Face Recognition

Pattern Recognition

Assignment 1

Omar Ahmed ElSammak 6867
Aly Hamdy Ibrahim 6760
Ahmed Mohamed Zayan 6714

Problem Statement

We intend to perform face recognition. Face recognition means that for a given image you can tell the subject id. Our database of subjects is very simple. It has 40 subjects. Below we will show the needed steps to achieve the goal of the assignment.

Download the Dataset and Understand the Format

- The ORL dataset is available at the following link: https://www.kaggle.com/kasikrit/att-database-of-faces/
- The dataset has 10 images per 40 subjects. Every image is a grayscale image of size 92x112.
- First, the dataset is downloaded and uploaded to the google drive, then it is mounted to the notebook using the following code:

```
from google.colab import drive
drive.mount('/content/drive')
path = '/content/drive/MyDrive/eigenfaces/'
```

Generate the Data Matrix and the Label vector

- The size of each image is 92*112, therefore it will be converted to a vector of length 10304 values.
- The number of subject is 40 with 10 images per each subject, therefore the Data Matrix will be 400*10304, and the label vector will be of length 400.
- The images are read and converted using the following code:

```
def read_single_image(image_path):
    ans = []
    with open(image_path, 'rb') as f:
        assert f.readline() == b'P5\n'
        assert f.readline() == b'92 112\n'
        assert f.readline() == b'255\n'

        for i in range(10304):
            ans.append(ord(f.read(1)))
    return ans

def get_images():

    images = []
    persons = []
    number_of_persons=40

    for x in range(1, number_of_persons + 1):
        current_person_path = path + 's' + str(x) + '/'
        for y in range(1, 11):
            persons.append(str(x))
            images.append(read_single_image(current_person_path +
str(y) + '.pgm'))

    images = np.array(images)
    return images, persons


(data, labels) = get_images()
```

Split the Dataset into Training and Test sets

- The dataset is going to be split into Training and Test sets with ratio 1:1 .
- Every subject has 10 images, 5 for Training and 5 for Test.
- Odd Rows of the data set will be for Training and the Even Rows will be Testing.
- The splitting is done using the following code:

```
def custom_train_test_split(data, labels, samples_no):
    X_train = []
    y_train = []
    X_test = []
    y_test = []
    for i in range(samples_no):
        if i%2 != 0:
            X_train.append(data[i])
            y_train.append(labels[i])
        else:
            X_test.append(data[i])
            y_test.append(labels[i])
    return X_train, y_train, X_test, y_test
```

```
X_train, y_train, X_test, y_test = custom_train_test_split(data,
labels,
train_data = pd.DataFrame(X_train,index=y_train)
```

Classification using PCA & Classifier tuning

- First, we will apply PCA on the dataset to reduce the dimensions.
- We will use 4 alphas for reduction and then observe the accuracy of each alpha.
- PCA Pseudocode:

ALGORITHM 7.1. Principal Component Analysis

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

1 $\boldsymbol{\mu} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_i$ // compute mean
2 $\mathbf{Z} = \mathbf{D} - \mathbf{1}\cdot\boldsymbol{\mu}^T$ // center the data
3 $\boldsymbol{\Sigma} = \frac{1}{n}(\mathbf{Z}^T\mathbf{Z})$ // compute covariance matrix
4 $(\lambda_1, \lambda_2, \ldots, \lambda_d) = \text{eigenvalues}(\boldsymbol{\Sigma})$ // compute eigenvalues
5 $\mathbf{U} = (\mathbf{u}_1 \quad \mathbf{u}_2 \quad \cdots \quad \mathbf{u}_d) = \text{eigenvectors}(\boldsymbol{\Sigma})$ // 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

