Face Recognition Using PCA - LDA

Pattern Recognition
Assignment (1)

Lara Hossam Eldine Mostafa

ID: 6853

Mohamed Alaa ElZeftawy

ID: 6886

Table of Contents

Table of Contents 2
Problem statement: 3
1. Download the Dataset and Understand the Format (10 points) 3
2. Generate the Data Matrix and the Label Vector (10 points) 4
3. Split the Dataset into Training and Testing Sets (10 points) 4
4. Classification using PCA (30 points) 4
5. Classification using LDA(30 points) 6
6. Classifier Tuning (20 points) 8
7. Compare vs Non-Face Images (15 Points) 9
8. Bonus - Different Train-Test splits (5 Points) 11
9. Bonus - Variations of PCA 11
10. Bonus - Variations of LDA 12

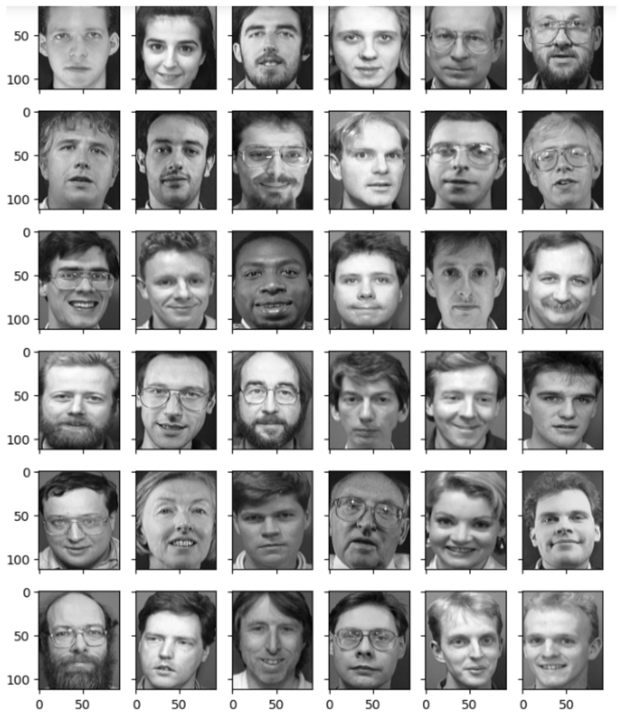
Problem statement:

We intend to perform face recognition. Face recognition means that for a given image you can tell the subject ID amongst 40 subjects.

1. Download the Dataset and Understand the Format (10 points)

The dataset available on https://www.kaggle.com/kasikrit/att-database-of-faces/ consists of 400 images, 10 images per subject, with subject IDs ranging from 1 to 40. The images are provided in a **.pgm** format which means **portable gray map.**

The PGM (Portable Gray Map) format is a common image file format used to store grayscale images. PGM files consist of a header and image data. The header contains information about the image such as its size, color depth, and other metadata. The image data follows the header and consists of a matrix of pixel values.



Using numpy and opencv libraries in python, we were able to decode the .**pgm** format into a dictionary where each key corresponds to **subject_ID/sample_number** and value corresponds to a **(112,92)** numpy array of pixel values with numbers ranging from 0-255 as the uint=8 parameter was passed.

Exploratory Data Analysis:

Exploring how samples from different subject IDs look like by plotting one picture out of each 10 successive ones.

Plotting again the images, but this time seeing the variations of a single subject.

2. Generate the Data Matrix and the Label Vector (10 points)

As we have 400 samples, each sample consisting of 112x92 = 10304 pixels, we convert each image to a 1x10304 vector of features. We add corresponding label in a separate array and stack all the samples together horizontally to form a (400x10304) data set and (400x1) labels column array.

3. Split the Dataset into Training and Testing Sets (10 points)

We splitted the dataset into 50% for training and 50% for testing, by keeping the odd rows for training and even rows for testing. Now, test_x and train_x have the shape (200, 10304) and test_y and train_y have the shape (200,1).

