Assignment #1 Face Recognition

Farah Moustafa 6888

Asala Ahmed 6916

Zahraa ElHareedy 6895

Table of Contents

Google CoLab Notebook link:

https://colab.research.google.com/drive/1Xb7UOgNcDrcuR3S0ng2uXtaGI7HkjSwi#scrollTo=k5MJhxIeqm4y

1.AT&T Database of Faces

Our Database of Faces, (formerly 'The ORL Database of Faces'), contains a set of face images taken between April 1992 and April 1994 at the lab. The database was used in the context of a face recognition project carried out in collaboration with the Speech, Vision and Robotics Group of the Cambridge University Engineering Department.

There are ten different images of each of 40 distinct subjects. For some subjects, the images were taken at different times, varying the lighting, facial expressions (open / closed eyes, smiling / not smiling) and facial details (glasses / no glasses). All the images were taken against a dark homogeneous background with the subjects in an upright, frontal position (with tolerance for some side movement). A preview image of the Database of Faces is available.

The files are in PGM format, and can conveniently be viewed on UNIX (TM) systems using the 'xv' program. The size of each image is 92x112 pixels, with 256 grey levels per pixel. The images are organised in 40 directories (one for each subject), which have names of the form sX, where X indicates the subject number (between 1 and 40). In each of these directories, there are ten different images of that subject, which have names of the form Y.pgm, where Y is the image number for that subject (between 1 and 10).

2.Imports

Imports

```
[ ]  from sklearn import metrics
     import os
     import numpy as np
     from PIL import Image
     from zipfile import ZipFile
     from sklearn.metrics.pairwise import euclidean_distances
     from sklearn.metrics import accuracy_score
     from collections import Counter
     import matplotlib.pyplot as plt
     from keras.datasets import cifar10
     from sklearn import metrics
     import time
```

3.Download the Dataset and Understand the Format

The Dataset is downloaded from the link https://www.kaggle.com/kasikrit/att-database-of-faces/ and uploaded on the Google CoLab Notebook as a zipped file.

3.1.They are unzipped using the following code to a folder named ‘dataset’:

Unzipping

```
[ ]
      folder_name = '/archive.zip'
      with ZipFile(folder_name, 'r') as zip:
        zip.extractall('/dataset')
```

4. Generate the Data Matrix and the Label vector

Two nested loops are used to create the D and y matrices which contain the Data and labels respectively.

Outer loop loops over subjects (40), while the inner loop loops over images for each subject(10). For each image, it is coverted to a vector using .reshape(-1) and added to its corresponding position in the two matrices. The whole procedure is done as following:

Generate the Data Matrix and the Label vector

```
[ ] subjects_no = 40
    images_per_subject = 10
    width = 92
    height = 112
    D = np.empty([subjects_no * images_per_subject , width * height])
    y = np.empty(subjects_no * images_per_subject)
    for i in range(1,subjects_no+1) : #loop on each subject inside main folder
      for j in range(1,images_per_subject+1): #loop for each image in each subject
                image_path = os.path.join('/dataset', f's{i}', f'{j}.pgm')
                image = Image.open(image_path)
                #np.array(image).shape gives (112 , 92) which are the dimensions of each image
                image_vector = np.array(image).reshape(-1) #convert it into 1D
                #len(image_vector) gives 10304 which is 92 x 112
                D[(i-1)*images_per_subject+(j-1) ,: ] = image_vector
                y[(i-1)*images_per_subject+(j-1) ] = i

    print(f'Data Matrix D shape: {D.shape}')
    print(f'Label vector y shape: {y.shape}')
```

```
Data Matrix D shape: (400, 10304)
Label vector y shape: (400,)
```

5.Split the Dataset into Training and Test sets

5.1.From the Data Matrix D400x10304 keep the odd rows for training and the even rows for testing. This will give you 5 instances per person for training and 5 instances per person for testing.

This is done using the function np.arange() , which is given a step of 2, and creates a sequence accordingly to produce the indices. It's done once for testing and another for training indices.

```
training_set_indices = np.arange(1,400,2) #returns evenly spaced values within the given interval, for odd
testing_set_indices = np.arange(0,400,2) #for even
training_set, testing_set, training_labels, testing_labels  = split_data(D,y,training_set_indices,testing_set_indices)
```

5.2.Split the labels vector accordingly.

A separate function is created for the splitting stage.

Split the Dataset into Training and Test sets

```
def split_data(D,y,training_set_indices,testing_set_indices):
    training_set = D[training_set_indices]
    testing_set = D[testing_set_indices]
    training_labels = y[training_set_indices]
    testing_labels = y[testing_set_indices]
    print(training_set)
    print(testing_set.shape)
    print(testing_labels.shape)
    print(training_labels.shape)
    return training_set, testing_set, training_labels, testing_labels
```

6. Classification using PCA

6.1. Eigen Faces

– Face Images are projected into a feature space (“Face Space”) that best encodes the variation among known face images.

– The face space is defined by the “eigenfaces”, which are the eigenvectors of the set of faces.

6.2.The algorithm Used is shown as below :

ALGORITHM 7.1. Principal Component Analysis

PCA $(\mathbf{D}, \alpha)$:

1 $\mu = \frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_i$ // compute mean
2 $\mathbf{Z} = \mathbf{D} - \mathbf{1}\cdot\mu^T$ // center the data
3 $\mathbf{\Sigma} = \frac{1}{n}(\mathbf{Z}^T\mathbf{Z})$ // compute covariance matrix
4 $(\lambda_1, \lambda_2, \ldots, \lambda_d) = \text{eigenvalues}(\mathbf{\Sigma})$ // compute eigenvalues
5 $\mathbf{U} = (\mathbf{u}_1 \quad \mathbf{u}_2 \quad \cdots \quad \mathbf{u}_d) = \text{eigenvectors}(\mathbf{\Sigma})$ // compute eigenvectors
6 $f(r) = \frac{\sum_{i=1}^{r}\lambda_i}{\sum_{i=1}^{d}\lambda_i}$, for all $r = 1, 2, \ldots, d$ // fraction of total variance
7 Choose smallest r so that $f(r) \geq \alpha$ // choose dimensionality
8 $\mathbf{U}_r = (\mathbf{u}_1 \quad \mathbf{u}_2 \quad \cdots \quad \mathbf{u}_r)$ // reduced basis
9 $\mathbf{A} = \{\mathbf{a}_i \mid \mathbf{a}_i = \mathbf{U}_r^T\mathbf{x}_i, \text{for } i = 1, \ldots, n\}$ // reduced dimensionality data

6.3.Implemented code is below:

Classification using PCA

```
[ ] def PCA(training_set):
      mean_vector = training_set.mean(axis=0)
      Z = training_set - mean_vector #centralize data
      Z_t = np.transpose(Z)
      cov_matrix = np.dot(Z_t,Z)/training_set.shape[0]
      eigen_values, U = np.linalg.eigh(cov_matrix)
      indices = np.argsort(eigen_values)[::-1] #sort descendingly
      eigen_values_vector = eigen_values[indices]
      eigen_values = np.diag(eigen_values_vector)
      U = U[:,indices]
      print(U.shape)
      print(eigen_values.shape)
      verified = np.dot(U,eigen_values)
      verified = np.dot(verified,np.transpose(U))
      print(np.allclose(cov_matrix, verified, rtol=1e-3, atol=1e-3)) #check if verified maps back to D
      return U , Z , eigen_values
```

6.4. Code explanation:

The code maps to the algorithm preceding it. From getting the mean vector, to the Z and Covariance matrix, followed by getting the eigen values and corresponding eigen vectors.

At the end of the function, 'allclose' is used to verify that $U\Lambda U^T$ is equivalent to the covariant matrix, to make sure sequence is correct.

6.5. Calling the PCA function to get eigen vectors for training data

PCA

```
[21] start_PCA = time.time()
     U ,Z , eigen_values= PCA(training_set)
     end_PCA = time.time()
     runtime_PCA = end_PCA - start_PCA
     print(runtime_PCA)
```

```
(10304, 10304)
(10304, 10304)
True
463.60799646377563
```

Note that True is for the verification.

Time complexity will be used later on, to compare with other algorithm used, Kernel PCA.

6.6.Centering testing data to be projected

Centering testing data

```
[ ]  mean_testing = testing_set.mean(axis=0)
     Z_testing = testing_set - mean_testing
```

6.7. Dimensions function

This function takes alpha and the eigen values as parameters, and returns the number of vectors needed to achieve this alpha, this is done by the method shown below:

Choosing Dimensions

```
[ ]  def dimensions( alpha, eigen_values):
       r = 0 #dimensions
       explained_variance = 0
       sum = np.sum(eigen_values) #get total
       for value in eigen_values :
         explained_variance = explained_variance + value[r] #each time add a value
         r = r + 1
         if ((explained_variance / sum) >= alpha): #check if threshold is reached
           break
       return r
```

6.8. Getting projection matrix U, and projecting the data. Define the alpha = {0.8,0.85,0.9,0.95}

Function 'Dimensions' mentioned above is used to get dimensions needed for each alpha, and then the matrix is used to project both the train and test data.

Note that the Projection Matrix is derived from training data, and testing data is not used at all, as it should be invisible to the model.

alpha 1, 80%

```
[ ] R1=dimensions(0.8,eigen_values)  #get numbers of dimensions for 80% accuracy
    print(R1)
    Ur = U[:, 0 : R1]  #choose only specified dimensions
    projected_matrix_R1 = np.dot(Z,Ur)  #project training data, with specified dimensions
    projected_matrix_test_R1 = np.dot(Z_testing,Ur) #project testing data
    print(projected_matrix_R1)
```

```
37
```

Alpha 2, 85%

```
[ ]  R2=dimensions(0.85,eigen_values)
     print(R2)
     Ur = U[:, 0 : R2]
     projected_matrix_R2 = np.dot(Z,Ur)
     projected_matrix_test_R2 = np.dot(Z_testing,Ur)
```

```
53
```

Alpha 3, 90%

```
[ ]
    R3=dimensions(0.9,eigen_values)
    print(R3)
    Ur = U[:, 0 : R3]
    projected_matrix_R3 = np.dot(Z,Ur)
    projected_matrix_test_R3 = np.dot(Z_testing,Ur)
```

```
77
```

Alpha 4, 95%

```
[ ]
    R4=dimensions(0.95,eigen_values)
    print(R4)
    Ur = U[:, 0 : R4]
    projected_matrix_R4 = np.dot(Z,Ur)
    projected_matrix_test_R4 = np.dot(Z_testing,Ur)
```

```
116
```

Note that : for 0.8 alpha, 37 vectors are used,

For 0.85 alpha, 53 vectors are used,

For 0.9 alpha, 77 vectors are used,

And For 0.95 accuracy, 116 vectors are used.