- PCA Code:

```
centralized_data = X_train - np.mean(X_train,axis=0, keepdims=True)
print("centralized_data shape:", centralized_data.shape)
cov_matrix=np.cov(np.transpose(centralized_data),bias=True)
eigen_values, eigen_vectors = np.linalg.eigh(cov_matrix)
mean = np.mean(train_data)
print(eigen_values)
```

```
def dimensionality(alpha, eig_values):
    sum = np.sum(eig_values)
    r = 0
    i = 0
    for value in eig_values:
        r = r + eig_values[i]
        i = i + 1
        if (r / sum >= alpha):
            break
    return i

def reduced_dimensions(alpha_values, eig_values):
    reduced_dimensions = []
    for alpha in alpha_values:
        reduced_dimensions.append(dimensionality(alpha, eig_values))
    return reduced_dimensions

def PCA(data, alphas,eigen_vectors,eigen_values):
    r = reduced_dimensions(alphas, eigen_values)
    return r, eigen_vectors
```

```
alphas = [0.8, 0.85, 0.9, 0.95]
r,eigen_vectors=PCA(data, alphas,eigen_vectors,eigen_values)
```

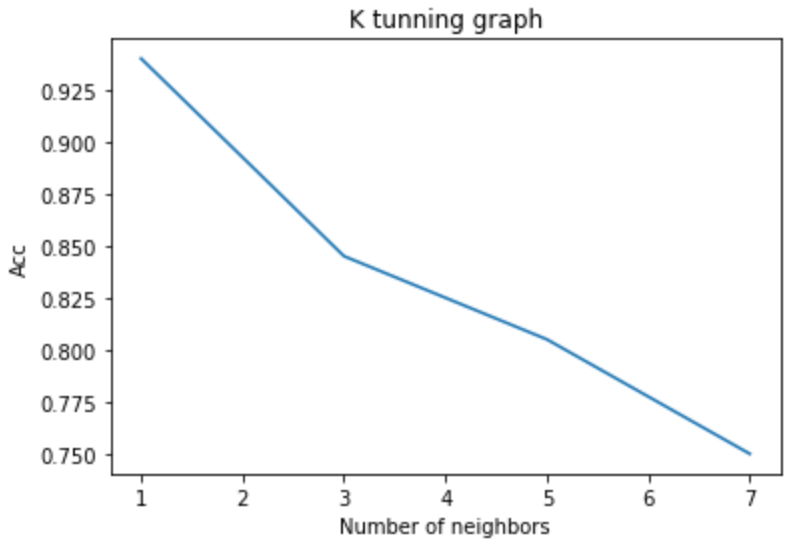
- After projecting the Training set and the Testing set on the new dimensions, we will classify using KNN with k=1.

```
from sklearn import metrics
def classify(X_train, y_train, X_test, y_test, n_neighbors, cm):
    model = KNeighborsClassifier(n_neighbors=n_neighbors)
    model.fit(X_train, y_train)
    y_predict = model.predict(X_test)
    print(f"Acc for train--> {model.score(X_train, y_train)}")
    acc = model.score(X_test, y_test)
    print(f"Acc for test--> {acc}")
    if cm:
      print("Classification report: " )
      print(classification_report(y_test, y_predict))
      confusion_matrix = metrics.confusion_matrix(y_test, y_predict)
      cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix =
confusion_matrix, display_labels = [0,1])
      cm_display.plot()
      plt.show()
    return acc
```

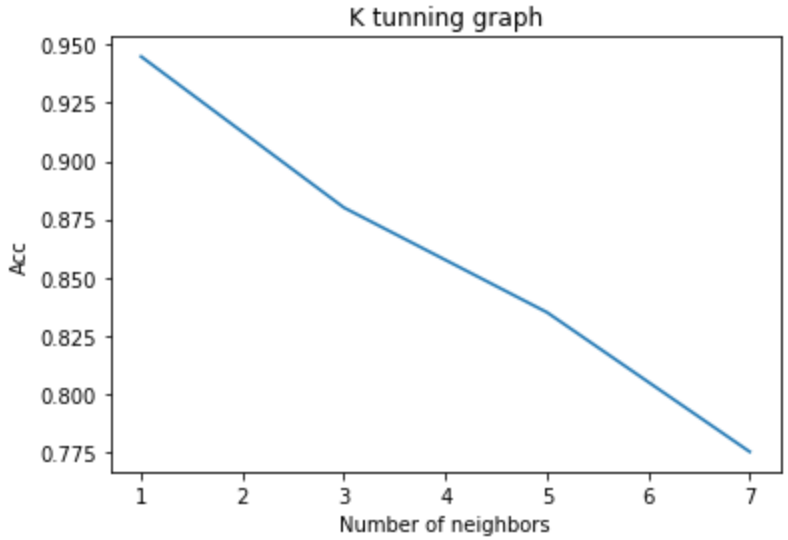
```
for i in r:
    U = eigen_vectors[:,0:i].T
    projected_train_data = np.array(np.matmul(centralized_data, U.T))
    projected_test_data = np.array(np.matmul(X_test - np.array(mean),
U.T))
    acc = []
    k_values = [1, 3, 5, 7]
    for k in k_values:
        acc.append(classify(projected_train_data, y_train,
projected_test_data, y_test, k, False))

    plt.plot(k_values, acc)
    plt.xlabel('Number of neighbors')
    plt.ylabel('Acc')
    plt.title('K tunning graph')
    plt.show()
```

