the k nearest neighbors. This is done by comparing the sum of distances for each class in the voting matrix and selecting the class with the smallest sum of distances. If there is a tie, the function chooses the class with the smallest index in the voting matrix. The function assumes that the training labels are integers starting from 1, so the class with the smallest index corresponds to the class with the smallest label.

The output of the function is a tuple containing the accuracy of the classification and the confusion matrix, respectively.

3.9 'images_display'

```
[ ]  def images_display(test_data_size,testing_set,training_labels,classify_vector,training_set):
       for o in range(test_data_size):
           fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
           array = np.reshape(testing_set[o,:], (112,92))
           data = im.fromarray(array)
           ax1.imshow(data)
           ax1.set_title('Testing Image')
           ax1.axis('off')
           indices = np.where(training_labels[:] == classify_vector[o])
           indices = np.array(indices)
           indices=indices[0,:]
           array1 = np.reshape(training_set[indices[0],:], (112,92))
           data1 = im.fromarray(array1)
           ax2.imshow(data1)
           ax2.set_title('Match')
           ax2.axis('off')
           plt.show()
```

The function images_display is used to display a side-by-side comparison of a testing image and its closest match from the training set, based on the output of a classifier. The function is called after the classification either by PCA or LDA.

The function takes in the following parameters:

- **test_data_size:** an integer representing the number of testing data samples
- testing_set: a numpy array of shape (test_data_size, num_features) containing the testing data samples
- **training_labels:** a numpy array of shape (num_training_samples,) containing the labels of the training data samples
- **classify_vector:** a numpy array of shape (test_data_size,) containing the predicted labels of the testing data samples
- **training_set:** a numpy array of shape (num_training_samples, num_features) containing the training data samples

For each testing image, the function creates a new matplotlib figure with two subplots, one showing the testing image and another showing its closest match from the training set. The closest match is identified by finding the indices of the training samples that have the same label as the predicted label of the testing sample, and then selecting the first index from that list. Both the testing image and the match are reshaped from a flat vector of length num_features to a 2D array of shape (112, 92) before being displayed.

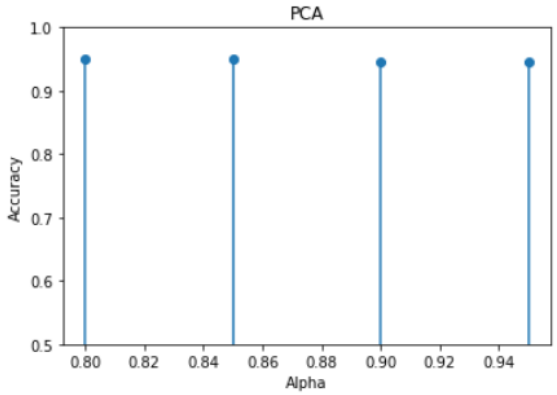
Finally, the function displays the figure using plt.show().

4. Sample Runs

4.1 PCA with different values of alpha

```
[16] alpha = [0.8, 0.85, 0.9, 0.95]
     training_portion = 0.5
     k = 1
     if isinstance(k, int):
         k=[k]
     if isinstance(alpha, (int,float)):
         alpha=[alpha]
     accuracy, cm = pca(data_matrix, label_vector, alpha, training_portion, number_of_classes, n_perclass, k, True)
     for i in range(len(alpha)):
       for j in range(len(k)):
         print("\tFor alpha = {}, the accuracy is {} % \n".format(alpha[i], accuracy[i,j]*100))

     plt.stem(alpha, accuracy[:,0])
     plt.xlabel('Alpha')
     plt.ylabel('Accuracy')
     plt.ylim(0.5, 1)
     plt.title('PCA')
     plt.show()
     print("The confusion matrix is: \n", cm)
```

```
        For alpha = 0.8, the accuracy is 95.0 %

        For alpha = 0.85, the accuracy is 95.0 %

        For alpha = 0.9, the accuracy is 94.5 %

        For alpha = 0.95, the accuracy is 94.5 %
```




```
The confusion matrix is:
 [[5 0 0 ... 0 0 0]
 [0 5 0 ... 0 0 0]
 [0 0 5 ... 0 0 0]
 ...
 [0 0 0 ... 5 0 0]
 [0 0 0 ... 0 4 0]
 [0 0 0 ... 0 0 4]]
```

Notes:

- If the dataset was uploaded in different order other training faces will be used for each class, as a result the accuracy will change.