This makes sense as, as alpha, which is the explained variance factor, increases, more dimensions are needed to achieve this accuracy.

6.9. K-NN function

K-NN is a function that takes k (number of nearest neighbors needed, both projected train and test data and labels as well, and returns the accuracy obtained for this specified k. It also displays the confusion matrix, and displays a combination of true positive, true negative, false positive and false negative, but in the case of faces vs non-faces (binary classification), which will be discussed later.

The function first computes the Euclidean distance matrix between training and testing values, then each row is sorted ascendingly to get the nearest k neighbors.

In case the function was called when the faces dataset is used, test_against_neighbor() function is called to print three random images, illustrating its predicted class, original class, and relation between them, whether they are the same or different. It will be discussed in a following point and outputs will be shown in visualization of facial recognition section.

In the case of k> 1, voting is done to choose just one neighbor. Voting function will be discussed in the following point.

KNN

```
[7] def KNN(k, projected_test, projected_train,training_labels,testing_labels,faces_nf_test=[],faces_nf_train=[]):
      euclidean_distance = euclidean_distances(projected_test,projected_train)
      neighbor_index = np.argsort(euclidean_distance)[:,0:k]
      neighbor_label = training_labels[neighbor_index]
      accuracy = accuracy_score(testing_labels,neighbor_label)
      if k != 1:
        neighbor_label = voting(neighbor_label)
      if testing_labels[0] == 1: # faces dataset
        test_against_neighbor(testing_labels,neighbor_label,testing_set,training_set,neighbor_index)
      if testing_labels[0] == 0: # face vs non face dataset
        confusion_matrix = metrics.confusion_matrix(testing_labels,neighbor_label)
        cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix)
        cm_display.plot()
        plt.show()
        show_success_failure(testing_labels,neighbor_label,faces_nf_test,faces_nf_train,neighbor_index)
      return accuracy*100
```

6.10. Voting Function

The voting function decides which neighbor is the winner.

For each row, it counts the number of occurrence of each class, and then gets the number of max occurrence.

If it's one, it means that this is the winner class.

If not, it means that there is a tie, and it has to be broken.

For this, the first occurrence neighbor is chosen, using the function discussed in the following point.

Lastly the winner class is appended to the winning labels array and returned to the KNN function.

Voting

```
[ ]  def voting(neighbor_label):
       winning_labels = []
       for row in neighbor_label:
         counts = Counter(row)
         # Find the elements with the highest count
         max_count = 0
         max_elements = []
         for element, count in counts.items():
             if count > max_count:
                 max_count = count
                 max_elements = [element]
             elif count == max_count:
                 max_elements.append(element)
         if len(max_elements) > 1 :
           max_elements = [find_first_occurrence(row,max_elements)]
         winning_labels.append(max_elements)
       return winning_labels
```

6.11. Find first occurrence function

In this function, there are two nested loops. The function passes on each element in the neighbor labels, note that they are sorted from the nearest distance to the furthest, and if it's in the max occurrence elements, the loop is broken and it returns the element.

Find First Occurence

```
[ ]  def find_first_occurrence(neighbor_labels, max_elements):
         for elem in neighbor_labels:
             if elem in max_elements:
                 return elem
         return None
```

6.12. Test against neighbor function

Function loops three times, for every image, it checks whether the predicted label was the same as the original label, prints correctness of classification accordingly, followed by the image itself and the closest neighbor used in classification of the image using display_photos() function shown faces vs non-face images section.

```
def test_against_neighbor(testing_labels,neighbor_labels,testing_set,training_set,neighbor_indices):
  counter =0
  while counter < 3 :
    random_index = np.random.choice(testing_labels.shape[0], 1, replace=False)
    neighbor_index = neighbor_indices[random_index]
    if testing_labels[random_index] == neighbor_labels[random_index]:
      print("This face is correcctly classified to its class")
      display_photos(random_index,testing_set,training_set,neighbor_index,testing_labels[random_index],neighbor_labels[random_index])
      print("------------------------------------")
    elif testing_labels[random_index] != neighbor_labels[random_index]:
      print("This face is incorrecctly classified to a wrong class")
      display_photos(random_index,testing_set,training_set,neighbor_index,testing_labels[random_index],neighbor_labels[random_index])
      print("------------------------------------")
    counter = counter + 1
```

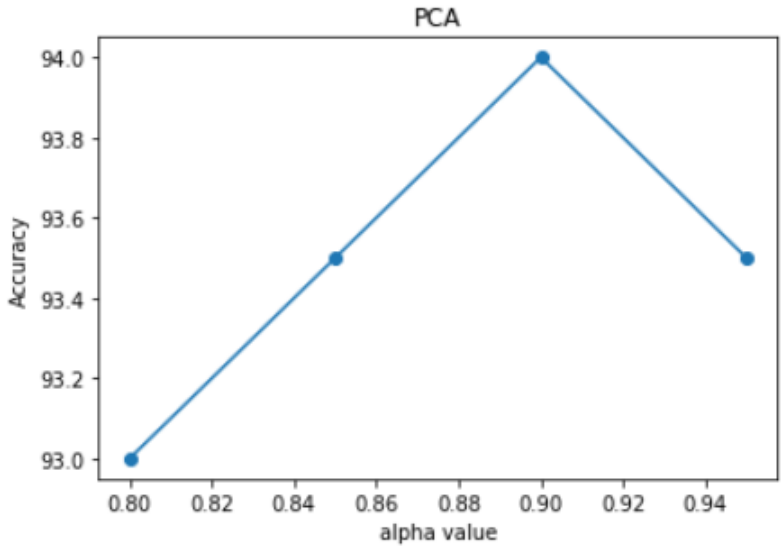
6.13. F-NN for each alpha

Note that F-NN is the K-NN with k = 1

```
accuracy_PCA_R1 = KNN(1,projected_matrix_test_R1,projected_matrix_R1,training_labels,testing_labels) #First NN is KNN with k =1
print(f'alpha = 0.8 accuracy:{accuracy_PCA_R1}%')

accuracy_PCA_R2 = KNN(1,projected_matrix_test_R2,projected_matrix_R2,training_labels,testing_labels) #First NN is KNN with k =1
print(f'alpha = 0.85 accuracy:{accuracy_PCA_R2}%')

accuracy_PCA_R3 = KNN(1,projected_matrix_test_R3,projected_matrix_R3,training_labels,testing_labels) #First NN is KNN with k =1
print(f'alpha = 0.9 accuracy:{accuracy_PCA_R3}%')

accuracy_PCA_R4 = KNN(1,projected_matrix_test_R4,projected_matrix_R4,training_labels,testing_labels) #First NN is KNN with k =1
print(f'alpha = 0.95 accuracy:{accuracy_PCA_R4}%')

accuracy_PCA_R5 = KNN(1,projected_matrix_test_R5,projected_matrix_R5,training_labels_new,testing_labels_new) #First NN is KNN with k =1
print(f'alpha = 0.9 accuracy:{accuracy_PCA_R5}%')
```

```
alpha = 0.8 accuracy:93.0%
alpha = 0.85 accuracy:93.5%
alpha = 0.9 accuracy:94.0%
alpha = 0.95 accuracy:93.5%
alpha = 0.9 accuracy:95.0%
```

6.14. Alpha vs accuracy



It can be noticed that as alpha increases, accuracy increases, until a certain threshold, accuracy returns to decrease, which is a result of overfitting.

Alpha = 0.9 gives the best result in our case.

7. Classification Using LDA

7.1. Intuition

We want to find the direction that gives us best classification results; which is to:

– Maximize the distances between classes.

– With respect to Total variance in the different classes.

7.2. Algorithm used

ALGORITHM 20.1. Linear Discriminant Analysis

LINEARDISCRIMINANT ($\mathbf{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$):

1 $\mathbf{D}_i \leftarrow \{\mathbf{x}_j \mid y_j = c_i, j = 1, \ldots, n\}, i = 1, 2$ // class-specific subsets
2 $\boldsymbol{\mu}_i \leftarrow \text{mean}(\mathbf{D}_i), i = 1, 2$ // class means
3 $\mathbf{B} \leftarrow (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)^T$ // between-class scatter matrix
4 $\mathbf{Z}_i \leftarrow \mathbf{D}_i - \mathbf{1}_{n_i} \boldsymbol{\mu}_i^T, i = 1, 2$ // center class matrices
5 $\mathbf{S}_i \leftarrow \mathbf{Z}_i^T \mathbf{Z}_i, i = 1, 2$ // class scatter matrices
6 $\mathbf{S} \leftarrow \mathbf{S}_1 + \mathbf{S}_2$ // within-class scatter matrix
7 $\lambda_1, \mathbf{w} \leftarrow \text{eigen}(\mathbf{S}^{-1}\mathbf{B})$ // compute dominant eigenvector

Note: We Replace B matrix by Sb, as we have more than 2 classes.

$$Sb = \sum_{k=1}^{m} n_k(\mu_k - \mu)\ (\mu_k - \mu)^T$$

7.3. Implemented LDA function

Classification using LDA

```
[ ]  def LDA(training_set,training_labels,subjects_no,width,height,eigen_vector_no):
       total_mean = training_set.mean(axis=0)
       Sb = np.zeros((width*height,width*height))
       S = np.zeros((width*height,width*height))
       class_mean = np.empty([subjects_no,width*height])
       for id in range(1,subjects_no+1):
         class_matrix = training_set[training_labels == id ]  #select data in this class
         samples_no_per_class = class_matrix.shape[0]
         class_mean[id-1] = class_matrix.mean(axis=0)

         class_Z = class_matrix - class_mean[id-1]
         class_Z_transpose = np.transpose(class_Z)
         Si = np.dot(class_Z_transpose,class_Z)
         S = S + Si
         mean_diff = (class_mean[id-1] - total_mean).reshape(width*height,1)
         mean_diff_transpose = np.transpose(mean_diff)
         Sb = Sb + samples_no_per_class * np.dot(mean_diff,mean_diff_transpose)

       S_inv = np.linalg.inv(S)
       S_inv_Sb = np.dot(S_inv,Sb)
       lamda_eigen_values, w = np.linalg.eigh(S_inv_Sb)
       lamda_indices = np.argsort(lamda_eigen_values)[::-1] #sort descendingly
       lamda_eigen_values_sorted = lamda_eigen_values[lamda_indices]
       w = w[lamda_indices]
       print(w.shape)
       U_lda = w[:,0:eigen_vector_no]
       return U_lda
```

7.3. Code explanation

The code maps to the algorithm preceding it.