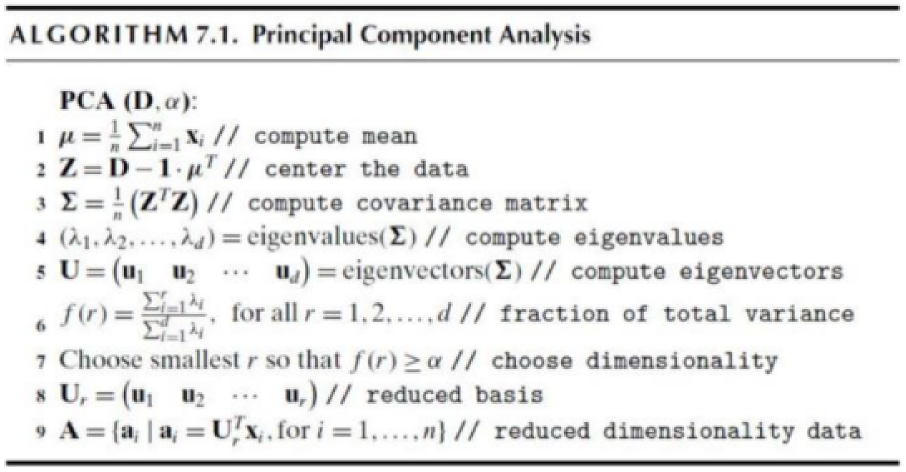
4. Classification using PCA (30 points)

PCA (Principal Component Analysis) is a popular technique used in facial recognition (classification) because it can effectively reduce the dimensionality of high-dimensional facial images, while preserving the most important features that are critical for recognizing faces. In other words, it can help to extract the most useful information from large sets of facial data, and make it easier to distinguish between different faces.

PCA works by finding **the linear combinations** of the original features that best explain the variation in the data. These linear combinations are called **principal components**, and they are ordered in terms of their **ability to capture the most variation in the data**. By retaining only the top k principal components, PCA can effectively reduce the dimensionality of the data from n to k, where k << n, without significant loss of information.

From 10304 components to a much smaller number determined by the degree of variance we want to capture, denoted by **alpha**, that we will set with different values and compute PCA each time. One common approach is to use PCA to extract a set of **eigenfaces** from a large database of facial images. Eigenfaces are essentially the principal components of the facial data, and they can be thought of as templates that capture the most important facial features. These eigenfaces can then be used to represent new facial images in a lower-dimensional space, where they can be more easily compared and classified.

This is the algorithm we will follow:



ALGORITHM 7.1. Principal Component Analysis

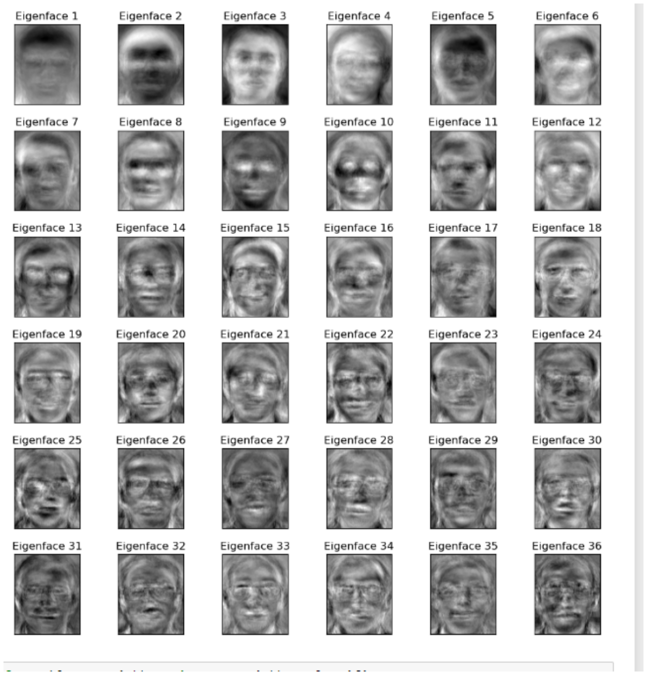
```
PCA ($\mathbf{D}, \alpha$):
1 $\mu = \frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_i$ // compute mean
2 $\mathbf{Z} = \mathbf{D} - \mathbf{1}\cdot\mu^T$ // center the data
3 $\mathbf{\Sigma} = \frac{1}{n}(\mathbf{Z}^T\mathbf{Z})$ // compute covariance matrix
4 $(\lambda_1, \lambda_2, \ldots, \lambda_d) = \text{eigenvalues}(\mathbf{\Sigma})$ // 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
```