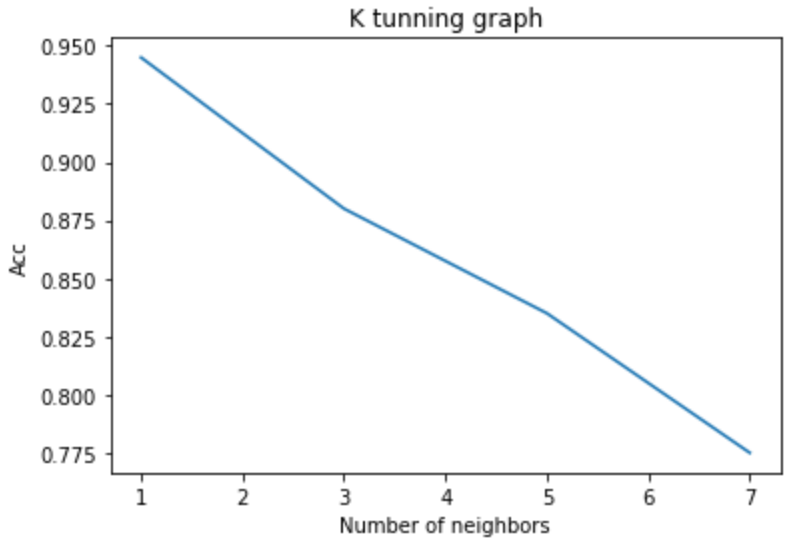
- Accuracy:
  - Alpha = 0.8



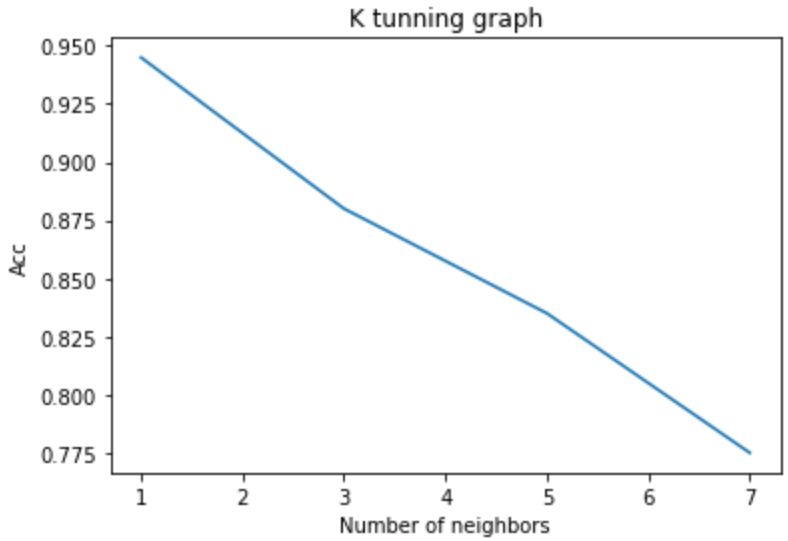
  - Alpha = 0.85



- Alpha = 0.9



- Alpha = 0.95



* White cells are for Training Accuracy, Yellow Cells are for Test Accuracy

	alpha = 0.8		alpha = 0.85		alpha = 0.9		alpha = 0.95	
k = 1	1.0	0.94	1.0	0.945	1.0	0.945	1.0	0.945
k = 3	0.925	0.845	0.965	0.88	0.965	0.88	0.965	0.88
k = 5	0.86	0.805	0.885	0.835	0.885	0.835	0.885	0.835
k = 7	0.78	0.75	0.825	0.775	0.825	0.775	0.825	0.775

