Alexandria University
Faculty of engineering
Computer and Systems Engineering Department
CSE-352: Pattern Recognition

Assignment #1

Face Recognition

Team members:

Name	ID
Ahmed Ashour El-Sayed Mohammed	19015267
Abdelrahman Adel AbdelFattah	17012296
Mayar Ayman Mahmoud Elkhouly	19016744

The Implementation Steps

The Google colab code is here ⇒

First, we imported the needed libraries and defined some constants:

```python
# Imports
from sklearn.neighbors import KNeighborsClassifier
from google.colab.patches import cv2_imshow
import matplotlib.pyplot as plt
from skimage.io import imread
from numpy import array
import numpy as np
import cv2
import csv
# Conatants & Arrays
SAMPLES = 400
SUBJECTS = 40
INSTANCES = 10
K = array([1, 3, 5, 7])
ALPHA = array([0.8, 0.85, 0.9, 0.95])
# Debugging-purpose functions
def printWithTitle(title, A):
    print(f'\033[1m{title}:\033[0m'); print(A, end = "\n\n")

def printTitle(title, end = '\n'):
    print(f'\033[1m{title}:\033[0m', end = end)
```

Then, we implemented every step separately as follows:

1. Downloading the Dataset & Understanding the Format

- The ORL dataset of faces has 10 images per 40 subjects (dataset of 40-person faces) organized in 40 folders named with 's{i}' where: i = 1, 2, 3, …, 40.
- Every image is a grayscale image of size 92x112 (**10304 pixels**) and has a **.pgm** extension.
- Every pixel corresponds to a **BYTE** (8 bits), so it holds an integer from 0 to 255.
- The following code extracts the zipped file & shows all images:

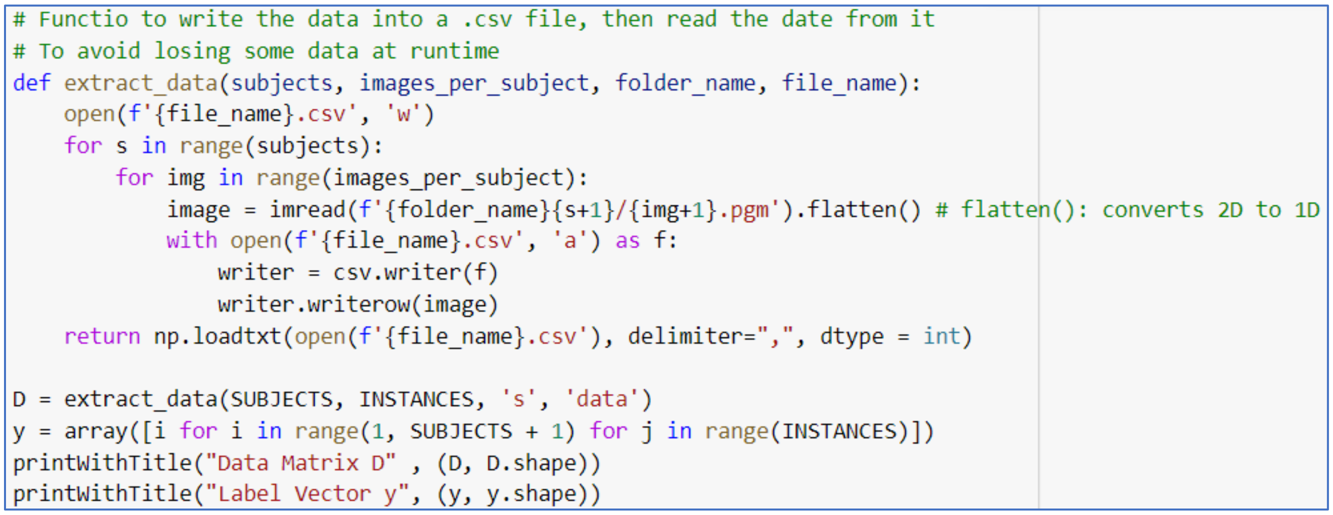
```python
# Uplaod the zip file, then run this code to unzip it to run remaining cells with no issues
from zipfile import ZipFile
file_name = 'archive.zip'

with ZipFile(file_name, 'r') as zip:
    zip.extractall()
```

```
# Done ^-^
# Showing all 400 images
for s in range(SUBJECTS): # 40 subjects
    images = array([[[0] * 92] * 112] * INSTANCES)
    for img in range(INSTANCES): # 10 images per subject
        images[img] = cv2.imread(f's{s+1}/{img+1}.pgm', cv2.IMREAD_UNCHANGED)
        person = np.concatenate(images, axis=1)
    cv2_imshow(person)
```