It loops for each subject (class), gets its mean, and centered class matrix, to get the between class scatter matrix, and within class scatter matrix, which it uses to get the corresponding eigen values and vectors.

7.4. Projecting training and testing sets

Classification using LDA

```
[32] eigen_vector_no = 39
     start_LDA = time.time()
     U_lda = LDA(training_set,training_labels,subjects_no,width,height,eigen_vector_no)
     end_LDA = time.time()
     runtime_LDA = end_LDA - start_LDA
     print(runtime_LDA)

     (10304, 10304)
     592.54332280159
```

Time complexity will be used later on, to compare with other algorithm used, Regularized LDA.

Pojection

```
[ ]  projected_matrix_lda = np.dot(Z,U_lda)
     projected_matrix_test_lda = np.dot(Z_testing,U_lda)
```

Note that the number of linear discriminants is at most c−1 (40-1) where c is the number of class labels, since the in-between scatter matrix SB is the sum of c matrices with rank 1 or less.
In addition, note that the Projection Matrix is derived from training data, and testing data is not used at all, as it should be invisible to the model**.**

7.5. Classifier to determine class labels and LDA accuracy

Previously used KNN function is used to classify the testing data and provide the accuracy, which turned out to be 95.5%.

First-NN LDA

```
[ ] accuracy_LDA = KNN(1,projected_matrix_test_lda,projected_matrix_lda,training_labels,testing_labels) # First-NN is KNN with k =1
    print(f'Accuracy LDA : {accuracy_LDA}% ' )

    Accuracy LDA : 95.5%
```

8.Classifier Tuning

8.1. Testing for different ks

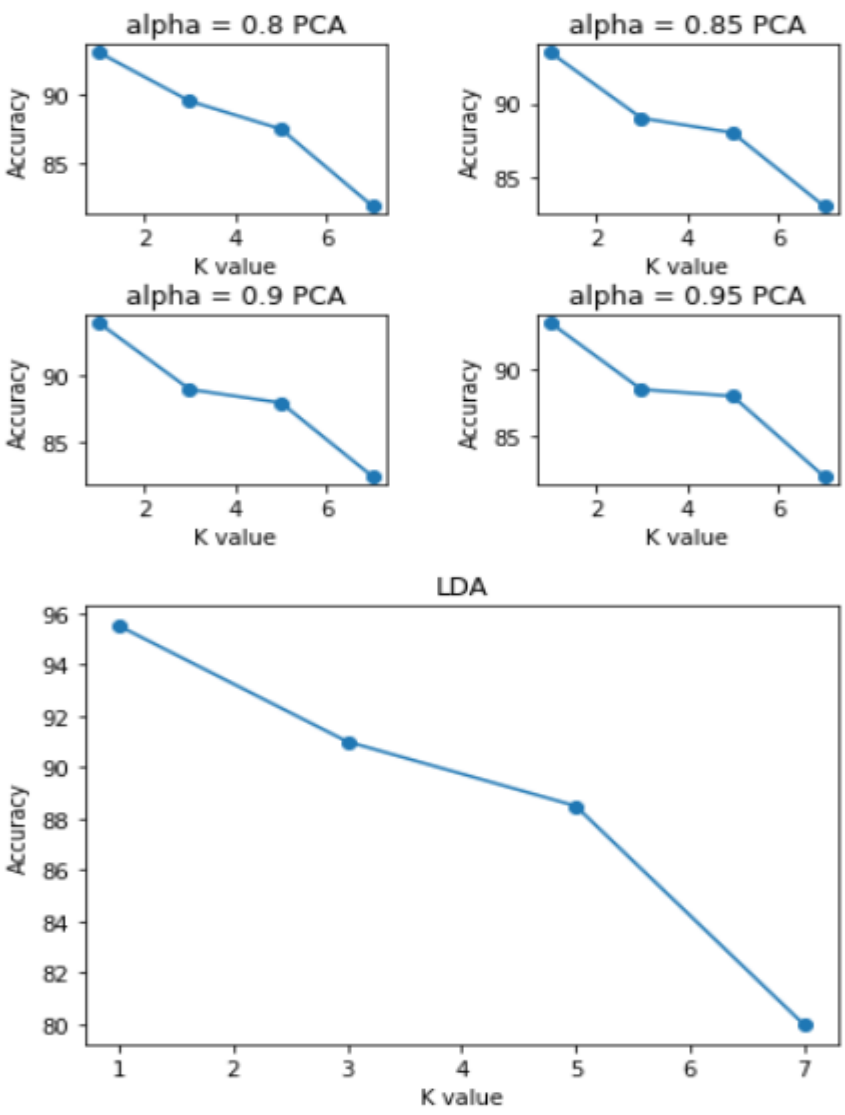
Using the previously described KNN function, ks of values 1,3,5 and 7 are tested for each alpha value in PCA, and for LDA as well.
K is the number of neighbors which will be used to choose the most probable neighbor then.

K vs accuracy PCA , different alphas , and LDA

```
[ ] ks = np.arange(1,8,2)
    accuracy_alpha1 = []
    accuracy_alpha2 = []
    accuracy_alpha3 = []
    accuracy_alpha4 = []
    accuracy_LDA = []
    for k in ks:

        accuracy_alpha1.append(KNN(k,projected_matrix_test_R1,projected_matrix_R1,training_labels,testing_labels))
        accuracy_alpha2.append(KNN(k,projected_matrix_test_R2,projected_matrix_R2,training_labels,testing_labels))
        accuracy_alpha3.append(KNN(k,projected_matrix_test_R3,projected_matrix_R3,training_labels,testing_labels))
        accuracy_alpha4.append(KNN(k,projected_matrix_test_R4,projected_matrix_R4,training_labels,testing_labels))
        accuracy_LDA.append(KNN(k,projected_matrix_test_lda,projected_matrix_lda,training_labels,testing_labels))
```

8.2. Plotting performance measure against the K value, for each alpha in PCA, and for LDA.

```
plt.subplot(2, 2, 1)
plt.subplots_adjust(wspace = 0.5, hspace = 0.6)
plt.plot(ks,accuracy_alpha1,'-o')
plt.xlabel('K value')
plt.ylabel('Accuracy')
plt.title('alpha = 0.8 PCA')
plt.subplot(2, 2, 2)
plt.plot(ks,accuracy_alpha2,'-o')
plt.xlabel('K value')
plt.ylabel('Accuracy')
plt.title('alpha = 0.85 PCA')
plt.subplot(2, 2, 3)
plt.plot(ks,accuracy_alpha3,'-o')
plt.xlabel('K value')
plt.ylabel('Accuracy')
plt.title('alpha = 0.9 PCA')
plt.subplot(2, 2, 4)
plt.plot(ks,accuracy_alpha4,'-o')
plt.xlabel('K value')
plt.ylabel('Accuracy')
plt.title('alpha = 0.95 PCA')
plt.show()
plt.plot(ks,accuracy_LDA,'-o')
plt.xlabel('K value')
plt.ylabel('Accuracy')
plt.title('LDA')
plt.show()
```