1) Compute the **mean** of each feature resulting in an array of dimensions (1,10304)
2) Center the data around the mean, resulting matrix Z of dimensions (200,10304)
3) Compute **covariance** matrix using the formula (Z.T.dot(Z))/200 of shape (10304, 10304)
4) Compute **eigenvectors and eigenvalues** of covariance matrix resulting in a (10304,10304) matrix of eigenvectors and (10304,1) column vector for eigenvalues.
5) Sort the eigen values descendingly, and choosing the number of eigen values to choose to achieve the fraction **alpha** we desire from the total variance, it's then when we choose the dimensionality reduced.

   For the alphas [0.8,0.85,0.9,0.95], the corresponding number of eigen vectors to consider is [36, 52, 76, 117]

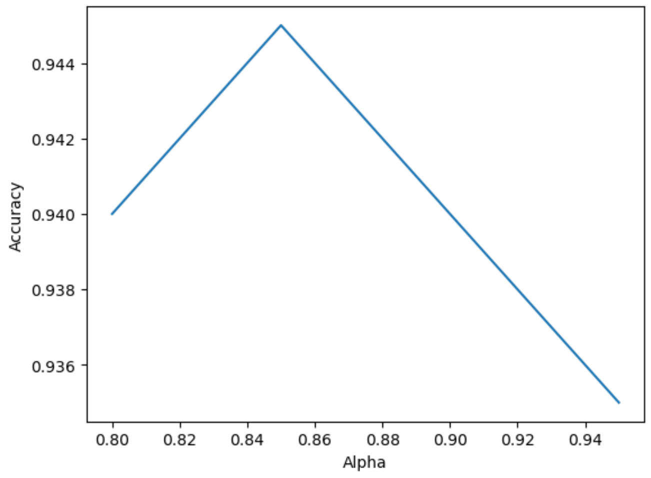
6) Reduce dimensionality by projecting the data on the new dimensions, by taking the dot product of the new eigen vector matrix and the data matrix.

We plotted the eigen vectors corresponding to the eigen faces to visualize the process:



7) We ran a KNN classifier with number of neighbors = 1, for 4 times each time corresponding to a specific alpha, and at each alpha the data is projected on the corresponding vector of dimensions reduced.

KNN is a simple yet effective algorithm for facial recognition that can classify facial images by comparing their feature vectors to those of labeled images in a training dataset. By selecting the k-nearest neighbors based on a distance metric, the algorithm can make accurate predictions on the identities of new facial images, even in the presence of noise and variability.



Accuracies corresponding to alphas : {0.8: 0.94, 0.85: 0.945, 0.9: 0.94, 0.95: 0.935}

We notice from the following plot that the accuracy is high and fluctuates when alpha is increased, but it decreases slowly, we can notice that the alpha with 0.85 produced the best results. By adding more eigenfaces we are adding more features of less significance so they reduce the accuracy of the recognition system as it becomes more sensitive to noise. The previous case might lead to a popular phenomenon called overfitting where the model is over trained to the train dataset but can't predict test cases correctly.

5. Classification using LDA(30 points)

LDA is a linear transformation technique that projects a dataset onto a lower-dimensional space, while preserving as much of the class-discriminatory information as possible. In other words, LDA aims to find a new set of features that best separates different classes of data. LDA is similar to Principal Component Analysis (PCA), but whereas PCA seeks to find the axes that explain the most variance in the data, LDA seeks to find the axes that best separate the classes. The LDA algorithm works by finding the eigenvectors of the covariance matrix of the data, and then projecting the data onto the space defined by these eigenvectors. The projection is such that the **between-class scatter is maximized, while the within-class scatter is minimized.** This means that LDA seeks to find a set of features that maximizes the ratio of the between-class variance to the within-class variance.