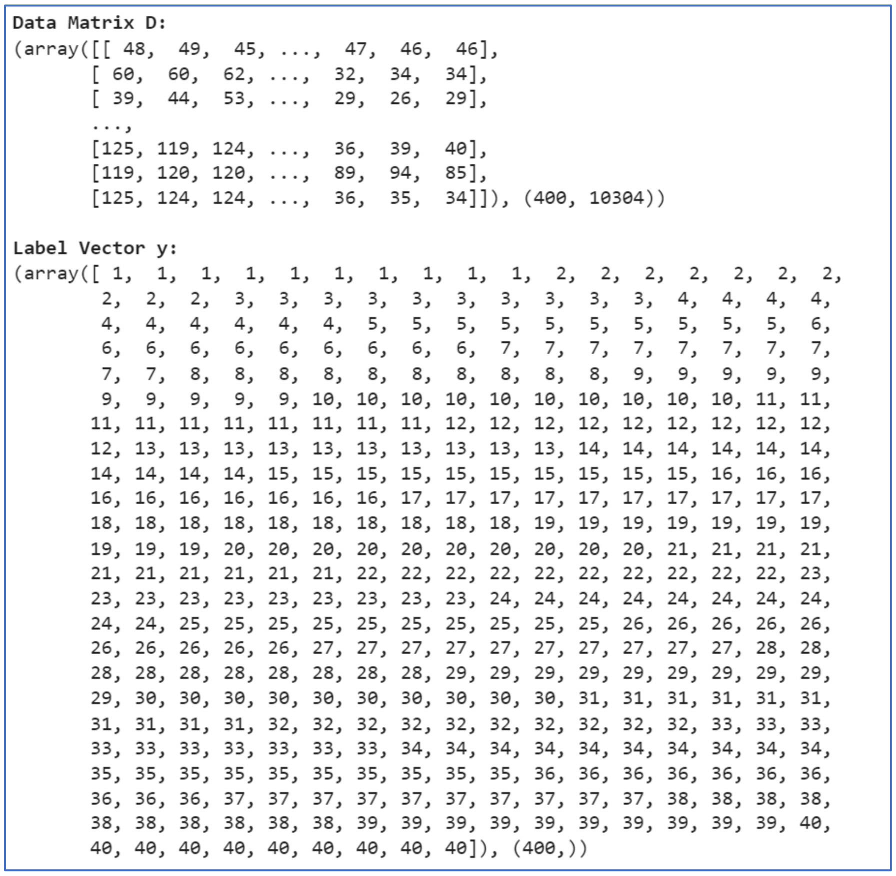
- Subset of the images showed:

2. Generating the Data Matrix and Label vector

- Defined function **`extract_data()`** to generate the Data Matrix D, which saves the 10304 values of each image in an excel file (.csv) and loads all values at once into a matrix.
- The label vector y generated using simple 2 nested for loops.



```python
# Functio to write the data into a .csv file, then read the date from it
# To avoid losing some data at runtime
def extract_data(subjects, images_per_subject, folder_name, file_name):
    open(f'{file_name}.csv', 'w')
    for s in range(subjects):
        for img in range(images_per_subject):
            image = imread(f'{folder_name}{s+1}/{img+1}.pgm').flatten() # flatten(): converts 2D to 1D
            with open(f'{file_name}.csv', 'a') as f:
                writer = csv.writer(f)
                writer.writerow(image)
    return np.loadtxt(open(f'{file_name}.csv'), delimiter=",", dtype = int)

D = extract_data(SUBJECTS, INSTANCES, 's', 'data')
y = array([i for i in range(1, SUBJECTS + 1) for j in range(INSTANCES)])
printWithTitle("Data Matrix D" , (D, D.shape))
printWithTitle("Label Vector y", (y, y.shape))
```



```
Data Matrix D:
(array([[ 48,  49,  45, ...,  47,  46,  46],
       [ 60,  60,  62, ...,  32,  34,  34],
       [ 39,  44,  53, ...,  29,  26,  29],
       ...,
       [125, 119, 124, ...,  36,  39,  40],
       [119, 120, 120, ...,  89,  94,  85],
       [125, 124, 124, ...,  36,  35,  34]]), (400, 10304))

Label Vector y:
(array([ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,
        2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  4,  4,  4,  4,
        4,  4,  4,  4,  4,  4,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  6,
        6,  6,  6,  6,  6,  6,  6,  6,  6,  7,  7,  7,  7,  7,  7,  7,  7,
        7,  7,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,
        9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11,
       11, 11, 11, 11, 11, 11, 11, 11, 12, 12, 12, 12, 12, 12, 12, 12, 12,
       12, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 14, 14, 14, 14, 14, 14,
       14, 14, 14, 14, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 16, 16, 16,
       16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17,
       18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19,
       19, 19, 19, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 21, 21, 21, 21,
       21, 21, 21, 21, 21, 21, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 23,
       23, 23, 23, 23, 23, 23, 23, 23, 23, 24, 24, 24, 24, 24, 24, 24, 24,
       24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26,
       26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 28, 28,
       28, 28, 28, 28, 28, 28, 28, 28, 29, 29, 29, 29, 29, 29, 29, 29, 29,
       29, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 31, 31, 31, 31, 31, 31,
       31, 31, 31, 31, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 33, 33, 33,
       33, 33, 33, 33, 33, 33, 33, 34, 34, 34, 34, 34, 34, 34, 34, 34, 34,
       35, 35, 35, 35, 35, 35, 35, 35, 35, 35, 36, 36, 36, 36, 36, 36, 36,
       36, 36, 36, 37, 37, 37, 37, 37, 37, 37, 37, 37, 37, 38, 38, 38, 38,
       38, 38, 38, 38, 38, 38, 39, 39, 39, 39, 39, 39, 39, 39, 39, 39, 40,
       40, 40, 40, 40, 40, 40, 40, 40, 40]), (400,))
```