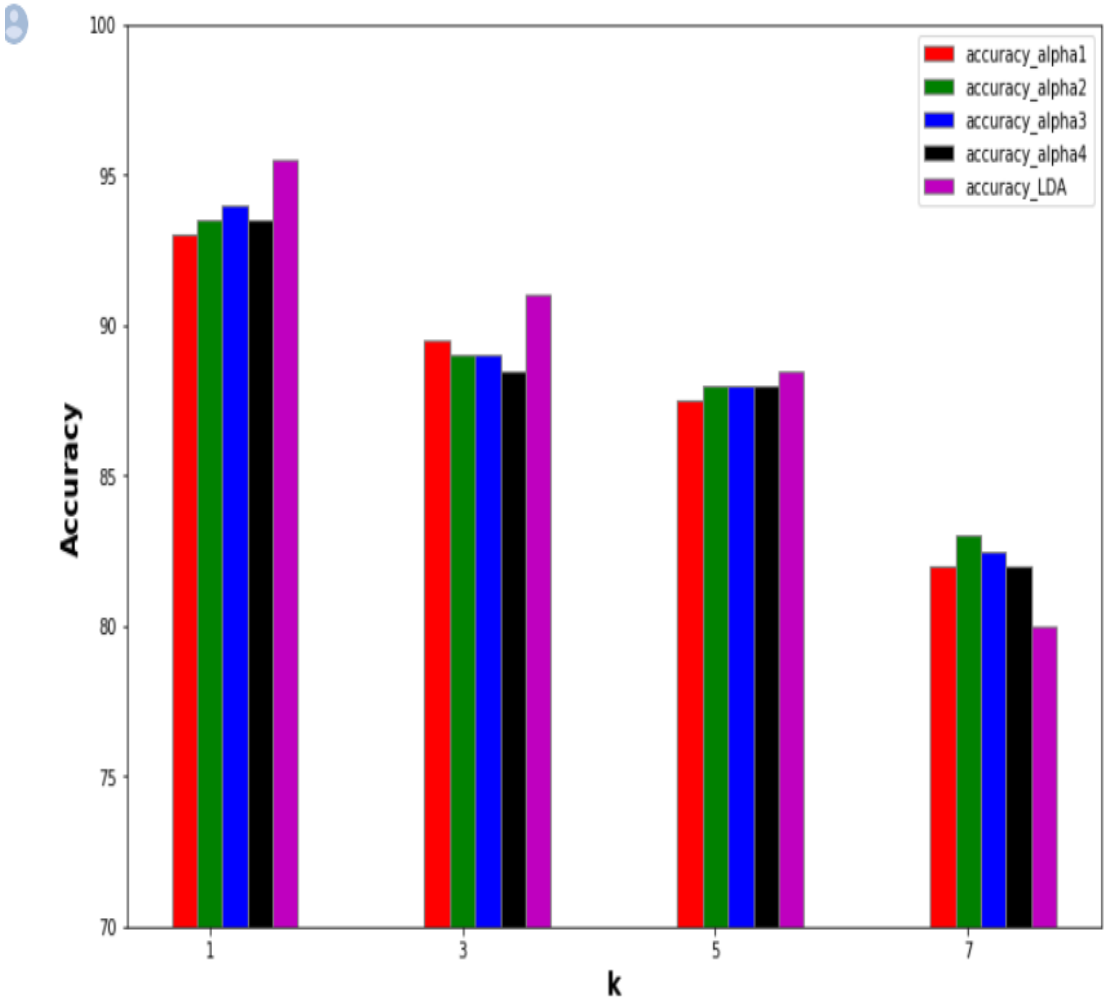


It can be shown that as K increases, accuracy decreases for all values of alphas in PCA, as well as LDA

9.PCA vs LDA Comparison

Comparing PCA od different alphas vs LDA

```
[ ]  barWidth = 0.1
     fig = plt.subplots(figsize =(12, 8))
     # Set position of bar on X axis
     br1 = np.arange(len(ks ))
     br2 = [x + barWidth for x in br1]
     br3 = [x + barWidth for x in br2]
     br4 = [x + barWidth for x in br3]
     br5 = [x + barWidth for x in br4]

     # Make the plot
     plt.bar(br1, accuracy_alpha1, color ='r', width = barWidth,
             edgecolor ='grey', label ='accuracy_alpha1')
     plt.bar(br2, accuracy_alpha2, color ='g', width = barWidth,
             edgecolor ='grey', label ='accuracy_alpha2')
     plt.bar(br3, accuracy_alpha3, color ='b', width = barWidth,
             edgecolor ='grey', label ='accuracy_alpha3')
     plt.bar(br4, accuracy_alpha4, color ='k', width = barWidth,
             edgecolor ='grey', label ='accuracy_alpha4')
     plt.bar(br5, accuracy_LDA, color ='m', width = barWidth,
             edgecolor ='grey', label ='accuracy_LDA')

     # Adding Xticks
     plt.xlabel('k', fontweight ='bold', fontsize = 15)
     plt.ylabel('Accuracy', fontweight ='bold', fontsize = 15)
     plt.xticks([r + barWidth for r in range(len(ks))],
             ['1', '3', '5', '7'])
     plt.ylim(70, 100)
     plt.legend()
     plt.show()
```



It can be shown that for k = 1,3 and 5, LDA out performs all of PCA methods.

As for k = 7, PCA with alpha = 0.85 outperforms LDA as well as other alphas for PCA.

For k = 1 and 2, PCA with alpha = 0.95 gives the worst performance, while for k = 5, alpha of 0.8 gives the worst.

As for k = 7, LDA gives the worst performance.

10.Visualization of facial recognition, test samples

10.1. Correctly Classified Images

This face is correcctly classified to its class
training
[[36.]]

testing
[36.]

This face is correcctly classified to its class
training
[[14.]]

testing
[14.]

---

alpha = 0.95 accuracy:93.5%

This face is correcctly classified to its c
training
[[22.]]

testing
[22.]

This face is correcctly classified to its class
training
[[3.]]

testing
[3.]

This face is correcctly classified to its class
training
[[9.]]

testing
[9.]

This face is correcctly classified to its class
training
[[40.]]

testing
[40.]

This face is correcctly classified to its class
training
[[20.]]

testing
[20.]

This face is correcctly classified to its class
training
[[15.]]

testing
[15.]

This face is correcctly classified to its class
training
[[11.]]

testing
[11.]

This face is correcctly classified to its class
training
[[36.]]

testing
[36.]

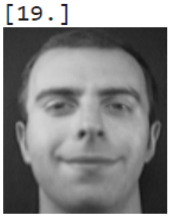
10.2 Incorrectly Classifies Images

```
This face is incorrecctly classified to a wrong class
training
[[40.]]
```

```
testing
[35.]
```

```
This face is incorrecctly classified to a wrong class
training
[[36.]]
```

```
testing
[19.]
```



```
This face is incorrecctly classified to a wrong class
training
[[3.]]
```

```
testing
[20.]
```

11. Faces Vs Non-faces

11.1. Cifar-10 Dataset

CIFAR-10 is an established computer-vision dataset used for object recognition. It is a subset of the 80 million tiny images dataset and consists of 60,000 32x32 color images containing one of 10 object classes, with 6000 images per class. It was collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.

11.2. load dataset and store only 300 random images

Compare Faces Vs Non-faces Images

Non-faces Adjustment

```
[ ]  # Load the CIFAR-10 dataset
     (train_nf, train_labels_nf), (test_nf, test_labels_nf) = cifar10.load_data()
     print(train_nf.shape)
     random_indices_total = np.random.choice(test_nf.shape[0], 300, replace=False)
     train_nf = train_nf[random_indices_total]
     test_nf = test_nf[random_indices_total]
```

```
(50000, 32, 32, 3)
```

11.3. Prepare non faces function

The Prepare non faces function takes in two sets of non-face images, one for training and one for testing. It first converts the color images to grayscale and resizes them to a fixed size of 92 by 112 pixels. The images are then flattened into 1-D vectors and stored in a list for each set. This conversion is done in order to have the same features of our Orl dataset so that we could have a fair comparison.

```
def Prepare_non_faces(train_nf, test_nf) :
  # Convert the color images to grayscale and resize to 92x112
  train_non_faces = []
  for img in train_nf:
      pil_img = Image.fromarray(img).convert('L').resize((92, 112)) #convert to gray images with size 92*112
      gray_img = np.array(pil_img.getdata()).reshape((112, 92))
      gray_img_vector = np.array(gray_img).reshape(-1) # convert to a 1-D vector
      train_non_faces.append(gray_img_vector)
  train_non_faces = np.array(train_non_faces)

  test_non_faces = []
  for img in test_nf:
      pil_img = Image.fromarray(img).convert('L').resize((92, 112))
      gray_img = np.array(pil_img.getdata()).reshape((112, 92))
      gray_img_vector = np.array(gray_img).reshape(-1)
      test_non_faces.append(gray_img_vector)

  test_non_faces = np.array(test_non_faces)
  return train_non_faces,test_non_faces
```

11.4. Selecting different random samples of 150, 200, 250, 300 cifar-10 images for training and testing and Preparing each of the selected samples

Size = 300

Selecting a sample of 300 non-faces for training-testing datasets

```
[ ]  random_indices_300 = np.random.choice(test_nf.shape[0], 300, replace=False)
     train_nf_300 = train_nf[random_indices_300]
     test_nf_300 = test_nf[random_indices_300]
     train_non_faces_300,test_non_faces_300 = Prepare_non_faces(train_nf_300,test_nf_300)
     print(train_non_faces_300)
     print(test_non_faces_300.shape)
```

```
[[132 132 132 ... 128 128 128]
 [158 158 157 ... 160 160 160]
 [144 144 145 ...   8  10  11]
 ...
 [227 227 226 ... 220 223 224]
 [138 134 122 ... 129 135 137]
 [154 154 154 ...  92  90  89]]
(300, 10304)
```

Size = 250

Selecting a sample of 250 non-faces for training-testing datasets

```
random_indices_250 = np.random.choice(test_nf.shape[0], 250, replace=False)
train_nf_250 = train_nf[random_indices_250]
test_nf_250 = test_nf[random_indices_250]
train_non_faces_250,test_non_faces_250 = Prepare_non_faces(train_nf_250,test_nf_250)
print(train_non_faces_250)
print(test_non_faces_250.shape)
```

```
[[144 144 144 ... 102 101 101]
 [187 186 185 ...  27  27  27]
 [ 63  58  48 ... 118 120 121]
 ...
 [164 163 160 ...  66  66  66]
 [ 46  48  54 ... 107 103 102]
 [ 81  82  86 ...  18  18  18]]
(250, 10304)
```

Size = 200

Selecting a sample of 200 non-faces for 50%-50% training-testing datasets

```
[ ]  random_indices = np.random.choice(test_nf.shape[0], 200, replace=False)
     train_nf = train_nf[random_indices]
     test_nf = test_nf[random_indices]
     train_non_faces,test_non_faces = Prepare_non_faces(train_nf,test_nf)
     print(train_non_faces)
     print(test_non_faces.shape)
```

Size = 150

Selecting a sample of 150 non-faces for training-testing datasets

```
[ ]  random_indices_150 = np.random.choice(test_nf.shape[0], 150, replace=False)
     train_nf_150 = train_nf[random_indices_150]
     test_nf_150 = test_nf[random_indices_150]
     train_non_faces_150,test_non_faces_150 = Prepare_non_faces(train_nf_150,test_nf_150)
     print(train_non_faces_150)
     print(test_non_faces_150.shape)
```

```
[[122 122 123 ...  92  89  88]
 [144 144 144 ... 136 135 135]
 [242 241 239 ... 119 119 119]
 ...
 [164 166 172 ...   4   3   3]
 [216 216 215 ... 127 128 128]
 [131 131 131 ... 157 158 158]]
(150, 10304)
```