This is the algorithm we will follow with some modifications where we take into consideration the multi-class case, The between class scatter matrix will be a sum over the 40 classes multiplied by the number of samples in each class:

$$S_b = \sum^{m} n_k(\mu_k - \mu)(\mu_k - \mu)^T$$

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 $\mu_i \leftarrow \text{mean}(\mathbf{D}_i), i = 1, 2$ // class means
3 $\mathbf{B} \leftarrow (\mu_1 - \mu_2)(\mu_1 - \mu_2)^T$ // between-class scatter matrix
4 $\mathbf{Z}_i \leftarrow \mathbf{D}_i - \mathbf{1}_{n_i}\mu_i^T, i = 1, 2$ // 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
```

1) Compute the **class mean** of each class-specific subset and saving them in a matrix (40,10304)
2) Compute **between class scatter matrix** from the formula above, and through each iteration adding the result to the matrix Sb, its final dimensions will be (10304,10304)
3) **Center the data** by subtracting the mean from each class
4) Compute **within class scatter matrices S** and summing them together
5) Computing the eigenvalues and eigenvalues of the matrix obtained by S_inv.dot(Sb)
6) Choosing the first 39 dominant eigenvectors corresponding to the 39 dominant eigenvalues and projecting the data
7) We ran a KNN classifier with number of neighbors = 1.

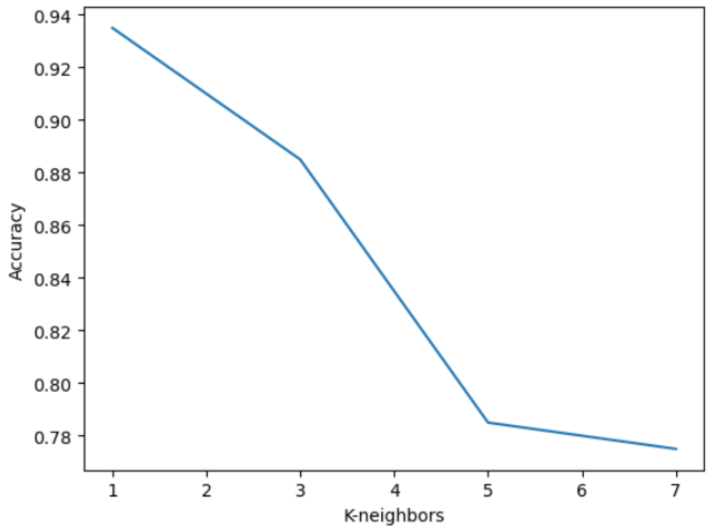
```
K = 1
Accuracy: 0.95
```

Comparing results to PCA: with a much lesser number of eigenvectors, LDA using KNN, k=1 achieved very close results to the PCA results when alpha was equal to 0.95 This shows how LDA works better than PCA in separating classes.

6. Classifier Tuning (20 points)

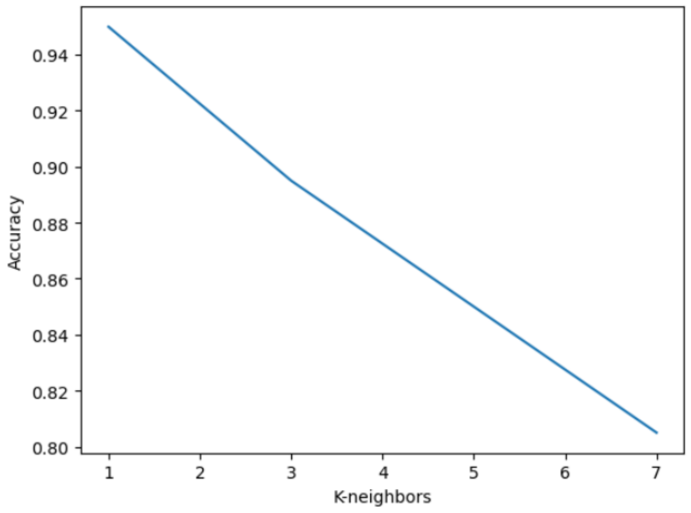
Trying the KNN algorithm with different number of neighbors, and with a range of alphas, these are the PCA results [plotting the accuracies of alpha: 0.95]:

```
K = 1
{0.8: 0.94, 0.85: 0.945, 0.9: 0.94, 0.95: 0.935}
K = 3
{0.8: 0.84, 0.85: 0.86, 0.9: 0.875, 0.95: 0.885}
K = 5
{0.8: 0.815, 0.85: 0.795, 0.9: 0.8, 0.95: 0.785}
K = 7
{0.8: 0.765, 0.85: 0.785, 0.9: 0.765, 0.95: 0.775}
[0.935, 0.885, 0.785, 0.775]
```



Varying the number of neighbors with the LDA algorithm:

```
K = 1
Accuracy=0.95
K = 3
Accuracy=0.895
K = 5
Accuracy=0.85
K = 7
Accuracy=0.805
[0.95, 0.895, 0.85, 0.805]
```



Tie-breaking:

The default tie-breaking strategy of sklearn KNeighbors classifiers that chooses the element that's present first in the training data when identical distances occur.

To avoid this issue, it is recommended to shuffle the training data before fitting the KNN model to ensure that the ordering of the training data does not affect the results. Additionally, using an odd value of k can help to reduce the chances of ties occurring in the first place.

Observations:

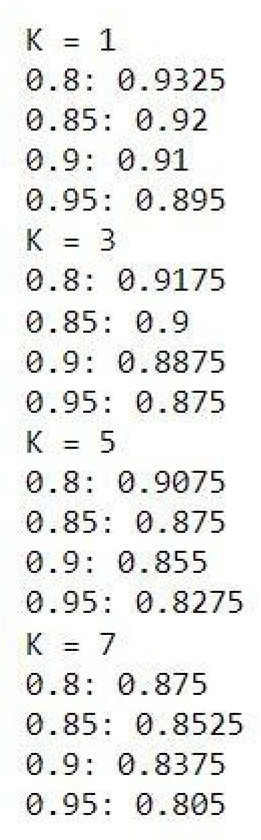
We notice that when the number of neighbors gets higher, the accuracy decreases quickly, which is logical as we only have 5 samples per class, if the K exceeds 5, the error will definitely go up. The best performance with this test-train split is k=1

7. Compare vs Non-Face Images (15 Points)

We got a new dataset of images that are non-face, including cars, cats, dogs, flowers, airplanes and nature. We resized them to be of size 92x112 and grayscale to match the initial faces dataset. We have 400 images of non faces labeled 0 and 400 other faces images labeled 1.

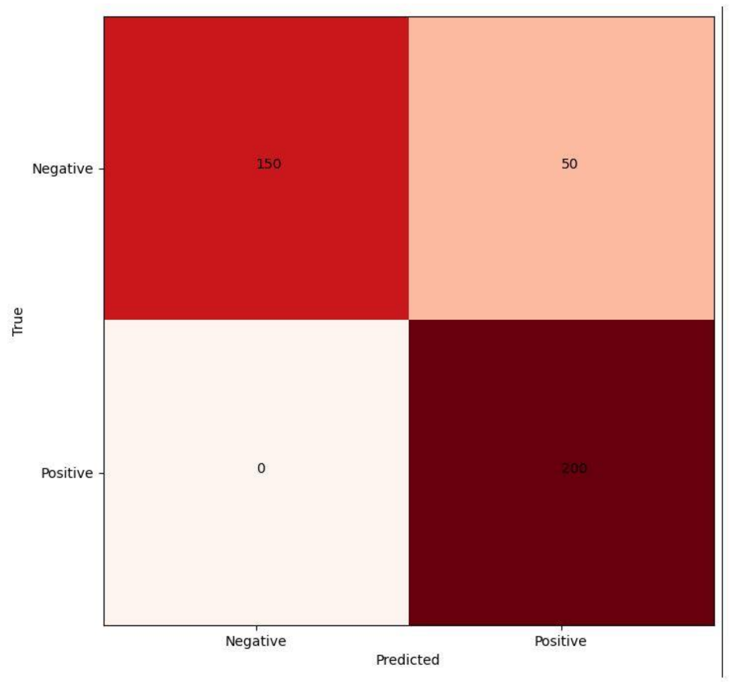
We splitted the data in the same way, odd rows for training in the non faces dataset and even rows for testing.

<u>Using PCA, the results are as follows varying the alpha and number of neighbors:</u>

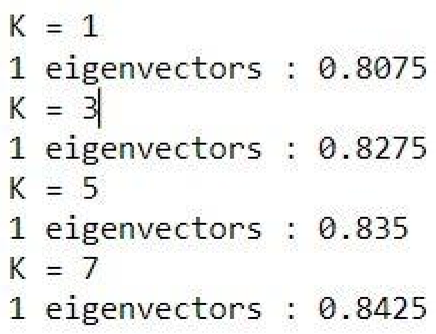