Classification Using LDA & Classifier Tuning

- The LDA pseudocode will be modified to work on multiclass classification rather than binary classification:

ALGORITHM 20.1. Linear Discriminant Analysis

```
LINEARDISCRIMINANT (D = {(x_i, y_i)}_{i=1}^n):
1 D_i ← {x_j | y_j = c_i, j = 1, ..., n}, i = 1,2 // class-specific subsets
2 μ_i ← mean(D_i), i = 1,2 // class means
3 B ← (μ_1 − μ_2)(μ_1 − μ_2)^T // between-class scatter matrix
4 Z_i ← D_i − 1_{n_i} μ_i^T, i = 1,2 // center class matrices
5 S_i ← Z_i^T Z_i, i = 1,2 // class scatter matrices
6 S ← S_1 + S_2 // within-class scatter matrix
7 λ_1, w ← eigen(S^{-1} B) // compute dominant eigenvector
```

- Calculate the mean vector for every class Mu1, Mu2, ..., Mu40.

```
dimensions = 10304
no_of_persons = 40

mean_vector = dict()
for person_no in range(1, no_of_persons + 1):
  person_no = str(person_no)
  person_data = train_data.loc[person_no]
  mean = np.mean(person_data)
  mean_vector[person_no] = mean
```

- The B will be replaced with S_b: $S_b = \mathbf{\Sigma}^{m}_{k=1} n_k (\mu_k - \mu)(\mu_k - \mu)^T$
  m is the number of classes, μ is the overall sample mean, and n_k is the number of samples in the k-th class.
- S matrix remains the same, but it sums S1, S2, S3, ...S40.

```
total_mean = np.mean(X_train)
nk = 5

S = np.zeros((dimensions,dimensions))
SB = np.zeros((dimensions,dimensions))


for person_no in range(1, no_of_persons+1):
```

```
  person_no = str(person_no)
  person_data = train_data.loc[person_no]
  mean_vector[person_no]
  Z = np.array(person_data - mean_vector[person_no])
  S += Z.T.dot(Z)
  diff = np.array(mean_vector[person_no] - total_mean)
  SB += nk*diff.dot(diff.T)
```

- Use 39 dominant eigenvectors instead of just one. You will have a projection matrix U39x10304.

```
A = np.linalg.inv(S).dot(SB)
eigen_values, eigen_vectors = np.linalg.eigh(A)
idx = eigen_values.argsort()[::-1]
eig_values = eigen_values[idx]
eig_vectors = eigen_vectors[:,idx]
U = eig_vectors[:, 0:39].T
```

- After applying LDA, the training and testing set is then projected using 39 dimensions in the new space.

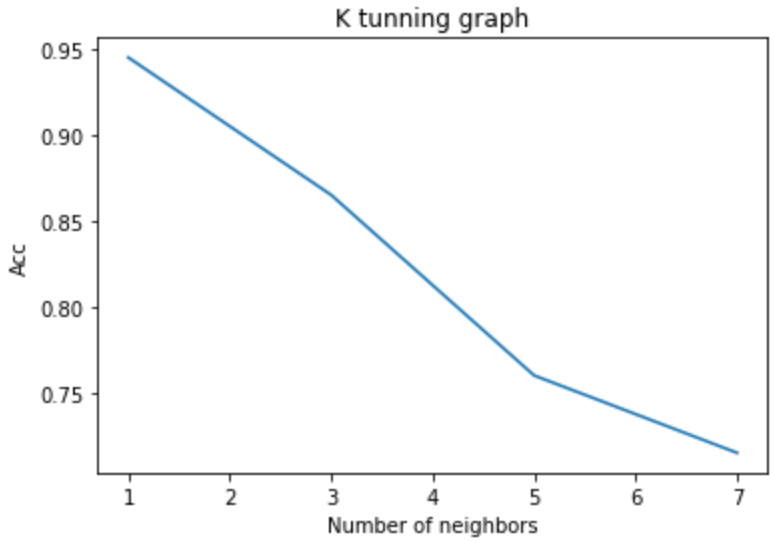
```
projected_train_data = np.array(np.matmul(X_train, U.T))
projected_test_data = np.array(np.matmul(X_test, U.T))
```