3. Splitting the Dataset into Training and Test sets

- The Data Matrix D is split 50%–50% into two sets:
  - **Training set** that contains the **odd** rows of D.
  - **Test set** that contains the **even** rows of D.
- The label vector y is also split accordingly.

```
(train_D, train_y) = (D[0 : SAMPLES : 2], y[0 : SAMPLES : 2]) # Training Data & labels
(test_D , test_y ) = (D[1 : SAMPLES : 2], y[1 : SAMPLES : 2]) # Testing  Data & labels
printWithTitle("Training Data", train_D); printWithTitle("Training Labels", train_y)
printWithTitle("Testing Data" , test_D ); printWithTitle("Testing Labels" , test_y )
```

```
Training Data:
[[ 48  49  45 ...  47  46  46]
 [ 39  44  53 ...  29  26  29]
 [ 64  76  80 ...  35  37  39]
 ...
 [128 125 125 ...  85  90  84]
 [129 127 133 ...  93  93  93]
 [119 120 120 ...  89  94  85]]

Training Labels:
[ 1  1  1  1  1  2  2  2  2  2  3  3  3  3  3  4  4  4  4  4  5  5  5  5
  5  6  6  6  6  6  7  7  7  7  7  8  8  8  8  8  9  9  9  9  9 10 10 10
 10 10 11 11 11 11 11 12 12 12 12 12 13 13 13 13 13 14 14 14 14 14 15 15
 15 15 15 16 16 16 16 16 17 17 17 17 17 18 18 18 18 18 19 19 19 19 19 20
 20 20 20 20 21 21 21 21 21 22 22 22 22 22 23 23 23 23 23 24 24 24 24 24
 25 25 25 25 25 26 26 26 26 26 27 27 27 27 27 28 28 28 28 28 29 29 29 29
 29 30 30 30 30 30 31 31 31 31 31 32 32 32 32 32 33 33 33 33 33 34 34 34
 34 34 35 35 35 35 35 36 36 36 36 36 37 37 37 37 37 38 38 38 38 38 39 39
 39 39 39 40 40 40 40 40]
```

```
Testing Data:
[[ 60  60  62 ...  32  34  34]
 [ 63  53  35 ...  41  10  24]
 [ 43  50  41 ... 158 153 169]
 ...
 [123 121 126 ...  40  35  42]
 [125 119 124 ...  36  39  40]
 [125 124 124 ...  36  35  34]]

Testing Labels:
[ 1  1  1  1  1  2  2  2  2  2  3  3  3  3  3  4  4  4  4  4  5  5  5  5
  5  6  6  6  6  6  7  7  7  7  7  8  8  8  8  8  9  9  9  9  9 10 10 10
 10 10 11 11 11 11 11 12 12 12 12 12 13 13 13 13 13 14 14 14 14 14 15 15
 15 15 15 16 16 16 16 16 17 17 17 17 17 18 18 18 18 18 19 19 19 19 19 20
 20 20 20 20 21 21 21 21 21 22 22 22 22 22 23 23 23 23 23 24 24 24 24 24
 25 25 25 25 25 26 26 26 26 26 27 27 27 27 27 28 28 28 28 28 29 29 29 29
 29 30 30 30 30 30 31 31 31 31 31 32 32 32 32 32 33 33 33 33 33 34 34 34
 34 34 35 35 35 35 35 36 36 36 36 36 37 37 37 37 37 38 38 38 38 38 39 39
 39 39 39 40 40 40 40 40]
```

4. PCA Classification

- Implementation of the Principle Component Analysis (PCA) Algorithm:
  - Function `PCA_eigh()` calculates the **eigenvalues** and **eigenvectors** sorted according to the eigenvalues.

```
def PCA_eigh(D):
    mean = np.mean(D, axis = 0)
    Z = np.matrix(D - mean)
    COV = (Z.T * Z) / Z.shape[0]
    lamdas, U = np.linalg.eigh(COV)
    # Sorting λ's & U descendingly according to values of λ's
    index_array = np.argsort(lamdas, axis=0)[::-1]
    eigenvalues = lamdas[index_array]
    eigenvectors = U[:, index_array]
    return eigenvalues, eigenvectors
```

  - Function `PCA_projMat()` computes **the projection matrix U** depending on the alpha (α) value passed.

```
def PCA_projMat(eigenvalues, eigenvectors, alpha):
    r, sum, fraction = 0, 0, 0
    eigenvalues_sum = eigenvalues.sum()
    for i in range(len(eigenvalues)):
        sum += eigenvalues[i]
        fraction = sum / eigenvalues_sum
        if (fraction >= alpha):
            r = i + 1; break
    return eigenvectors[:, 0 : r]
```

  - Function `data_projection()` multiplies Data Matrix D by Projection Matrix U (D×U) which produces **the projected data**.
  - Function `data_recovering()` does the inverse of the previous one.

```
def data_projection(D, U):
    return np.dot(D, U)

def data_recovering(A, U):
    return np.dot(A, U.T)
```