```
K = 1
0.8: 0.9325
0.85: 0.92
0.9: 0.91
0.95: 0.895
K = 3
0.8: 0.9175
0.85: 0.9
0.9: 0.8875
0.95: 0.875
K = 5
0.8: 0.9075
0.85: 0.875
0.9: 0.855
0.95: 0.8275
K = 7
0.8: 0.875
0.85: 0.8525
0.9: 0.8375
0.95: 0.805
```

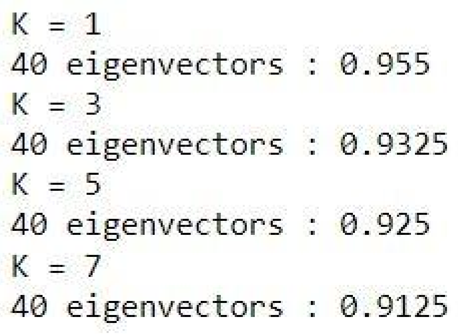
<u>Confusion Matrix K = 1, alpha = 0.9</u>



Using LDA, the results are as follows:



```
K = 1
1 eigenvectors : 0.8075
K = 3
1 eigenvectors : 0.8275
K = 5
1 eigenvectors : 0.835
K = 7
1 eigenvectors : 0.8425
```

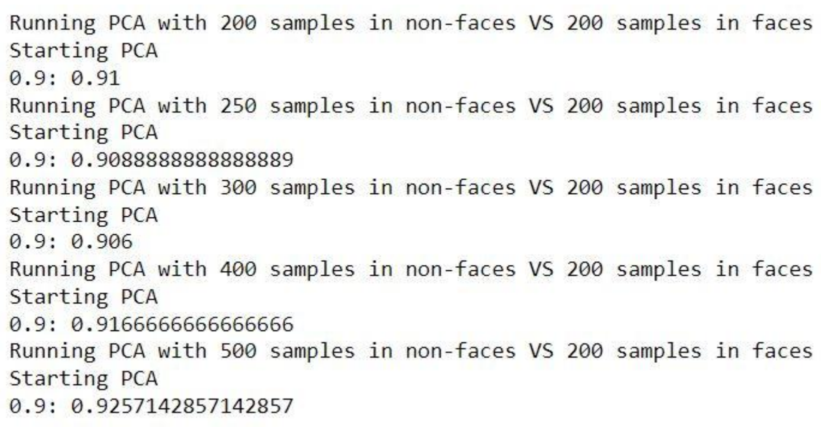