- Increasing the value of alpha does not always improve the performance of the program, sometimes overfitting may occur. Overfitting means that the program trains more than needed and starts to memorize the pattern instead of learning it, leading to inaccuracy for generalization.

4.2 Kernel PCA

```
#K = 1
training_portion = 0.5
number_of_components = 10
accuracy, cm = kernel_pca(data_matrix, label_vector, training_portion, number_of_classes, n_perclass, number_of_components)
print("The accuracy is: {} %". format(accuracy * 100))
print("The confusion matrix is: \n", cm)
print(cm.shape)
```

```
The accuracy is: 90.0 %
The confusion matrix is:
 [[5 0 0 ... 0 0 0]
 [0 5 0 ... 0 0 0]
 [0 0 5 ... 0 0 0]
 ...
 [0 0 0 ... 4 0 0]
 [0 0 0 ... 0 5 0]
 [0 0 0 ... 0 0 4]]
(40, 40)
```

4.3 Kernel PCA vs PCA

Code:

```
training_portion = 0.5
number_of_components = 10
explained_variance = 0.95
k = 1

start_time = time.perf_counter()
accuracy_kpca, cm_kpca = kernel_pca(data_matrix, label_vector, training_portion, number_of_classes, n_perclass, number_of_components)
end_time = time.perf_counter()
execution_time_kpca = end_time - start_time
print("Kernel PCA execution time: {:.5f} seconds".format(execution_time_kpca))
print("The accuracy obtained is: ", accuracy_kpca * 100)
print("The confusion matrix is: \n", cm_kpca)

start_time = time.perf_counter()
accuracy, cm = pca(data_matrix, label_vector, explained_variance, training_portion, number_of_classes, n_perclass, k,True)
end_time = time.perf_counter()
execution_time = end_time - start_time
print("\n\nPCA execution time: {:.5f} seconds".format(execution_time))
print("The accuracy received is:  {} %". format(accuracy[0,0] * 100))
print("The confusion matrix is: \n", cm)
```

Output:

```
Kernel PCA execution time: 0.38061 seconds
The accuracy obtained is:  90.0
The confusion matrix is:
 [[5 0 0 ... 0 0 0]
 [0 5 0 ... 0 0 0]
 [0 0 5 ... 0 0 0]
 ...
 [0 0 0 ... 4 0 0]
 [0 0 0 ... 0 5 0]
 [0 0 0 ... 0 0 4]]


PCA execution time: 292.24397 seconds
The accuracy received is:  94.5 %
The confusion matrix is:
 [[5 0 0 ... 0 0 0]
 [0 5 0 ... 0 0 0]
 [0 0 5 ... 0 0 0]
 ...
 [0 0 0 ... 5 0 0]
 [0 0 0 ... 0 4 0]
 [0 0 0 ... 0 0 4]]
```

Notes:

KernelPCA built-in function is shown to be running much faster than a manual implementation of PCA due to a combination of optimization, preprocessing, memory management that the developers of the function managed to do.