- Both Training and Test data are projected separately using the same projection matrix U each time with a different α value.
- The *first Nearest Neighbor classifier* is used to classify them.
  - Function KNN() represents **the K-Nearest-Neighbors Classifier**. It sets the number of nearest neighbors (k), then trains itself using the Training data, and finally predicts the test data & calculates its score.

```
def KNN(k, train, train_labels, test, test_labels):
    knn = KNeighborsClassifier(n_neighbors = k)
    knn.fit(np.asarray(train), train_labels)
    y_predicted = knn.predict(np.asarray(test))
    return y_predicted, float(knn.score(np.asarray(test), test_labels))
```

- The Accuracy for every value of α, using the **First Nearest Neighbor** classifier (1-NN):

```
eigenvalues, eigenvectors = PCA_eigh(train_D)
PCA_accuracy = array([0.0] * len(ALPHA))
printTitle('PCA-Accuracy using 1-NN')

# for each alpha value
for a in range(len(ALPHA)):
    print(f'\tAt alpha = {ALPHA[a]}', end = '');
    print(' ' if a%2==0 else '', end = '')

    # Projection Matrix
    U = PCA_projMat(eigenvalues, eigenvectors, ALPHA[a])

    # Projecting both Training & Test data
    train_projected = data_projection(train_D, U)
    test_projected = data_projection(test_D, U)

    # Classify the projected data using 1-NN Classifier
    test_predicted, PCA_accuracy[a] = KNN(1, train_projected, train_y, test_projected, test_y)
    print(', accuracy =', PCA_accuracy[a])
```

```
PCA-Accuracy using 1-NN:
        At alpha = 0.8 , accuracy = 0.95
        At alpha = 0.85, accuracy = 0.95
        At alpha = 0.9 , accuracy = 0.94
        At alpha = 0.95, accuracy = 0.94
```

⇒ The Accuracy seems to be maximum for an interval of alpha values and going far from this interval (above or below) will affect the accuracy to decrease by different amounts.

5. LDA Classification

1) Function LDA (train_D, instances , classes):

It takes the Training data, number of instances, and number of classes as parameters.

- First, we calculate the overall mean of the data.

```
overall_mean = np.mean(train_D, axis = 0)
overall_mean = overall_mean.reshape(10304, 1)
printWithTitle("overALL_mean",overall_mean)
```

- Then we split the data into its original classes (40) and every class is 5 instances and put them in classes_data string

```
classes_data = []
c=0
for i in range(classes):
  data=train_D[c : c+instances]
  classes_data.append(data)
  c += instances
printWithTitle("classes_data dim ",np.array(classes_data).shape)
```

- Then we calculate the mean of each class of these classes and put them in classes_mean string

```
classes_means = []
for i in range(classes):
    mean = np.mean(classes_data[i], axis = 0)
    classes_means.append(mean.reshape(10304,1))
printWithTitle("classes_means dim ",np.array(classes_means).shape)
printWithTitle("means for each class ",np.array(classes_means))
```

- Then we calculate the Sb by using its rule in the lab

```
#finding Sb with size 10304 * 10304
Sb= np.zeros((10304, 10304))
for i in range(classes):
    means_diffrence = classes_means[i] - overall_mean
    temp =  np.dot(instances, np.dot( means_diffrence, means_diffrence.T))
    Sb = np.add(Sb,temp)
```

- Then we calculate S and Z

```
s = np.zeros((10304, 10304))
for i in range(classes):
  z = classes_data[i] - classes_means[i].T
  s = np.add(s, np.dot(z.T, z))  #sum from s1 to s40
```

- Finally, we calculate the eigenvalues and eigenvectors after calculating S_inverse * Sb

```
s_inverse = np.linalg.inv(s)
s_invB=np.dot(s_inverse, Sb)
eigen_values, eigen_vectors = np.linalg.eigh(s_invB)
```

2) Function projection(train_D,test_D,instances, classes,eigen_values,eigen_vectors):

It takes the Training and Test data, number of instances, number of classes, eigenvalues, and eigenvectors as parameters.

- First, we sort the eigenvalues and corresponding eigenvectors to choose the largest 39 of them

```
#sorting the eigen value and eigen vector
 idx=eigen_values.argsort()[::-1]
 eigen_values=eigen_values[idx]
 eigen_vectors=eigen_vectors[:,idx]
```

- The projection matrix will have the eigenvectors corresponding to the largest 39 eigenvalues.

```
projection_mat= eigen_vectors[:,:classes-1]
```

- The data after projection:

```
# data after projection
projected_train = np.dot(train_D, projection_mat)
#printWithTitle("projected train data dim",projected_train.shape)
projected_test = np.dot(test_D, projection_mat)
#printWithTitle("projected test data dim",projected_test.shape)
```

- Finally, we apply the 1-NN classifier and test the accuracy: it was **0.95**.

```
test_predicted, LDA_accuracy = KNN(1, p_train, train_y, p_test, test_y)
printWithTitle("n-neighbours",test_predicted)
printWithTitle("LDA Accuracy",LDA_accuracy)
```