- Classification is done using KNN with k=1

```
acc = []
k_values = [1, 3, 5, 7]
for k in k_values:
    acc.append(classify(projected_train_data, y_train,
projected_test_data, y_test, k, False))

plt.plot(k_values, acc)
plt.xlabel('Number of neighbors')
plt.ylabel('Acc')
plt.title('K tunning graph')
plt.show()
```

- Accuracy:



* White cells are for Training Accuracy, Yellow Cells are for Test Accuracy

	Accuracy	
k = 1	1.0	0.945
k = 3	0.93	0.865
k = 5	0.84	0.76
k = 7	0.785	0.715

Compare vs Non-Face Images

- Download non-face images and make them of the same size 92x112. and try to solve the classification problem faces vs. Non-faces.

```from PIL import Image
```from google.colab.patches import cv2_imshow

```
from google.colab import drive
import cv2



images = []
img_no =[]

dir_path = r'/content/drive/MyDrive/resized_images/'
images = []



for filename in os.listdir(dir_path):
   if filename.endswith('.jpg') or filename.endswith('.png') or
filename.endswith('.pgm'):
        image = Image.open(os.path.join(dir_path, filename))
        image = image.resize((92, 112))
        image = image.convert('L')
        image_array = np.array(image).flatten()
        images.append(image_array)

images = np.array(images)
print(images.shape)
```

- PCA

non faces pca

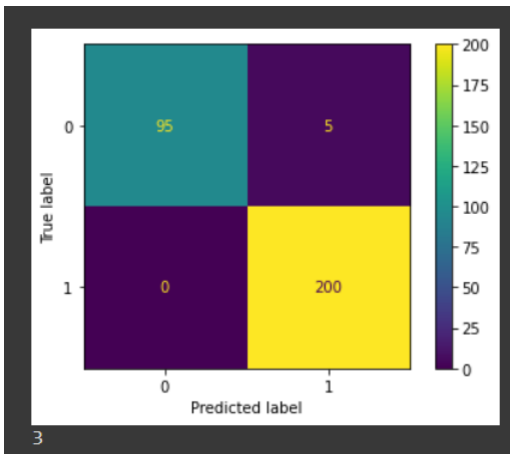
```
eigen_values_sum2 = np.sum(eigen_values2)
alphas = [0.8, 0.85, 0.9, 0.95]
r2,eigen_vectors2=PCA(data2, alphas,eigen_vectors2,eigen_values2)
```

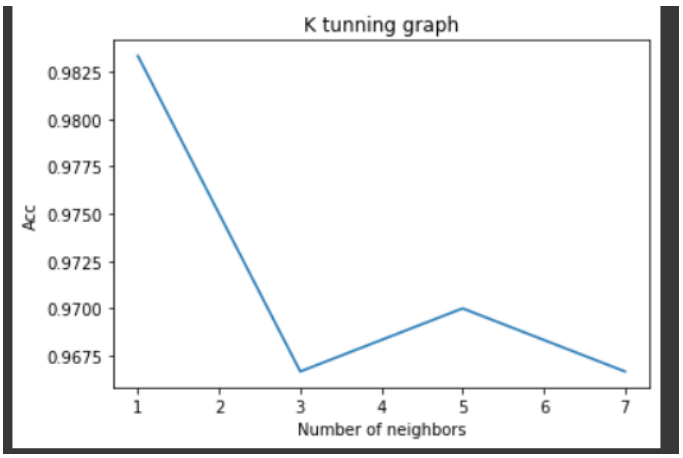
```
for i in r2:
    U2 = eigen_vectors2[:,0:i].T
    projected_train_data2 = np.array(np.matmul(centralized_data2, U2.T))
    projected_test_data2 = np.array(np.matmul(X_test2 - np.array(mean2), U2.T))
    acc2 = []
    k_values = [1, 3, 5, 7]
    for k in k_values:
        acc2.append(classify(projected_train_data2, y_train2, projected_test_data2, y_test2, k, True))

    plt.plot(k_values, acc2)
    plt.xlabel('Number of neighbors')
    plt.ylabel('Acc')
    plt.title('K tunning graph')
    plt.show()
```