4.4 LDA

```
train_portion = 0.5
n_classes = 40
n_perclass = 10
n_eig_used = 39
k = 1
accuracy, cm = lda(data_matrix, label_vector, train_portion, n_classes, n_perclass, n_eig_used, k, True)
if isinstance(k, int):
    k=[k]
for i in range(len(k)):
  print("\tFor k = {}, the accuracy is {} % \n".format(k[i], accuracy[0, i]*100))
print("The confusion matrix is: \n", cm)
```

```
        For k = 1, the accuracy is 94.5 %

The confusion matrix is:
 [[4 0 0 ... 0 0 0]
 [0 5 0 ... 0 0 0]
 [0 0 5 ... 0 0 0]
 ...
 [0 0 0 ... 3 0 0]
 [0 0 0 ... 0 5 0]
 [0 0 0 ... 0 0 5]]
```

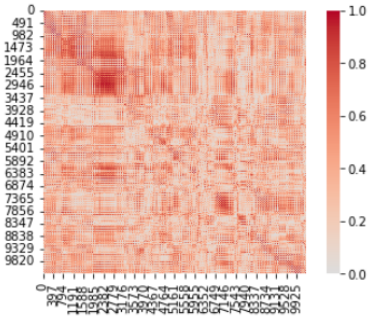
In the following figures we can see that some of the faces that were detected correctly and last one was not classified correctly due to some similarities in the features of both faces.

4.5 QDA

```
training_portion = 0.5
heatmap_bool = True
accuracy, cm = qda(data_matrix, label_vector, training_portion, number_of_classes, n_perclass, reg_param, heatmap_bool)
print("The accuracy is: {} %". format(accuracy * 100))
print("The confusion matrix is: \n", cm)
```

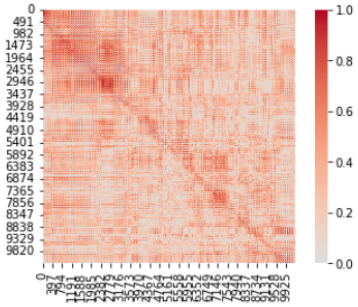
Collinearity in the data is disadvantageous when using the Quadratic Discriminant Analysis. As the face images are highly correlated, the QDA showed a very low accuracy when training and testing on the dataset of faces only. This can be visualized by the following heatmap:

```
/usr/local/lib/python3.9/dist-packages/sklearn/discriminant_analysis.py:926: UserWarning: Variables are collinear
  warnings.warn("Variables are collinear")
The accuracy is: 6.000000000000005 %
The confusion matrix is:
 [[0 0 0 ... 0 0 0]
 [2 0 0 ... 0 0 0]
 [0 0 1 ... 0 0 1]
 ...
 [0 0 0 ... 0 0 0]
 [0 1 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
```



On the other hand, when the dataset was of faces and non-faces, a significant increase in the accuracy occurred. An interpretation for that, is that the non-face images are totally random (ex: airplanes, flowers, etc..), thus their features have a considerably low covariances with respect to features of other objects. A visualization for the covariances of testing samples from the mixed dataset is as follow:

```
The accuracy is: 81.5 %
The confusion matrix is:
 [[144  56]
 [ 18 182]]
```



4.6 PCA and LDA (different splitting ratio)

```
#PCA
alpha = [0.95]
k = [1]
print("for the 70 - 30 split:\n")
training_portion = 0.7
accuracy, cm = pca(data_matrix, label_vector, alpha, training_portion, number_of_classes, n_perclass, k,True)
print("PCA: for alpha = {}, for k = {}, the accuracy is {} % \n".format(alpha[0], k[0], accuracy[0,0]*100))
#LDA
accuracy, cm = lda(data_matrix, label_vector, training_portion, number_of_classes, n_perclass, n_eig_used, k, True)
print("LDA: for k = {}, the accuracy is {} % \n".format(k[0], accuracy[0, 0]*100))
```

```
print("for the 20 - 80 split:\n")
training_portion = 0.2
accuracy, cm = pca(data_matrix, label_vector, alpha, training_portion, number_of_classes, n_perclass, k,True)
print("PCA: for alpha = {}, for k = {}, the accuracy is {} % \n".format(alpha[0], k[0], accuracy[0,0]*100))
#LDA
accuracy, cm = lda(data_matrix, label_vector, training_portion, number_of_classes, n_perclass, n_eig_used, k, True)
print("LDA: for k = {}, the accuracy is {} % \n".format(k[0], accuracy[0, 0]*100))
```