6. Classifier Tuning

Tie Breaking strategy:

When the K-NN are tied we use the closest neighbor as a tiebreaker, the closer the neighbor, the more likely it is to be classified as the same class.

```python
def KNN_weighted(k, train, train_labels, test, test_labels):
    knn = KNeighborsClassifier(n_neighbors = k, weights='distance')
    knn.fit(np.asarray(train), train_labels)
    y_predicted = knn.predict(np.asarray(test))
    return y_predicted, float(knn.score(np.asarray(test), test_labels))
```

Accuracy Table:

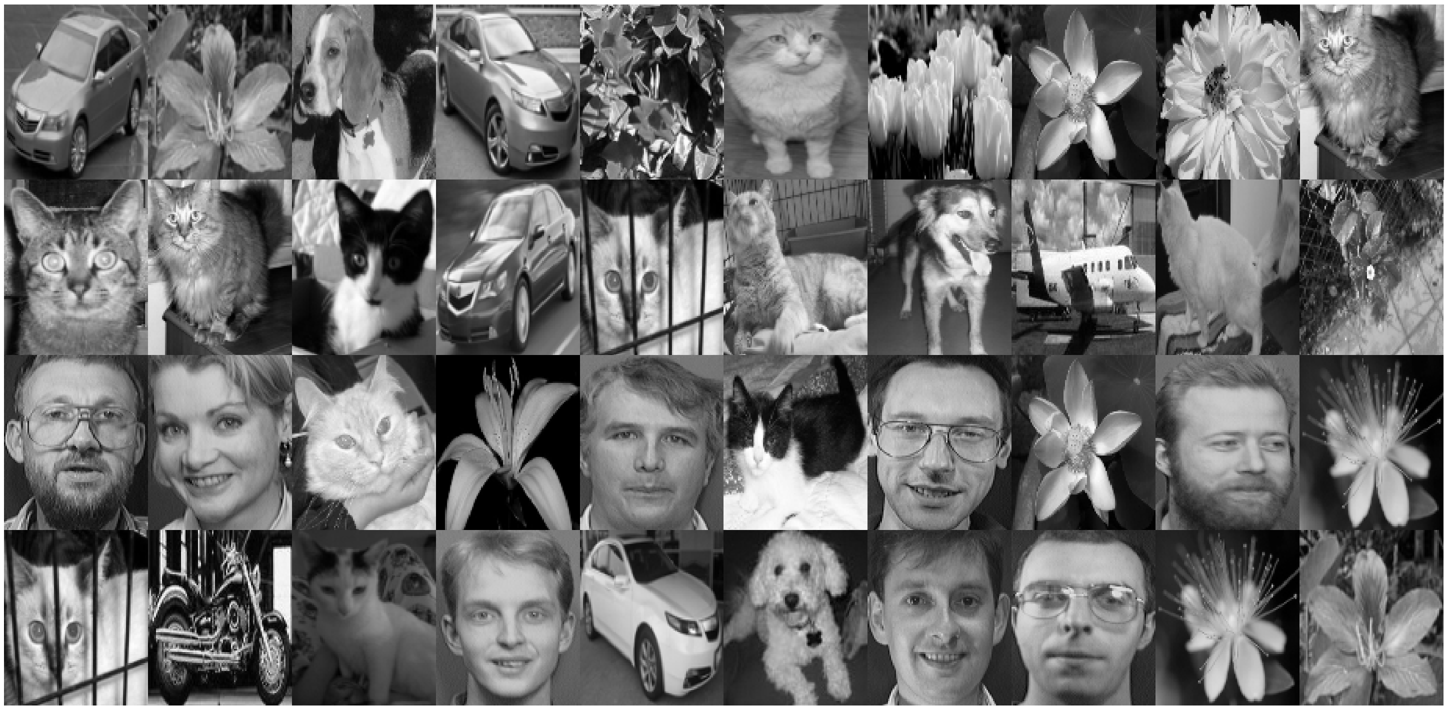
K	PCA (mean of different alphas)	LDA
1	0.945	0.95
3	0.9275	0.92
5	0.91125	0.885
7	0.87875	0.88

7. Comparing vs Non-Face Images

The success and failure cases are picked randomly.

Success Cases (first two rows ⇒PCA, second two rows ⇒LDA):

Failure Cases (first two rows ⇒PCA, second two rows ⇒LDA):



Dominant eigenvectors:

We will choose 1 (c-1) dominant eigenvectors, which corresponds to the 2 classes, Face or No-Face

Accuracy Table:

Num of Non-Face at K = 1 and Face = 400 images	PCA	LDA
50	0.957778	0.9466666666666667
100	0.914	0.864
200	0.8858	0.8566666666666667
300	0.8971	0.7742857142857142
400	0.9225	0.7525