11.5. Concatenating Faces and Non Faces into one dataset for each sample, as well as their labels

Size = 300

```
faces_nf_train_300 = np.concatenate((train_non_faces_300,training_set),axis=0)
faces_nf_test_300 = np.concatenate((test_non_faces_300,testing_set),axis=0)
zeros_300 = np.zeros(300)
ones_300 =  np.ones(200)
train_labels_f_nf_300 = np.concatenate((zeros_300,ones_300),axis=0)
test_labels_f_nf_300 = np.concatenate((zeros_300,ones_300),axis=0)
print(faces_nf_train_300.shape)
print(faces_nf_test_300.shape)
```

```
(500, 10304)
(500, 10304)
```

Size = 250

```
faces_nf_train_250 = np.concatenate((train_non_faces_250,training_set),axis=0)
faces_nf_test_250 = np.concatenate((test_non_faces_250,testing_set),axis=0)
zeros_250 = np.zeros(250)
ones_250 =  np.ones(200)
train_labels_f_nf_250 = np.concatenate((zeros_250,ones_250),axis=0)
test_labels_f_nf_250 = np.concatenate((zeros_250,ones_250),axis=0)
print(faces_nf_train_250.shape)
print(faces_nf_test_250.shape)
```

```
(450, 10304)
(450, 10304)
```

Size = 200

Merging faces and non-faces in a single dataset

```
faces_nf_train = np.concatenate((train_non_faces,training_set),axis=0)
faces_nf_test = np.concatenate((test_non_faces,testing_set),axis=0)
zeros = np.zeros(200)
ones =  np.ones(200)
train_labels_f_nf = np.concatenate((zeros,ones),axis=0)
test_labels_f_nf = np.concatenate((zeros,ones),axis=0)
print(faces_nf_train.shape)
print(faces_nf_test.shape)
```

```
(400, 10304)
(400, 10304)
```

Size = 150

```
faces_nf_train_150 = np.concatenate((train_non_faces_150,training_set),axis=0)
faces_nf_test_150 = np.concatenate((test_non_faces_150,testing_set),axis=0)
zeros_150 = np.zeros(150)
ones_150 =  np.ones(200)
train_labels_f_nf_150 = np.concatenate((zeros_150,ones_150),axis=0)
test_labels_f_nf_150 = np.concatenate((zeros_150,ones_150),axis=0)
print(faces_nf_train_150.shape)
print(faces_nf_test_150.shape)
```

```
(350, 10304)
(350, 10304)
```

11.6. PCA for each sample and get accuracy

Size = 300

```
eigen_faces_vector_300 ,Z_faces_300 , eigen_faces_values_300 = PCA(faces_nf_train_300)
```

```
(10304, 10304)
(10304, 10304)
True
```

```
mean_testing_fnf_300 = faces_nf_test_300.mean(axis=0)
Z_faces_testing_300 = faces_nf_test_300 - mean_testing_fnf_300
```

```
fnf_300=dimensions(0.9,eigen_faces_values_300)  #get numbers of dimensions for 90% accuracy
print(fnf_300)
Ur_faces_300 = eigen_faces_vector_300[:, 0 : fnf_300]  #choose only specified dimensions
projected_matrix_fnf_300 = np.dot(Z_faces_300,Ur_faces_300)  #project training data, with specified dimensions
projected_matrix_test_fnf_300 = np.dot(Z_faces_testing_300,Ur_faces_300) #project testing data
print(projected_matrix_fnf_300)
```

```
65
[[-1.69562440e+03 -1.19924039e+03 -3.21435034e+03 ...  4.39352577e+01
  -5.43110830e+01  8.40036042e+01]
 [ 2.67749484e+03  1.62917120e+03  3.35393540e+03 ... -3.44585152e+00
   -8.61669004e+01  2.51816754e+01]
 [-6.24656523e+03  5.94994327e+03  5.99103719e+02 ...  1.33078614e+02
   -1.44857231e+02  1.17418527e+02]
 ...
 [-2.35829374e+02 -8.84643912e+02  1.30416529e+03 ... -2.12900034e+02
    8.88006792e+01  1.43468254e+02]
 [-2.31102289e+02 -8.37530225e+02  1.55763840e+03 ... -3.20222326e+02
    7.83074291e+01  1.87398743e+02]
 [-2.31463480e+02 -5.43269799e+02  1.20832172e+03 ...  2.16359741e+01
   -1.08712355e+02 -1.41817586e+02]]
```

```
accuracy_PCA_fnf_300 = KNN(1,projected_matrix_test_fnf_300,projected_matrix_fnf_300,train_labels_f_nf_300,test_labels_f_nf_300,faces_nf_test_300,faces_nf_train_300) #First NN is KNN
print(f'alpha = 0.9 accuracy:{accuracy_PCA_fnf_300}%')
```

```
-------------------------------------
alpha = 0.9 accuracy:96.2%
```

Size = 250

```
eigen_faces_vector_250 ,Z_faces_250 , eigen_faces_values_250 = PCA(faces_nf_train_250)
```

```
(10304, 10304)
(10304, 10304)
True
```

```
mean_testing_fnf_250 = faces_nf_test_250.mean(axis=0)
Z_faces_testing_250 = faces_nf_test_250 - mean_testing_fnf_250
```

```
fnf_250=dimensions(0.9,eigen_faces_values_250)  #get numbers of dimensions for 90% accuracy
print(fnf_250)
Ur_faces_250 = eigen_faces_vector_250[:, 0 : fnf_250]  #choose only specified dimensions
projected_matrix_fnf_250 = np.dot(Z_faces_250,Ur_faces_250)  #project training data, with specified dimensions
projected_matrix_test_fnf_250 = np.dot(Z_faces_testing_250,Ur_faces_250) #project testing data
print(projected_matrix_fnf_250)
```

```
64
[[ 1.72852298e+03  4.12309318e+02 -3.75097325e+02 ...  1.03154819e+01
  -6.95484752e+01 -2.83663441e+02]
 [-1.44228117e+03  9.20670922e+02 -5.82274094e+02 ... -3.07394731e+01
    9.34269671e+01  1.02923608e+02]
 [-4.85182690e+03 -1.21492565e+03  2.06530467e+03 ...  9.18565803e-01
   -1.60207873e+02  1.25031008e+01]
 ...
 [-2.87561178e+02 -8.82584880e+02 -1.18375353e+03 ...  1.73803580e+02
    2.04912632e+02  1.88198096e+02]
 [-2.87822056e+02 -8.36002002e+02 -1.44138490e+03 ... -3.13275522e+01
    2.23464807e+02  2.30480466e+02]
 [-2.48721670e+02 -5.69664304e+02 -1.07335601e+03 ... -3.91906062e+01
    6.99789010e+01  3.31655392e+02]]
```

```
accuracy_PCA_fnf_250 = KNN(1,projected_matrix_test_fnf_250,projected_matrix_fnf_250,train_labels_f_nf_250,test_labels_f_nf_250,faces_nf_test_250,faces_nf_train_250) #First NN is KNN
print(f'alpha = 0.9 accuracy:{accuracy_PCA_fnf_250}%')
```

```
alpha = 0.9 accuracy:96.0%
```

Size = 200

PCA for faces vs non-faces classification

```
[ ]  eigen_faces_vector ,Z_faces , eigen_faces_values= PCA(faces_nf_train)
```

```
(10304, 10304)
(10304, 10304)
True
```

Center testing data

```
[ ]  mean_testing_fnf = faces_nf_test.mean(axis=0)
     Z_faces_testing = faces_nf_test - mean_testing_fnf
```

Projection

```
[ ] fnf=dimensions(0.9,eigen_faces_values)  #get numbers of dimensions for 90% accuracy
    print(fnf)
    Ur_faces = eigen_faces_vector[:, 0 : fnf]  #choose only specified dimensions
    projected_matrix_fnf = np.dot(Z_faces,Ur_faces)  #project training data, with specified dimensions
    projected_matrix_test_fnf = np.dot(Z_faces_testing,Ur_faces) #project testing data
    print(projected_matrix_fnf)
```

```
65
[[  408.24127122 -1906.21934438 -1713.6628803  ...   -48.46902662
   -121.43746472     5.53114915]
 [ 2486.29277445  3106.73165623   852.07508351 ...  -228.56986332
   -125.5364973    -97.76279807]
 [  818.70955862   109.19950543 -1601.41504504 ...   -25.34908969
     27.93336756   -42.1873359 ]
 ...
 [   71.52562528  -856.01367408  1071.25128537 ...   170.92191868
    183.22959814  -289.87698819]
 [   67.71066271  -821.44489945  1325.35363716 ...   310.43846357
     -9.42403248  -428.57144841]
 [   82.24697791  -491.50561194  1046.0663726  ...  -213.26297743
   -357.16006995  -221.99028657]]
```

F-NN PCA Face vs Non-faces

```
[ ] accuracy_PCA_fnf = KNN(1,projected_matrix_test_fnf,projected_matrix_fnf,train_labels_f_nf,test_labels_f_nf,faces_nf_test,faces_nf_train) #First NN is KNN with k =1
    print(f'alpha = 0.9 accuracy:{accuracy_PCA_fnf}%')
```

```
---------------------------------------
alpha = 0.9 accuracy:95.75%
```