Outputs

```
for the 70 - 30 split:

PCA: for alpha = 0.95, for k = 1, the accuracy is 96.66666666666667 %

LDA: for k = 1, the accuracy is 97.5 %
```

```
for the 20 - 80 split:

PCA: for alpha = 0.95, for k = 1, the accuracy is 83.125 %

LDA: for k = 1, the accuracy is 84.375 %
```

Comments: As we can see, when the number of faces used for training increases, the accuracy of the program increases as well.

From section (4.1) the accuracy of the PCA was 94.5%, but when the ratio of training data becomes 0.7 for all the set of training, the accuracy increases to 96.67%. Also, it decreases as well when the training ratio became 0.2 to 83.125%.

4.7 QDA vs LDA

```
training_portion = 0.5
k = 1
heatmap_bool = False

start_time = time.perf_counter()
accuracy_qda, cm_qda = qda(data_matrix, label_vector, training_portion, number_of_classes, n_perclass, reg_param, heatmap_bool)
end_time = time.perf_counter()
execution_time_qda = end_time - start_time
print("\n\nQDA execution time: {:.5f} seconds".format(execution_time_qda))
print("The accuracy is: {} %". format(accuracy_qda * 100))
print("The confusion matrix is: \n", cm_qda)


start_time = time.perf_counter()
accuracy, cm = lda(data_matrix, label_vector, training_portion, number_of_classes, n_perclass, n_eig_used,k, True)
end_time = time.perf_counter()
execution_time = end_time - start_time
print("\n\nLDA execution time: {:.5f} seconds".format(execution_time))
print("The accuracy is:  {} %". format(accuracy[0,0] * 100))
print("The confusion matrix is: \n", cm)
```

```
QDA execution time: 1.78057 seconds
The accuracy is: 6.000000000000005 %
The confusion matrix is:
 [[0 0 0 ... 0 0 0]
 [2 0 0 ... 0 0 0]
 [0 0 1 ... 0 0 1]
 ...
 [0 0 0 ... 0 0 0]
 [0 1 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]


LDA execution time: 459.25362 seconds
The accuracy is:  94.5 %
The confusion matrix is:
 [[4 0 0 ... 0 0 0]
 [0 5 0 ... 0 0 0]
 [0 0 5 ... 0 0 0]
 ...
 [0 0 0 ... 3 0 0]
 [0 0 0 ... 0 5 0]
 [0 0 0 ... 0 0 5]]
```

Notes:

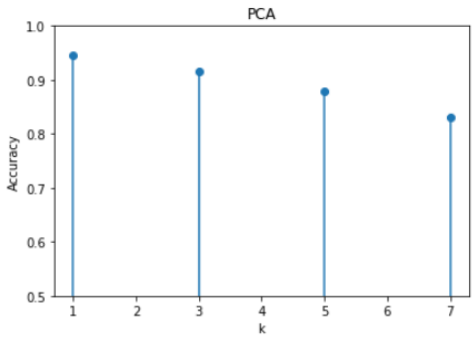
QDA built-in function is shown to be running much faster than a manual implementation of LDA due to a combination of optimization, preprocessing, memory management that the developers of the function managed to do.