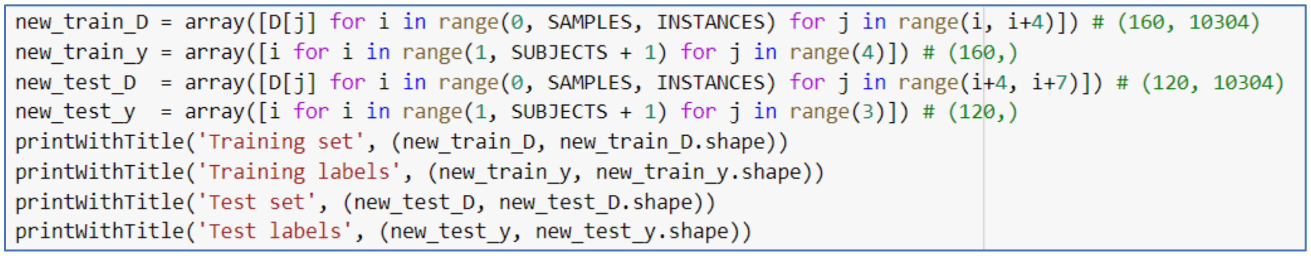
Criticizing more non-Face Data:

Increasing the data can lead to the model overfitting the data which means that if you add more pictures, it is not always better nor that it will achieve better results. But especially in LDA, the accuracy keeps decreasing.

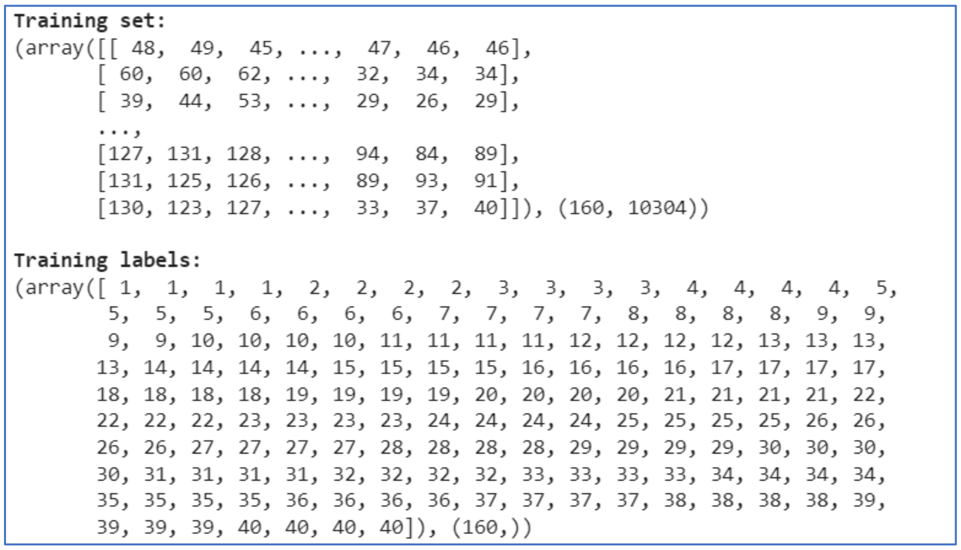
8. Bonus

a. Using different Training and Test splits

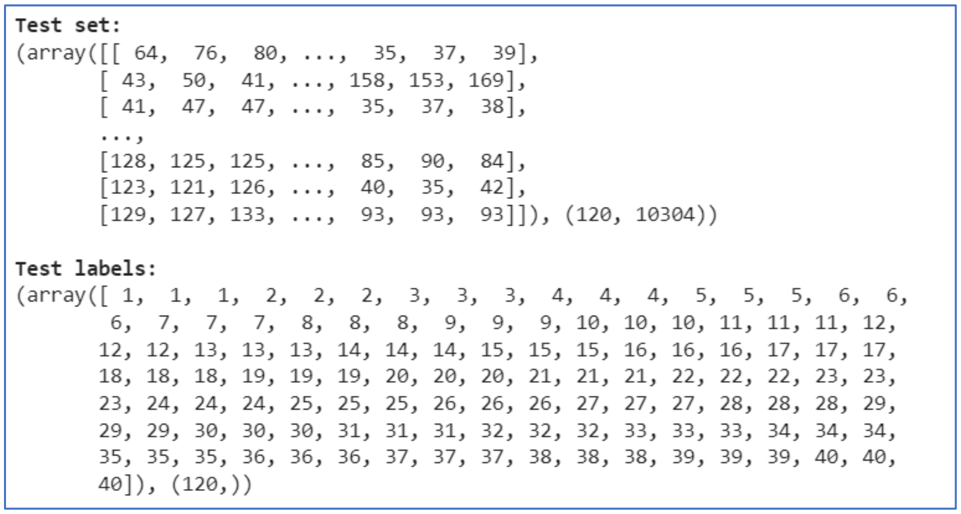
- To perform these splits, only **the first 7 instances** from each subject are considered (a total of 280 samples) and then they split as follows:
  - **Training set** contains **the first 4 instances** (160 samples).
  - **Test set** contains **the remaining 3 instances** (120 samples).



```
new_train_D = array([D[j] for i in range(0, SAMPLES, INSTANCES) for j in range(i, i+4)]) # (160, 10304)
new_train_y = array([i for i in range(1, SUBJECTS + 1) for j in range(4)]) # (160,)
new_test_D  = array([D[j] for i in range(0, SAMPLES, INSTANCES) for j in range(i+4, i+7)]) # (120, 10304)
new_test_y  = array([i for i in range(1, SUBJECTS + 1) for j in range(3)]) # (120,)
printWithTitle('Training set', (new_train_D, new_train_D.shape))
printWithTitle('Training labels', (new_train_y, new_train_y.shape))
printWithTitle('Test set', (new_test_D, new_test_D.shape))
printWithTitle('Test labels', (new_test_y, new_test_y.shape))
```