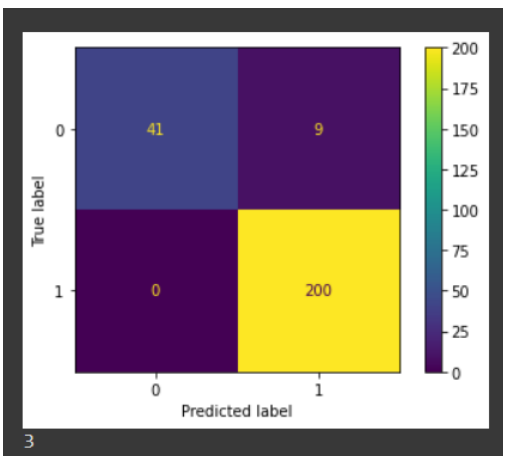
➢ Show failure and success cases

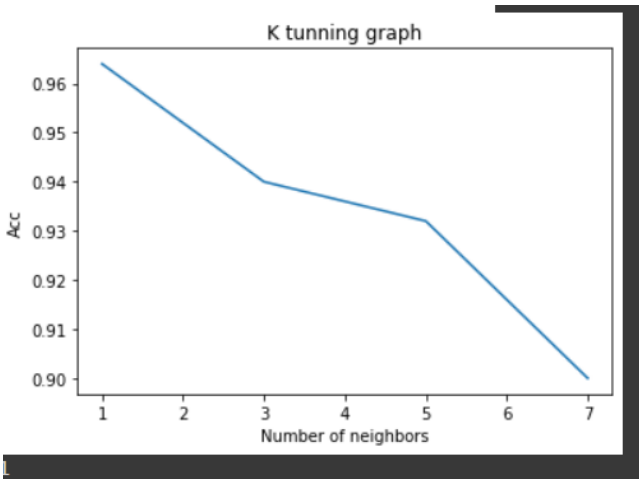
400 faces/200 non faces





400 faces / 200 non faces





- LDA

```dimensions = 10304
```

```
mean_vector2 = dict()
for img_no in range(0, 2):
  img_no = str(img_no)
  img_data = train_data2.loc[img_no]
  mean2 = np.mean(img_data)
  mean_vector2[img_no] = mean2
```

```# Calculating S and Sb Matrices
```

```
total_mean = np.mean(X_train2)
nk = [200, nonfaces_no/2]

S = np.zeros((dimensions,dimensions))
SB = np.zeros((dimensions,dimensions))

for img_no_ in range(0, 2):
  img_no = str(img_no_)
  img_data = train_data2.loc[img_no]
  mean_vector2[img_no]
  Z = np.array(img_data - mean_vector2[img_no])
  S += Z.T.dot(Z)
  diff = np.array(mean_vector2[img_no] - total_mean)
  SB += nk[img_no_]*diff.dot(diff.T)
```

```A = np.linalg.inv(S).dot(SB)
```

```
eigen_values2, eigen_vectors2 = np.linalg.eigh(A)
idx = eigen_values2.argsort()[::-1]
eig_values2 = eigen_values2[idx]
eig_vectors2 = eigen_vectors2[:,idx]
U2 = eig_vectors2[:, 0:39].T
```