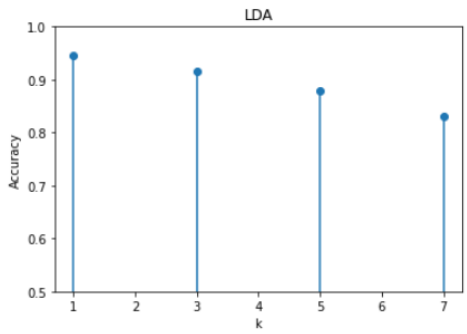
4.8 PCA vs LDA (different values for k)

```
alpha = [0.95]
training_portion = 0.5
k = [1, 3, 5, 7]
print('PCA: \n')
accuracy, cm = pca(data_matrix, label_vector, alpha, training_portion, number_of_classes, n_perclass, k, True)
for i in range(len(k)):
  print("For k = {}, the accuracy is {} % \n".format(k[i],  accuracy[0,i]*100))
plt.stem(k, accuracy[0,:])
plt.xlabel('k')
plt.ylabel('Accuracy')
plt.ylim(0.5, 1)
plt.title('PCA')
plt.show()


accuracy2, cm2 = lda(data_matrix, label_vector, training_portion, number_of_classes, n_perclass, n_eig_used, k, True)
print('LDA: \n')
for i in range(len(k)):
  print("\tFor k = {}, the accuracy is {} % \n".format(k[i], accuracy[0, i]*100))
plt.stem(k, accuracy[0,:])
plt.xlabel('k')
plt.ylabel('Accuracy')
plt.ylim(0.5, 1)
plt.title('LDA')
plt.show()
```

```
PCA:

For k = 1, the accuracy is 94.5 %

For k = 3, the accuracy is 91.5 %

For k = 5, the accuracy is 88.0 %

For k = 7, the accuracy is 83.0 %
```

```
LDA:

        For k = 1, the accuracy is 94.5 %

        For k = 3, the accuracy is 91.5 %

        For k = 5, the accuracy is 88.0 %

        For k = 7, the accuracy is 83.0 %
```





Comments: This is a result of 1 main factor, which is:

- Noise: In some cases, increasing k may lead to a decrease in accuracy due to noise in the data. As k increases, the model may become more susceptible to noise and outliers, which can negatively impact its performance.

In conclusion, the optimal value of k will depend on the specific dataset and problem at hand. It's important to experiment with different values of k and evaluate the performance using appropriate metrics to find the best value for your particular use case.

4.9 Faces vs Non-faces Classification

4.9.1 PCA

```
k_knn=1
alpha = 0.95
training_portion=0.5
accuracy = pca(data_matrix, label_vector, alpha, training_portion, 2, 400, k_knn, False)
print("The accuracy received is:  {} %". format(accuracy[0,0] * 100))
```

This is a face image

This is a face image

This is a face image

```
The accuracy received is:  93.0 %
Confusion Matrix:
 [[172  28]
 [  0 200]]
```

The confusion matrix means that there are 200 faces correctly classified as faces. However, from the 200 non face, only 172 were correctly classified as non faces.

4.9.2 LDA

```
n_eig_used=2
train_portion=0.5
accuracy, cm = lda(data_matrix,label_vector, train_portion, number_of_classes, n_perclass, n_eig_used, 1, True)
print("The accuracy received is:  {} %". format(accuracy[0,0] * 100))
print("The confusion matrix:\n", cm)
```

```
The accuracy received is:  88.25 %
The confusion matrix:
 [[176  24]
 [ 23 177]]
```