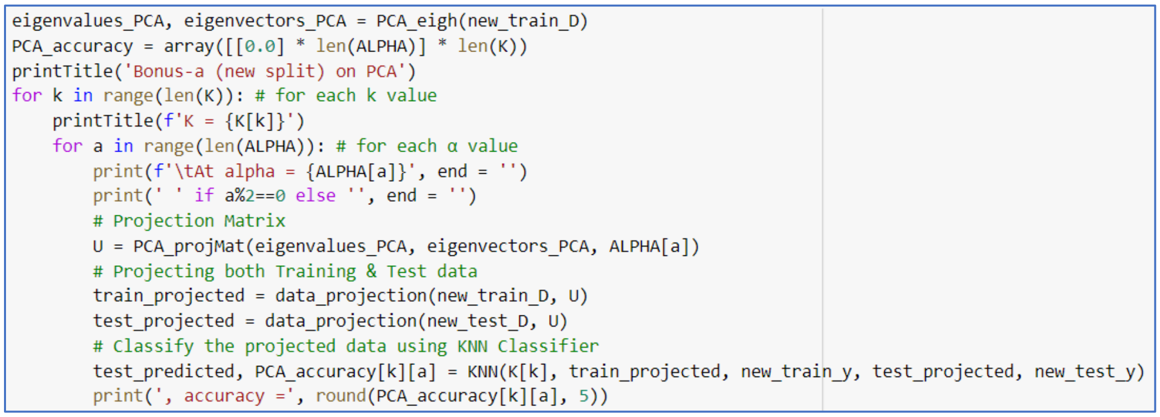


```
Training set:
(array([[ 48,  49,  45, ...,  47,  46,  46],
       [ 60,  60,  62, ...,  32,  34,  34],
       [ 39,  44,  53, ...,  29,  26,  29],
       ...,
       [127, 131, 128, ...,  94,  84,  89],
       [131, 125, 126, ...,  89,  93,  91],
       [130, 123, 127, ...,  33,  37,  40]]), (160, 10304))

Training labels:
(array([ 1,  1,  1,  1,  2,  2,  2,  2,  3,  3,  3,  3,  4,  4,  4,  4,  5,
        5,  5,  5,  6,  6,  6,  6,  7,  7,  7,  7,  8,  8,  8,  8,  9,  9,
        9,  9, 10, 10, 10, 10, 11, 11, 11, 11, 12, 12, 12, 12, 13, 13, 13,
       13, 14, 14, 14, 14, 15, 15, 15, 15, 16, 16, 16, 16, 17, 17, 17, 17,
       18, 18, 18, 18, 19, 19, 19, 19, 20, 20, 20, 20, 21, 21, 21, 21, 22,
       22, 22, 22, 23, 23, 23, 23, 24, 24, 24, 24, 25, 25, 25, 25, 26, 26,
       26, 26, 27, 27, 27, 27, 28, 28, 28, 28, 29, 29, 29, 29, 30, 30, 30,
       30, 31, 31, 31, 31, 32, 32, 32, 32, 33, 33, 33, 33, 34, 34, 34, 34,
       35, 35, 35, 35, 36, 36, 36, 36, 37, 37, 37, 37, 38, 38, 38, 38, 39,
       39, 39, 39, 40, 40, 40, 40]), (160,))
```

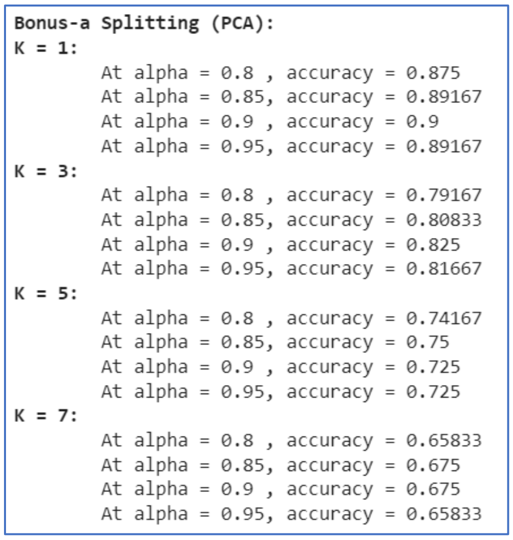


```
Test set:
(array([[ 64,  76,  80, ...,  35,  37,  39],
       [ 43,  50,  41, ..., 158, 153, 169],
       [ 41,  47,  47, ...,  35,  37,  38],
       ...,
       [128, 125, 125, ...,  85,  90,  84],
       [123, 121, 126, ...,  40,  35,  42],
       [129, 127, 133, ...,  93,  93,  93]]), (120, 10304))

Test labels:
(array([ 1,  1,  1,  2,  2,  2,  3,  3,  3,  4,  4,  4,  5,  5,  5,  6,  6,
        6,  7,  7,  7,  8,  8,  8,  9,  9,  9, 10, 10, 10, 11, 11, 11, 12,
       12, 12, 13, 13, 13, 14, 14, 14, 15, 15, 15, 16, 16, 16, 17, 17, 17,
       18, 18, 18, 19, 19, 19, 20, 20, 20, 21, 21, 21, 22, 22, 22, 23, 23,
       23, 24, 24, 24, 25, 25, 25, 26, 26, 26, 27, 27, 27, 28, 28, 28, 29,
       29, 29, 30, 30, 30, 31, 31, 31, 32, 32, 32, 33, 33, 33, 34, 34, 34,
       35, 35, 35, 36, 36, 36, 37, 37, 37, 38, 38, 38, 39, 39, 39, 40, 40,
       40]), (120,))
```