```
K = 1
40 eigenvectors : 0.955
K = 3
40 eigenvectors : 0.9325
K = 5
40 eigenvectors : 0.925
K = 7
40 eigenvectors : 0.9125
```

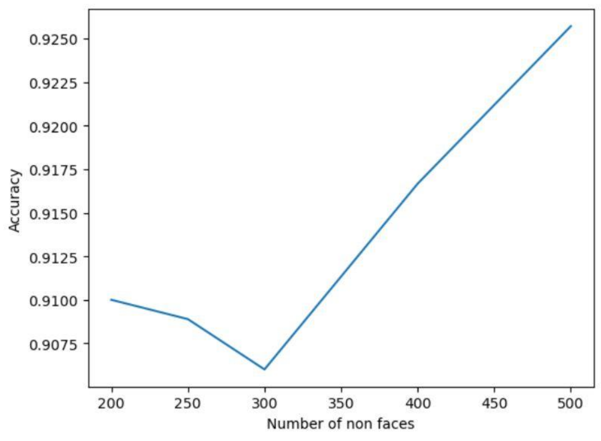
- Theoretically, the optimal number of eigenvectors to choose in the LDA is **1 eigenvector** as the optimal number of eigenvectors is (**number of classes -1)**, but as we are dealing with images, we noticed that it's more convenient to use eigenvectors varying from 39 to 50.
- While varying the number of input non faces we find that the accuracy decreases with the increase of the number of non faces.
- The main reason for this is that LDA assumes that the input features are normally distributed, which may not be the case for image data. Images typically have a high-dimensional feature space, with each pixel or group of pixels representing a feature. These features are often highly correlated, and their distribution may be far from normal. In addition, LDA is a linear technique, which means that it may not be able to capture the complex nonlinear relationships between image features that are necessary for effective image classification. Another issue with LDA for image classification is that images can vary greatly in terms of their content, composition, and lighting conditions, which can make it difficult to identify common patterns that are representative of a particular class. This can lead to poor performance when attempting to classify images based on a limited set of features.

Varying the number of samples of non-face images and testing results:

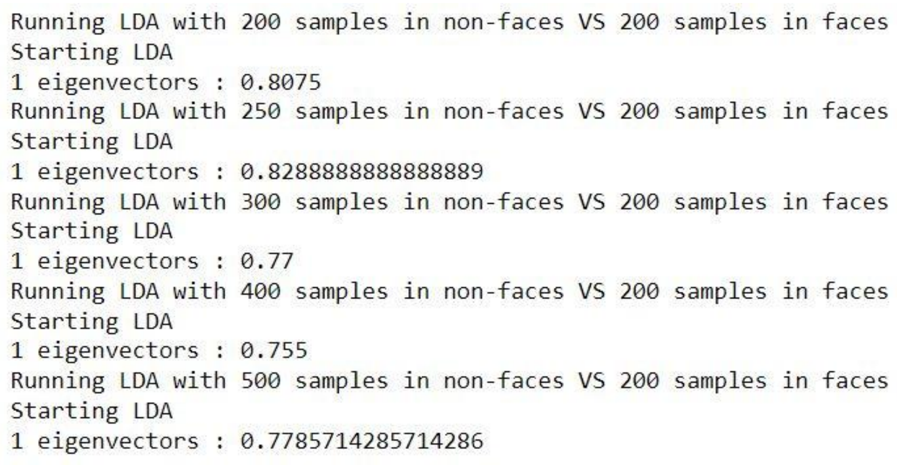
PCA:



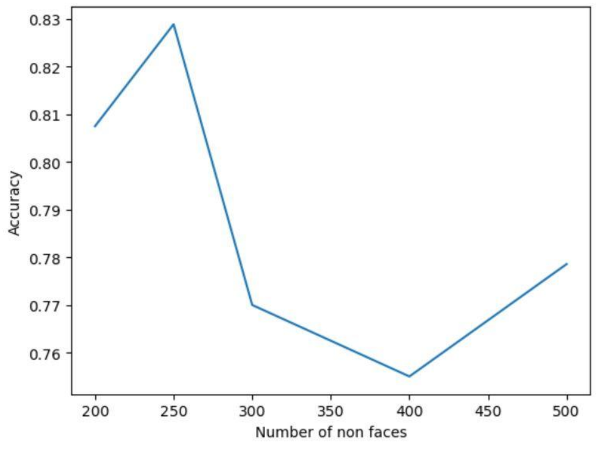
```
Running PCA with 200 samples in non-faces VS 200 samples in faces
Starting PCA
0.9: 0.91
Running PCA with 250 samples in non-faces VS 200 samples in faces
Starting PCA
0.9: 0.9088888888888889
Running PCA with 300 samples in non-faces VS 200 samples in faces
Starting PCA
0.9: 0.906
Running PCA with 400 samples in non-faces VS 200 samples in faces
Starting PCA
0.9: 0.9166666666666666
Running PCA with 500 samples in non-faces VS 200 samples in faces
Starting PCA
0.9: 0.9257142857142857
```



LDA:



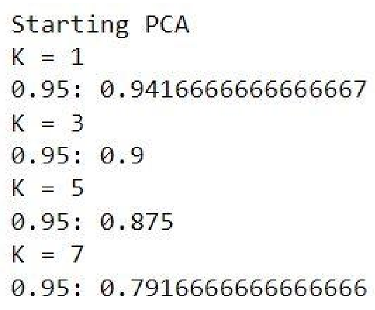
```
Running LDA with 200 samples in non-faces VS 200 samples in faces
Starting LDA
1 eigenvectors : 0.8075
Running LDA with 250 samples in non-faces VS 200 samples in faces
Starting LDA
1 eigenvectors : 0.8288888888888889
Running LDA with 300 samples in non-faces VS 200 samples in faces
Starting LDA
1 eigenvectors : 0.77
Running LDA with 400 samples in non-faces VS 200 samples in faces
Starting LDA
1 eigenvectors : 0.755
Running LDA with 500 samples in non-faces VS 200 samples in faces
Starting LDA
1 eigenvectors : 0.7785714285714286
```



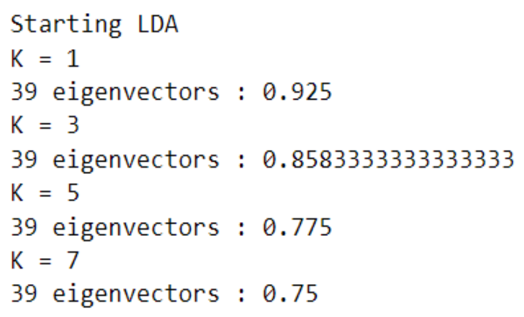
Observation: PCA is better than LDA in this task and performs better when the number of samples is more so that it can capture more variance between features. LDA was better in the 40 class classification because it works best in class separability.

8. Bonus - Different Train-Test splits (5 Points)

Using different train-test splits, we found that the 70-30 test split was way better.