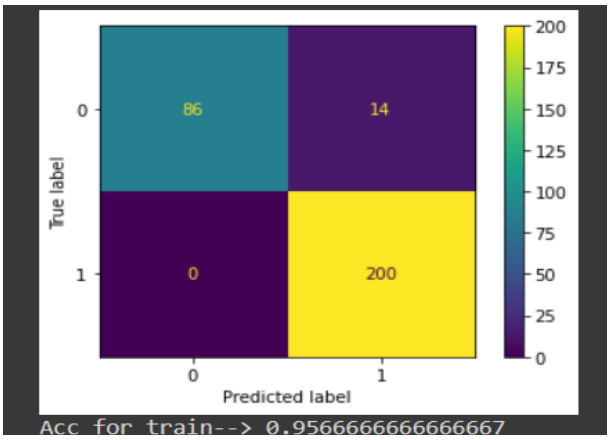
```projected_train_data3 = np.array(np.matmul(X_train2, U2.T))
```

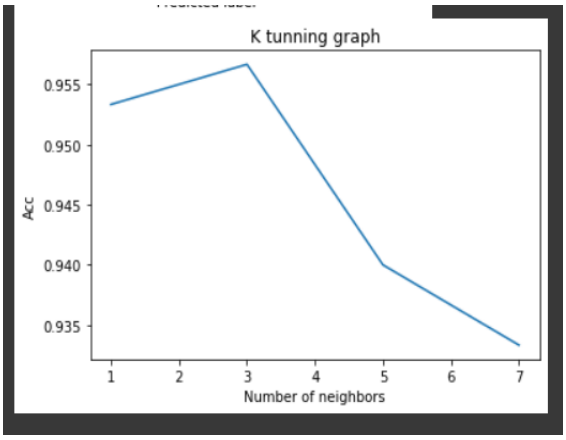
```
projected_test_data3 = np.array(np.matmul(X_test2, U2.T))
acc3 = []
```

```
k_values = [1, 3, 5, 7]
for k in k_values:
    acc3.append(classify(projected_train_data3, y_train2,
projected_test_data3, y_test2, k,True))

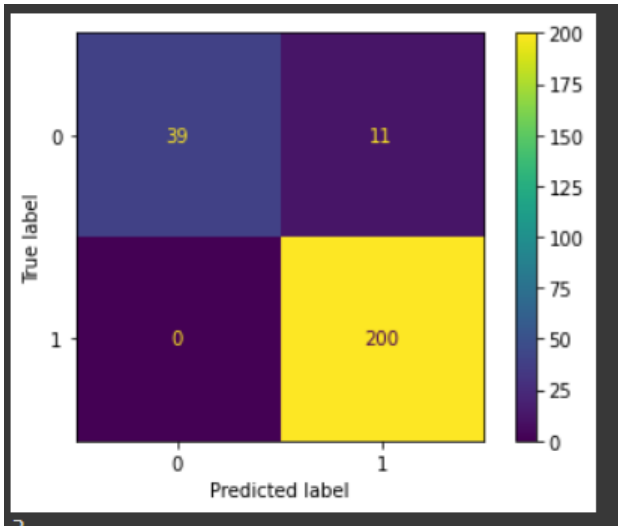
plt.plot(k_values, acc3)
plt.xlabel('Number of neighbors')
plt.ylabel('Acc')
plt.title('K tunning graph')
plt.show()
```