- Applying **PCA** on the new split:

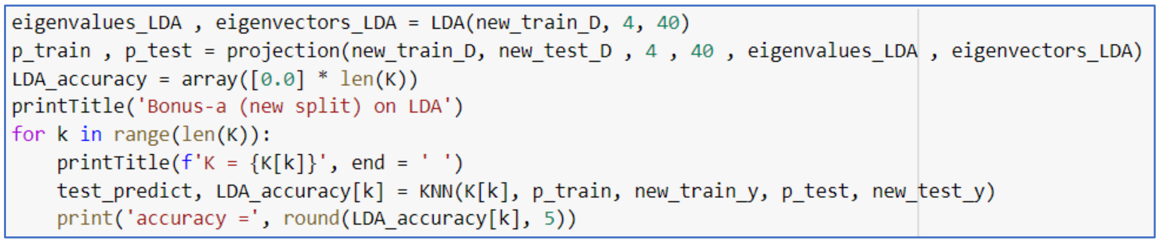


```
eigenvalues_PCA, eigenvectors_PCA = PCA_eigh(new_train_D)
PCA_accuracy = array([[0.0] * len(ALPHA)] * len(K))
printTitle('Bonus-a (new split) on PCA')
for k in range(len(K)): # for each k value
    printTitle(f'K = {K[k]}')
    for a in range(len(ALPHA)): # for each α value
        print(f'\tAt alpha = {ALPHA[a]}', end = '')
        print(' ' if a%2==0 else '', end = '')
        # Projection Matrix
        U = PCA_projMat(eigenvalues_PCA, eigenvectors_PCA, ALPHA[a])
        # Projecting both Training & Test data
        train_projected = data_projection(new_train_D, U)
        test_projected = data_projection(new_test_D, U)
        # Classify the projected data using KNN Classifier
        test_predicted, PCA_accuracy[k][a] = KNN(K[k], train_projected, new_train_y, test_projected, new_test_y)
        print(', accuracy =', round(PCA_accuracy[k][a], 5))
```

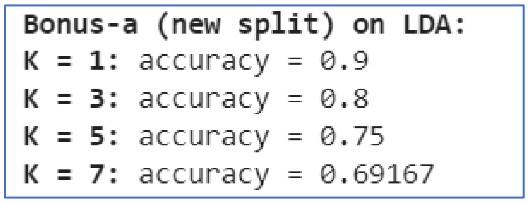


```
Bonus-a Splitting (PCA):
K = 1:
        At alpha = 0.8 , accuracy = 0.875
        At alpha = 0.85, accuracy = 0.89167
        At alpha = 0.9 , accuracy = 0.9
        At alpha = 0.95, accuracy = 0.89167
K = 3:
        At alpha = 0.8 , accuracy = 0.79167
        At alpha = 0.85, accuracy = 0.80833
        At alpha = 0.9 , accuracy = 0.825
        At alpha = 0.95, accuracy = 0.81667
K = 5:
        At alpha = 0.8 , accuracy = 0.74167
        At alpha = 0.85, accuracy = 0.75
        At alpha = 0.9 , accuracy = 0.725
        At alpha = 0.95, accuracy = 0.725
K = 7:
        At alpha = 0.8 , accuracy = 0.65833
        At alpha = 0.85, accuracy = 0.675
        At alpha = 0.9 , accuracy = 0.675
        At alpha = 0.95, accuracy = 0.65833
```

- Applying **LDA** on the new split:



```
eigenvalues_LDA , eigenvectors_LDA = LDA(new_train_D, 4, 40)
p_train , p_test = projection(new_train_D, new_test_D , 4 , 40 , eigenvalues_LDA , eigenvectors_LDA)
LDA_accuracy = array([0.0] * len(K))
printTitle('Bonus-a (new split) on LDA')
for k in range(len(K)):
    printTitle(f'K = {K[k]}', end = ' ')
    test_predict, LDA_accuracy[k] = KNN(K[k], p_train, new_train_y, p_test, new_test_y)
    print('accuracy =', round(LDA_accuracy[k], 5))
```



```
Bonus-a (new split) on LDA:
K = 1: accuracy = 0.9
K = 3: accuracy = 0.8
K = 5: accuracy = 0.75
K = 7: accuracy = 0.69167
```

⇒ The Accuracies obtained are **lower** compared to the 50% split accuracies due to decreasing the number of samples used in the Training set.