Size = 150

```
[ ]  eigen_faces_vector_150 ,Z_faces_150 , eigen_faces_values_150 = PCA(faces_nf_train_150)
```

```
(10304, 10304)
(10304, 10304)
True
```

```
[ ]  mean_testing_fnf_150 = faces_nf_test_150.mean(axis=0)
     Z_faces_testing_150 = faces_nf_test_150 - mean_testing_fnf_150
```

```
fnf_150=dimensions(0.9,eigen_faces_values_150)  #get numbers of dimensions for 90% accuracy
print(fnf_150)
Ur_faces_150 = eigen_faces_vector_150[:, 0 : fnf_150]  #choose only specified dimensions
projected_matrix_fnf_150 = np.dot(Z_faces_150,Ur_faces_150)  #project training data, with specified dimensions
projected_matrix_test_fnf_150 = np.dot(Z_faces_testing_150,Ur_faces_150) #project testing data
print(projected_matrix_fnf_150)
```

```
66
[[  -98.1243339    649.6317996   1453.79105449 ...   307.87570066
   -187.33760597   121.63417961]
 [-4355.82431414   -38.36293594  2621.46772733 ...    53.85546555
    -19.37384941   -52.10187156]
 [ -234.02281136  3407.52625717 -2382.6608183  ...   157.8688499
    -38.50145947  -505.68145759]
 ...
 [ -115.89765727  -928.95582996   716.60219182 ...   276.21125765
    -33.83648643   197.39915874]
 [ -105.22284126  -947.30463626   981.51948007 ...    38.1180554
   -150.60243034   207.64505215]
 [ -100.27699867  -569.23210105   802.38048308 ...  -156.67703457
    -80.66680424  -177.11152788]]
```

```
[ ] accuracy_PCA_fnf_150 = KNN(1,projected_matrix_test_fnf_150,projected_matrix_fnf_150,train_labels_f_nf_150,test_labels_f_nf_150,faces_nf_test_150,faces_nf_train_150) #First NN is KNN w
    print(f'alpha = 0.9 accuracy:{accuracy_PCA_fnf_150}%')
```

```
---------------------------------------
alpha = 0.9 accuracy:96.57142857142857%
```

11.7. Linear LDA Implementation

```
def binary_LDA(faces_nf_train,train_labels_f_nf):
  class_non_faces = faces_nf_train[train_labels_f_nf == 0 ]  #select data in this class
  class_faces = faces_nf_train[train_labels_f_nf == 1 ]  #select data in this class
  Mean1=class_non_faces.mean(axis=0)
  Mean2=class_faces.mean(axis=0)
  B=(Mean1-Mean2).reshape(width*height,1)
  B_t=np.transpose(B)
  print(B.shape)
  SB=[]
  SB=np.dot(B,B_t)
  print(SB.shape)

  class_non_faces_centred= class_non_faces - Mean1
  class_non_faces_t = np.transpose(class_non_faces_centred)
  class_faces_centred= class_faces - Mean2
  class_faces_t = np.transpose(class_faces_centred)
  S1 = np.dot(class_non_faces_t,class_non_faces_centred)
  S2 = np.dot(class_faces_t,class_faces_centred)
  SW = S1 + S2
  SW_inv = np.linalg.inv(SW)
  test= np.dot(SW_inv,SB)
  eigen_values,eigen_vector=np.linalg.eigh(test)
  indeces=np.argsort(eigen_values)[::-1]
  eigen_values=eigen_values[indeces]
  eigen_vector=eigen_vector[:,indeces]
  print(eigen_values.shape)
  print(eigen_vector.shape)
  return eigen_values,eigen_vector
```

The code maps to the algorithm preceding it.

It gets its mean, and centered class matrix, to get the between class scatter matrix, and within class scatter matrix, which it uses to get the corresponding eigen values and vectors for only 2 classes .

11.8. Linear LDA for each sample

Size = 300

Binary LDA

```
[ ] eigen_values_blda_300,eigen_vector_blda_300 = binary_LDA(faces_nf_train_300,train_labels_f_nf_300)
```

```
(10304, 1)
(10304, 10304)
(10304,)
(10304, 10304)
```

```
[ ] dominant_eigen_values_no_300 = dimensions(0.9,np.diag(eigen_values_blda_300))
    print(dominant_eigen_values_no_300)
```

```
1
```

Projection

```
[ ] projected_matrix_lda_fnf_300 = np.dot(Z_faces_300,eigen_vector_blda_300[:,0:dominant_eigen_values_no_300])
    projected_matrix_test_lda_fnf_300 = np.dot(Z_faces_testing_300,eigen_vector_blda_300[:,0:dominant_eigen_values_
```

F-NN faces vs non-faces

```
[ ] accuracy_LDA_fnf_300 = KNN(1,projected_matrix_test_lda_fnf_300,projected_matrix_lda_fnf_300,train_labels_f_nf_
    print(f'alpha = 0.9 accuracy:{accuracy_LDA_fnf_300}%')
```

```
---------------------------------------
alpha = 0.9 accuracy:79.0%
```

Size = 250

Binary LDA

```
[ ] eigen_values_blda_250,eigen_vector_blda_250 = binary_LDA(faces_nf_train_250,train_labels_f_nf_250)
```

```
(10304, 1)
(10304, 10304)
```

```
[ ] dominant_eigen_values_no_250 = dimensions(0.9,np.diag(eigen_values_blda_250))
    print(dominant_eigen_values_no_250)
```

Projection

```
[ ] projected_matrix_lda_fnf_250 = np.dot(Z_faces_250,eigen_vector_blda_250[:,0:dominant_eigen_values_no_250])
    projected_matrix_test_lda_fnf_250 = np.dot(Z_faces_testing_250,eigen_vector_blda_250[:,0:dominant_eigen_values_no_250])
```

F-NN faces vs non-faces

```
[ ] accuracy_LDA_fnf_250 = KNN(1,projected_matrix_test_lda_fnf_250,projected_matrix_lda_fnf_250,train_labels_f_nf_250,test_labels_f_nf_250,faces_nf_test_250,faces_nf_train_250) #First NN
    print(f'alpha = 0.9 accuracy:{accuracy_LDA_fnf_250}%')
```

Size = 200

Binary LDA for faces vs non-faces

```
[ ] eigen_values_blda,eigen_vector_blda = binary_LDA(faces_nf_train,train_labels_f_nf)
```

```
[ ] dominant_eigen_values_no = dimensions(0.9,np.diag(eigen_values_blda))
    print(dominant_eigen_values_no)
```

Projection

```
[ ] projected_matrix_lda_fnf = np.dot(Z_faces,eigen_vector_blda[:,0:dominant_eigen_values_no])
    projected_matrix_test_lda_fnf = np.dot(Z_faces_testing,eigen_vector_blda[:,0:dominant_eigen_values_no])
```

F-NN faces vs non-faces

```
[ ] accuracy_LDA_fnf = KNN(1,projected_matrix_test_lda_fnf,projected_matrix_lda_fnf,train_labels_f_nf,test_labels_f_nf,faces_nf_test,faces_nf_train) #First NN is KNN with k =1
    print(f'alpha = 0.9 accuracy:{accuracy_LDA_fnf}%')
```

Size = 150

Binary LDA

```
[ ] eigen_values_blda_150,eigen_vector_blda_150 = binary_LDA(faces_nf_train_150,train_labels_f_nf_150)
```

```
[ ] dominant_eigen_values_no_150 = dimensions(0.9,np.diag(eigen_values_blda_150))
    print(dominant_eigen_values_no_150)
```

Projection

```
[ ] projected_matrix_lda_fnf_150 = np.dot(Z_faces_150,eigen_vector_blda_150[:,0:dominant_eigen_values_no_150])
    projected_matrix_test_lda_fnf_150 = np.dot(Z_faces_testing_150,eigen_vector_blda_150[:,0:dominant_eigen_values_no_150])
```

F-NN faces vs non-faces

```
[ ] accuracy_LDA_fnf_150 = KNN(1,projected_matrix_test_lda_fnf_150,projected_matrix_lda_fnf_150,train_labels_f_nf_150,test_labels_f_nf_150,faces_nf_test_150,faces_nf_train_150) #First NN
    print(f'alpha = 0.9 accuracy:{accuracy_LDA_fnf_150}%')
```

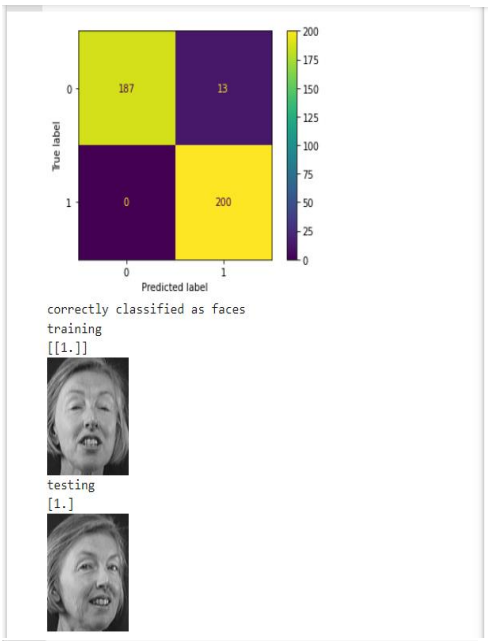
11.9. Show success failure function

```
def show_success_failure(testing_labels,neighbor_labels,testing_set,training_set,neighbor_indices):
  tp = 0
  tn = 0
  fp = 0
  fn = 0
  counter = 0
  while (not(tp and tn and fp and fn)) and counter < 100:
    random_index = np.random.choice(testing_labels.shape[0], 1, replace=False)
    if testing_labels[random_index] == neighbor_labels[random_index]:
      if testing_labels[random_index] == 1 and tp ==0 : # correctly classified as faces
        tp = 1
        tp_index = random_index
        neighbor_index = neighbor_indices[random_index]
        print("correctly classified as faces")
        display_photos(tp_index,testing_set,training_set,neighbor_index,testing_labels[random_index],neighbor_labels[random_index])
        print("-------------------------------------")
      elif testing_labels[random_index] == 0 and tn == 0:# correctly classified as non-faces
        tn = 1
        tn_index = random_index
        neighbor_index = neighbor_indices[random_index]
        print("correctly classified as non-faces")
        display_photos(tn_index,testing_set,training_set,neighbor_index,testing_labels[random_index],neighbor_labels[random_index])
        print("-------------------------------------")
    elif testing_labels[random_index] == 0 and fp == 0: # incorrectly classified as faces
      fp = 1
      fp_index = random_index
      neighbor_index = neighbor_indices[random_index]
      print("incorrectly classified as faces")
      display_photos(fp_index,testing_set,training_set,neighbor_index,testing_labels[random_index],neighbor_labels[random_index])
      print("-------------------------------------")
    elif testing_labels[random_index] == 1 and fn == 0: # incorrectly classified as non-faces
      fn = 1
      fn_index = random_index
      neighbor_index = neighbor_indices[random_index]
      print("incorrectly classified as non-faces")
      display_photos(fn_index,testing_set,training_set,neighbor_index,testing_labels[random_index],neighbor_labels[random_index])
      print("-------------------------------------")
    counter = counter + 1
```

The show_success_failure function takes in the labels and predicted neighbor labels of the testing set, along with the testing and training sets, and the indices of the nearest neighbors for each test instance. It then randomly selects a test instance and checks whether it was correctly classified as a face or non-face by comparing the true label with the predicted neighbor label. It prints out the result and displays the images of the test instance, its nearest neighbor, and the true and predicted labels. The function stops after displaying one correctly classified face and one correctly classified non-face, and one incorrectly classified face and one incorrectly classified non-face, or after 100 iterations.