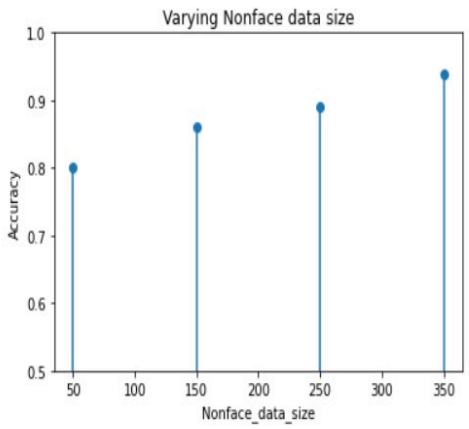
4.9.3 Different training size

```
[ ]  #Loop faces and nonfaces with variable nonfaces
     #100, 200, 300 and 400 nonface images in the dataset
     #50-50 split to be used with k=1 only
     nonface_data_size= [100, 200, 300, 400]
     # at each time testing samples are fixed to 50 while traning samples changes through 50,100,150,200
     explained_variance = 0.95
     training_portion = 0.5
     k = 1
     shuffled_data_matrix=np.empty([2*n_perclass , 10304])
     shuffled_data_matrix[0:n_perclass,:] = np.random.permutation(data_matrix[0:n_perclass,:])
     shuffled_data_matrix[n_perclass:,:] = np.random.permutation(data_matrix[n_perclass :,:])
     print(shuffled_data_matrix.shape)
     accuracy = [0.0, 0.0, 0.0, 0.0]
     for i in range(len(nonface_data_size)):
       #Initialisation of new data matrix and label vector
       new_sample_size = n_perclass + nonface_data_size[i]
       new_data_matrix=np.empty([new_sample_size , 10304])
       new_label_vector=np.empty([new_sample_size,1])
       #Data and labels splitting
       new_data_matrix[0:nonface_data_size[i],:]=shuffled_data_matrix[0:nonface_data_size[i],:]
       new_label_vector[0:nonface_data_size[i],:]=label_vector[0:nonface_data_size[i],:]
       new_data_matrix[nonface_data_size[i]:n_perclass+nonface_data_size[i],:]=shuffled_data_matrix[n_perclass:data_size,:]
       new_label_vector[nonface_data_size[i]:n_perclass+nonface_data_size[i],:]=label_vector[n_perclass:data_size,:]
       accuracy_received, cm = pca(new_data_matrix, new_label_vector, explained_variance, training_portion, number_of_classes, n_perclass, k, False)
       accuracy[i] = accuracy_received[0,0]
     print("Number of faces is fixed to 400 samples\n")
     print("Number of non-faces training: ")
     for i in range(len(nonface_data_size)):
           print("\tFor {} non face training samples, the accuracy is {} % \n".format(nonface_data_size[i]-50*(i+1), accuracy[i]*100))
     print("Note: Testing samples are fixed to 50\n")
     nonface_data_size = nonface_data_size-50
     plt.stem(nonface_data_size, accuracy)
     plt.xlabel('Nonface_data_size')
     plt.ylabel('Accuracy')
     plt.ylim(0.5, 1)
     plt.title('Varying Nonface data size')
     plt.show()
```

In this part of code, the dataset of faces and non-faces is used. This dataset is an 800x10304 matrix containing first the non-faces then the faces. To compare the performance of the program, we fix the size of testing set, only 50 faces and 50 non-faces will be used. For training, always 200 faces are used. However, for the non faces the program will run 4 times while changing the size of the non-faces (50, 100, 150, 200). It is shown in the output that the program succeeds in classifying the faces, because they are very related to each other. Increasing the number of non-faces in training, improve the performance of the program in detecting that an image is a non-face.

This can be shown in the following outputs:

```
Number of faces is fixed to 400 samples

Number of non-faces training:
        For 50 non face training samples, the accuracy is 80.0 %

        For 100 non face training samples, the accuracy is 86.0 %

        For 150 non face training samples, the accuracy is 89.0 %

        For 200 non face training samples, the accuracy is 94.0 %

Note: Testing samples are fixed to 50
```



To more understand the performance let's take an example.

This image was classified as a face when the number of non-faces used in training was 50, 100, 150. But when the size becomes 200, the program finally correctly classifies it as non-face.

Figure 1: non-faces = 50, 100, 150

Figure 2: non-faces 200

Another example, this image was classified as a face for the size 50 and 100, and for the size 150 it is correctly classified as non-face.

Figure 3: non-faces = 50, 100

Figure 4: non-faces = 150, 200

It must be noted that, for the all the four values of the size of the non-faces used for training, the program succeeds in detecting the faces with accuracy 100%