```
Starting PCA
K = 1
0.95: 0.9416666666666667
K = 3
0.95: 0.9
K = 5
0.95: 0.875
K = 7
0.95: 0.7916666666666666
```



```
Starting LDA
K = 1
39 eigenvectors : 0.925
K = 3
39 eigenvectors : 0.8583333333333333
K = 5
39 eigenvectors : 0.775
K = 7
39 eigenvectors : 0.75
```

9. Bonus - Variations of PCA

Using variations of PCA, we can use **IPCA** [Incremental PCA]:

It is an implementation of Principal Component Analysis (PCA) using **Singular Value Decomposition (SVD)** for **incremental batch processing of large datasets**.

It keeps only the most significant singular vectors to project the data to a lower dimensional space. The input data is centered but not scaled for each feature before applying the SVD.

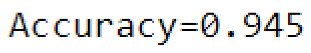
Depending on the size of the input data, this algorithm can be much more memory efficient than a PCA, and allows sparse input.

This algorithm has **constant memory complexity**, on the order of **batch_size * n_features**, enabling use of np.memmap files without loading the entire file into memory. For sparse matrices, the input is converted to dense in batches (in order to be able to subtract the mean) which avoids storing the entire dense matrix at any one time.

The computational overhead of each SVD is O(batch_size * n_features ** 2), but only 2 * batch_size samples remain in memory at a time. There will be n_samples / batch_size SVD computations to get the principal components, versus 1 large SVD of complexity O(n_samples * n_features ** 2) for PCA.

SVD is used to decompose the data matrix into its singular values and corresponding eigenvectors, which can be used to perform dimensionality reduction. Therefore, sklearn.decomposition.IncrementalPCA works with SVD and the eigenvectors obtained from the SVD are used as the principal components for the PCA.

The time complexity is way better in IPCA and provides a very high accuracy compared to the normal PCA.



```
Accuracy=0.945
```

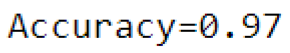
10. Bonus - Variations of LDA

Using variations of LDA, we can use **RLDA** [Regularized LDA]:

Regularized Linear Discriminant Analysis (RLDA) is a variation of Linear Discriminant Analysis (LDA) that incorporates **regularization to prevent overfitting**. RLDA solves the same optimization problem as LDA, but adds **a regularization term to the objective function to penalize large eigenvalues**. The regularization parameter **alpha** controls the amount of regularization applied.

Shrinkage is a form of regularization used to improve the estimation of covariance matrices in situations where the number of training samples is small compared to the number of features. In this scenario, the empirical sample covariance is a poor estimator, and shrinkage helps improve the generalization performance of the classifier. Shrinkage LDA can be used by setting the shrinkage parameter of the LinearDiscriminantAnalysis class to 'auto'. This automatically determines the optimal shrinkage parameter in an analytic way. Note that currently shrinkage only works when setting the solver parameter to 'lsqr' or 'eigen'.

The time complexity is way better in LDA and provides a very high accuracy compared to the normal LDA.



```
Accuracy=0.97
```