11.10. Visualizing True positive, True negative, False positive, and False negative cases as well as confusion matrices.

PCA, non face sample size 200



correctly classified as faces
training
[[1.]]

testing
[1.]

---

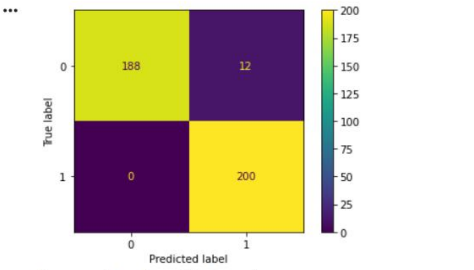
correctly classified as non-faces
training
[[0.]]

testing
[0.]

---

incorrectly classified as faces
training
[[1.]]

testing
[0.]



incorrectly classified as faces
training
[[1.]]

testing
[0.]

correctly classified as non-faces
training
[[0.]]

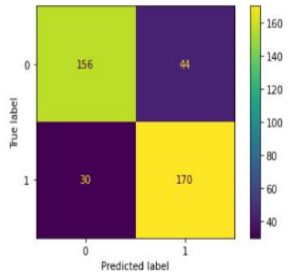
testing
[0.]

---

correctly classified as faces
training
[[1.]]

testing
[1.]

LDA, non face sample 200



correctly classified as faces
training
[[1.]]

testing
[1.]

incorrectly classified as faces
training
[[1.]]

testing
[0.]

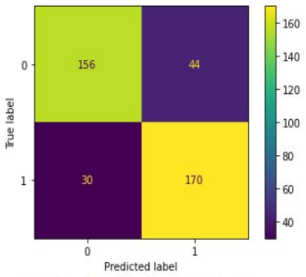
---

incorrectly classified as non-faces
training
[[0.]]

testing
[1.]

---

alpha = 0.9 accuracy:81.5%



correctly classified as non-faces
training
[[0.]]

testing
[0.]

incorrectly classified as non-faces
training
[[0.]]

testing
[1.]

---

correctly classified as faces
training
[[1.]]

testing
[1.]

incorrectly classified as faces
training
[[1.]]

testing
[0.]

---

alpha = 0.9 accuracy:81.5%

correctly classified as non-faces
training
[[0.]]

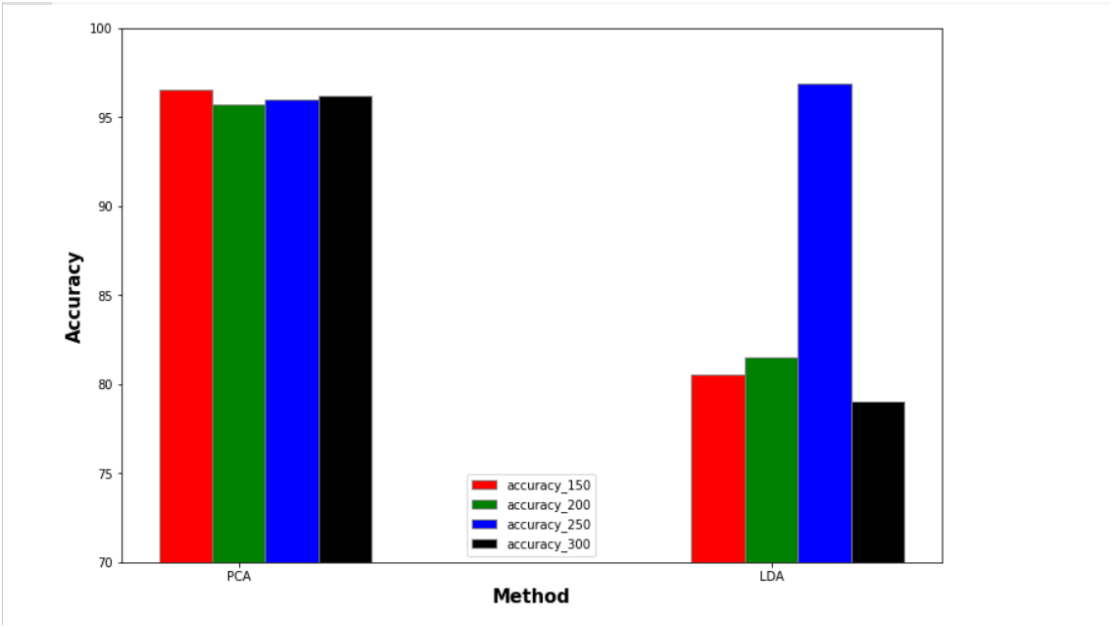
testing
[0.]

11.11. display photos function

```
def display_photos(index,testing_set,training_set,neighbor_index,testing_label,neighbor_label):
  face_nf_image_train=np.array(training_set[neighbor_index]).reshape(112,92)
  img_train = Image.fromarray( face_nf_image_train)
  face_nf_image_test=np.array(testing_set[index]).reshape(112,92)
  img_test = Image.fromarray( face_nf_image_test)
  # Display the image
  print("training")
  print(neighbor_label)
  img_train.show()
  print("testing")
  print(testing_label)
  img_test.show()
```

This function displays a pair of images one from the training set and one from the testing set. The image from the training set is selected as the nearest neighbor of the image from the testing set. The function takes the index of the testing image, the testing set, the training set, the index of the nearest neighbor, the label of the testing image, and the label of the nearest neighbor as inputs. The images are first reshaped from 1D vectors to 2D arrays and then converted into PIL image objects. Finally, the images are displayed.

11.12. Plot the accuracy vs the number of non-faces images while fixing the number of face images.



11.13. Criticizing the accuracy measure for large numbers of non-faces images in the training data.

The variation of the number of samples taken in the training set in PCA results in almost the same accuracy percentage. However, in LDA when taking 150 samples it resulted in a low accuracy percentage by increasing the number of samples to 200 the accuracy started to increase reaching the highest accuracy percentage by taking 250 samples and then dropped to an extreme low accuracy percentage by taking 300 samples.

BONUS

12.Using different Training and Test splits(70-30)

12.1. Getting the new indices and calling split function.

Different Split of Dataset into Training and Testing sets (70-30)

```
[21] training_set_indices_new = []
     testing_set_indices_new = []
     for i in range(0,subjects_no * images_per_subject, 10) :
       for j in range(i,i+7) :
         training_set_indices_new.append(j)
       for j in range(i+7,i+10) :
         testing_set_indices_new.append(j)
     print(training_set_indices_new)
     print(testing_set_indices_new)
     training_set_new, testing_set_new, training_labels_new, testing_labels_new  = split_data(D,y,training_set_indices_new,testing_set_indices_new)
```

```
[0, 1, 2, 3, 4, 5, 6, 10, 11, 12, 13, 14, 15, 16, 20, 21, 22, 23, 24, 25, 26, 30, 31, 32, 33, 34, 35, 36, 40, 41, 42, 43, 44, 45, 46, 50, 51, 52
[7, 8, 9, 17, 18, 19, 27, 28, 29, 37, 38, 39, 47, 48, 49, 57, 58, 59, 67, 68, 69, 77, 78, 79, 87, 88, 89, 97, 98, 99, 107, 108, 109, 117, 118, 1
[[ 48.  49.  45. ...  47.  46.  46.]
 [ 60.  60.  62. ...  32.  34.  34.]
 [ 39.  44.  53. ...  29.  26.  29.]
 ...
 [128. 125. 125. ...  85.  90.  84.]
 [123. 121. 126. ...  40.  35.  42.]
 [129. 127. 133. ...  93.  93.  93.]]
(120, 10304)
(120,)
(280,)
```

12.2. Calling the PCA function to get eigen vectors for training data

PCA New

```
[ ]  U_new ,Z_new , eigen_values_new= PCA(training_set_new)
```

```
(10304, 10304)
(10304, 10304)
True
```

12.3. Centering testing data to be projected

Centering new testing data

```
[ ]  mean_testing_new = testing_set_new.mean(axis=0)
     Z_testing_new = testing_set_new - mean_testing_new
```

12.4. Getting projection matrix U, and projecting the data

```
R5=dimensions(0.9,eigen_values_new)
print(R5)
Ur_new = U_new[:, 0 : R5]
projected_matrix_R5 = np.dot(Z_new,Ur_new)
projected_matrix_test_R5 = np.dot(Z_testing_new,Ur_new)
```

12.5. F-NN for alpha = 0.9

```
accuracy_PCA_R5 = KNN(1,projected_matrix_test_R5,projected_matrix_R5,training_labels_new,testing_labels_new) #First NN is KNN with k =1
print(f'alpha = 0.9 accuracy:{accuracy_PCA_R5}%')
```

12.6. Calling the LDA function to get eigen vectors for training data

LDA new

```
eigen_vector_no = 39
U_lda_new = LDA(training_set_new,training_labels_new,subjects_no,width,height,eigen_vector_no)
```

```
(10304, 10304)
```

12.7. Getting projection matrix U, and projecting the data

Projection new

```
[ ]  projected_matrix_lda_new = np.dot(Z_new,U_lda_new)
     projected_matrix_test_lda_new = np.dot(Z_testing_new,U_lda_new)
```

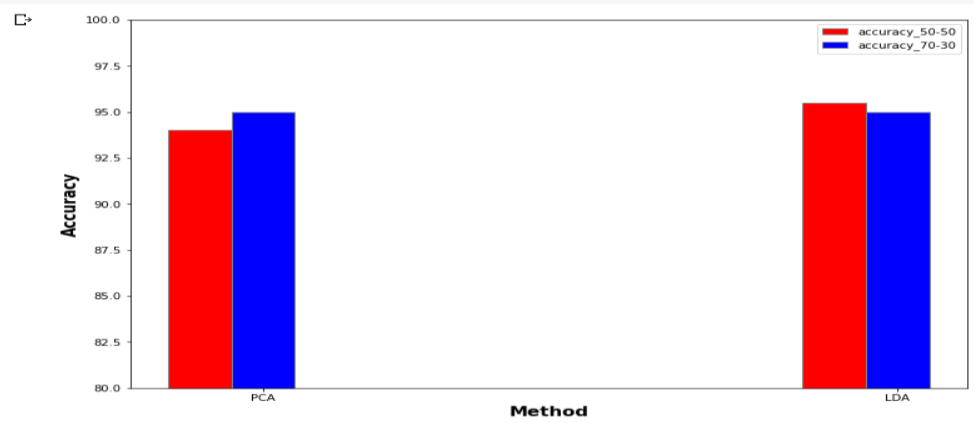
12.8. F-NN for alpha = 0.9

First-NN LDA New

```
[ ] accuracy_LDA_new = KNN(1,projected_matrix_test_lda_new,projected_matrix_lda_new,training_labels_new,testing_labels_new) # First-NN is KNN with k =1
    print(f'Accuracy LDA New : {accuracy_LDA_new}% ' )
```

```
Accuracy LDA New : 95.0%
```

12.9. compare the results you have with the ones got earlier with 50% split.



13. other variation of PCA beyond original algorithms

13.1. What is kernel PCA?

PCA is limited to linear transformations and is performed to identify the principal components of the data. It may not be effective for datasets with nonlinear relationships between the features. KPCA addresses this limitation by using a nonlinear kernel function to map the data into a higher-dimensional feature space, where the nonlinear relationships between the features can be more easily captured by a linear PCA. It is useful for datasets with nonlinear relationships between the features and has applications in various fields.

13.2. Kernel PCA vs original PCA

PCA is useful for linear datasets and is computationally efficient, while KPCA is useful for nonlinear datasets but can be more computationally expensive. The choice between PCA and KPCA depends on the nature of the data and the computational resources available.

13.3. Kernel PCA function

This is a Python function that performs Kernel PCA on a given training set using a Gaussian kernel with a specified gamma value. The function first calculates the mean of the training set and centralizes the data by subtracting the mean from each data point. It then calculates the Euclidean distances between all pairs of points in the training set and uses these distances to construct a Gaussian kernel matrix with the specified gamma value. The kernel matrix is then centered and its eigenvectors and eigenvalues are calculated using the linalg.eigh function. The eigenvectors and eigenvalues are sorted in descending order and returned as the principal components and corresponding eigenvalues of the kernel PCA. The function also returns the centralized training set and the diagonal matrix of eigenvalues for further analysis.

Kernel PCA

gamma = 1 / training_set.shape[1] # default gamma is 1 / n_features

```
def kernel_pca(training_set,gamma):
  mean_vector = training_set.mean(axis=0)
  Z_kernel = training_set - mean_vector #centralize data

  # Calculate euclidean distances of each pair of points in the data set
  dist = euclidean_distances(training_set, training_set, squared=True)

  # Calculate Gaussian kernel matrix
  K = np.exp(-gamma * dist)
  # Center the kernel matrix
  n = K.shape[0]
  one_n = np.ones((n, n)) / n
  K = K - np.dot(one_n, K) - np.dot(K, one_n) + np.dot(one_n, np.dot(K, one_n))

  eigen_values, U_kernel = np.linalg.eigh(K)
  indices = np.argsort(eigen_values)[::-1] #sort descendingly
  eigen_values_vector = eigen_values[indices]
  eigen_values_kernel = np.diag(eigen_values_vector)
  U_kernel = U_kernel[:,indices]
  print(U_kernel.shape)
  print(eigen_values.shape)
  return U_kernel , Z_kernel , eigen_values_kernel
```

13.4. Comparing the time complexity and accuracy between the 2 different PCA models

Kernel PCA

```
[ ] start_kernel = time.time()
    U_kernel , Z_kernel , eigen_values_kernel = kernel_pca(training_set,1/(width*height)) #gamma =  1/no. of features
    end_kernel = time.time()
    runtime_kernel = end_kernel - start_kernel
    print(runtime_kernel)
    U_test_kernel , Z_test_kernel , eigen_values_test_kernel = kernel_pca(testing_set,1/(width*height)) #gamma =  1/no. of features
```

```
(200, 200)
(200,)
0.061191558837890625
(200, 200)
(200,)
```

```
[ ] kernel_dimensions=dimensions(0.9,eigen_values_kernel)
    print(kernel_dimensions)
    projected_matrix_kernel = U_kernel[:, 0 : kernel_dimensions]

    projected_matrix_test_kernel = U_test_kernel[:, 0 : kernel_dimensions]
```

```
180
```

F-NN Kernel PCA , First NN is KNN with k =1

```
[ ] accuracy_kernel = KNN(1,projected_matrix_test_kernel,projected_matrix_kernel,training_labels,testing_labels) #First NN is KNN with k =1
    print(f'alpha = 0.9 accuracy:{accuracy_kernel}%')
```

```
alpha = 0.9 accuracy:100.0%
```

	Time Complexity	Accuracy
PCA	540.4974234104156	94.0%
Kernel PCA	0.061191558837890625	100.0%

14. Other variation of LDA beyond original algorithms

14.1. What is Regularized LDA?

Regularized LDA is a modified version of LDA that incorporates regularization to address singularity and instability issues in high-dimensional data. LDA is a technique used for dimensionality reduction and classification that assumes equal covariance matrices in each class, which may not be true in high-dimensional settings. Regularized LDA introduces a penalty term to the covariance matrix in the objective function, allowing for a trade-off between bias and variance and better performance on high-dimensional data. Regularized LDA is widely used in fields like image recognition, speech processing, and bioinformatics, where high-dimensional data is common.

14.2. Regularized LDA vs original LDA

Regularized LDA is a modified version of LDA that addresses limitations of the original LDA in high-dimensional settings by introducing a regularization term to the covariance matrix. This regularization term helps to overcome singularity and instability issues, and allows for a trade-off between bias and variance by controlling a regularization parameter. The result is better classification performance on high-dimensional data, making Regularized LDA a useful tool in fields such as image recognition, speech processing, and bioinformatics.

14.3. Regularized LDA function

The Regularized LDA algorithm is used for dimensionality reduction and classification of high-dimensional data. The code takes a training set and labels, computes the within-class and between-class scatter matrices, adds a regularization term to the within-class scatter matrix to prevent overfitting, and computes the eigenvalues and eigenvectors of the product of the inverse of the within-class scatter matrix and the between-class scatter matrix. The sorted eigenvectors and eigenvalues are returned as the output.

Regularized LDA

```
[18] def regularized_LDA(training_set, training_labels, subjects_no, width, height, alpha):
         total_mean = training_set.mean(axis=0)
         Sb = np.zeros((width*height, width*height))
         Sw = np.zeros((width*height, width*height))
         class_mean = np.empty([subjects_no, width*height])
         for id in range(1, subjects_no+1):
             class_matrix = training_set[training_labels == id] # select data in this class
             samples_no_per_class = class_matrix.shape[0]
             class_mean[id-1] = class_matrix.mean(axis=0)
             class_Z = class_matrix - class_mean[id-1]
             class_Z_transpose = np.transpose(class_Z)
             Si = np.dot(class_Z_transpose, class_Z)
             Sw = Sw + Si

             mean_diff = (class_mean[id-1] - total_mean).reshape(width*height, 1)
             mean_diff_transpose = np.transpose(mean_diff)
             Sb = Sb + samples_no_per_class * np.dot(mean_diff, mean_diff_transpose)

         Sw = Sw + alpha*np.eye(width*height) # add regularization term
         Sw_inv = np.linalg.inv(Sw)
         Sw_inv_Sb = np.dot(Sw_inv, Sb)
         lamda_eigen_values, w = np.linalg.eigh(Sw_inv_Sb)
         lamda_indices = np.argsort(lamda_eigen_values)[::-1] # sort descendingly
         lamda_eigen_values = lamda_eigen_values[lamda_indices]
         w = w[:, lamda_indices]
         return w, lamda_eigen_values
```

Regularized LDA

```
[20] start_regularized_LDA = time.time()
     alpha = 0.25
     U_regularized_LDA,eigen_values_regularized_LDA = regularized_LDA(training_set, training_labels, subjects_no, width, height, alpha)
     end_regularized_LDA = time.time()
     runtime_regularized_LDA = end_regularized_LDA - start_regularized_LDA
     print(runtime_regularized_LDA)
```

```
639.081925868988
```

```
eigen_values_regularized_LDA_no = dimensions(0.9,np.diag(eigen_values_regularized_LDA))
print(eigen_values_regularized_LDA_no)
```

```
5
```

```
[22] U_regularized_LDA = U_regularized_LDA[:, eigen_values_regularized_LDA_no]
```

Projection

```
[27] projected_matrix_regularized_LDA = np.dot(Z,U_regularized_LDA)
     projected_matrix_test_regularized_LDA = np.dot(Z_testing,U_regularized_LDA)
```

F-NN Regularized LDA

```
accuracy_regularized_LDA = KNN(1,projected_matrix_test_regularized_LDA,projected_matrix_regularized_LDA,training_labels,testing_labels)
print(f'alpha = 0.9 accuracy:{accuracy_regularized_LDA}%')
```

```
This face is correctly classified to its class
training
[[28.]]
```

```
testing
[28.]
```

This face is correctly classified to its class
training
[[7.]]

testing
[7.]

This face is correctly classified to its class
training
[[36.]]

testing
[36.]

---
alpha = 0.9 accuracy:90.5%

14.4. Comparing the time complexity and accuracy between the 2 different LDA models

	Time Complexity	Accuracy
LDA	520.5594136714935	95.5%
Regulized LDA	563.5903015136719	90.5%