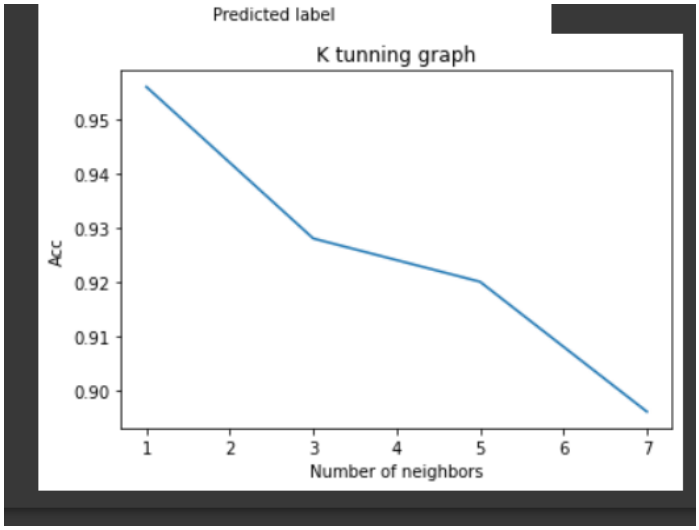
400 faces/200 non faces



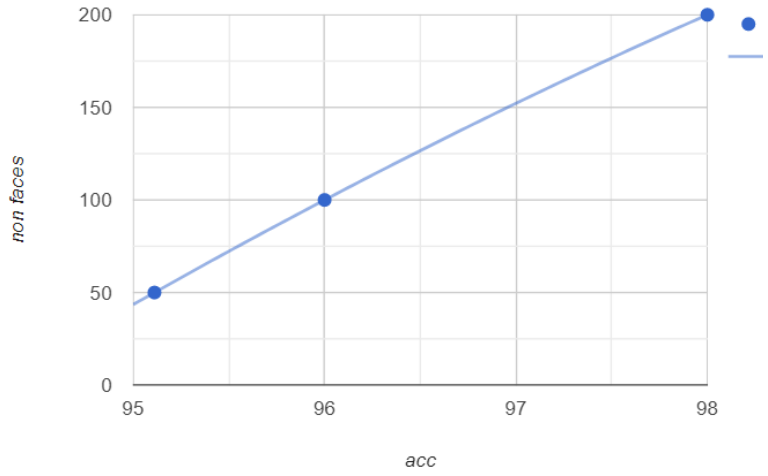


400 faces/100 non faces





- 2 Dominant Eigenvectors
-



- As number of non faces increase in data accuracy increase

Bonus

➔ Different train test split

```
```def bonus_train_test_split(data, labels):
    X_train = []
    y_train = []
    X_test = []
    y_test = []
    for i in range(len(data)):
        i_str = str(i)
        i_str = i_str[-1]#last letter of name
        if i_str == '7' or i_str == '8' or i_str == '9':
            X_test.append(data[i])
            y_test.append(labels[i])

        else:
            X_train.append(data[i])
            y_train.append(labels[i])
    return X_train, y_train, X_test, y_test
```

We used a 70-30 split to apply PCA & LDA

We used the same approach as above and did k-tuning for the k nearest neighbours too
```

* White cells are for Training Accuracy, Yellow Cells are for Test Accuracy

- PCA

	alpha = 0.8		alpha = 0.85		alpha = 0.9		alpha = 0.95	
k = 1	1.0	0.945	1.0	0.95	1.0	0.95	1.0	0.95
k = 3	0.978	0.925	0.989	0.933	0.98	0.93	0.989	0.933
k = 5	0.942	0.866	0.943	0.908	0.94	0.90	0.94	0.908
k = 7	0.878	0.8	0.90	0.83	0.90	0.833	0.903	0.833

- LDA

	Accuracy	
k = 1	1.0	0.916
k = 3	0.989	0.883
k = 5	0.928	0.825
k = 7	0.853	0.725

- **Different variations of PCA & LDA**
    - For PCA we used incremental PCA & Randomized PCA

PCA is a classical method for dimensionality reduction that finds the linear subspace of a dataset that contains the most variance. PCA computes the eigenvectors of the covariance matrix of the dataset and projects the data onto the subspace spanned by the top-k eigenvectors. PCA is computationally efficient for small to medium-sized datasets but can become slow for large datasets, especially when the number of features is high.

1. Incremental PCA: Incremental PCA (IPCA) is an extension of PCA that allows for incremental updates to the covariance matrix and eigenvectors, making it well-suited for large datasets that do not fit into memory. IPCA processes the data in chunks, updating the covariance matrix and eigenvectors after each chunk is processed. IPCA is computationally efficient and memory-friendly for large datasets, but it may produce slightly different results than classical PCA due to the incremental updates.
2. Randomized PCA: Randomized PCA is another variation of PCA that is designed to handle very large datasets by using random projections to find the top-k eigenvectors of the covariance matrix. Randomized PCA computes a low-rank approximation of the covariance matrix using random projections, and then uses classical PCA to compute the eigenvectors of the low-rank approximation. Randomized PCA is computationally efficient and well-suited for very large datasets, but may produce slightly different results than classical PCA due to the random projections.

1. PCA: The time complexity of PCA is O(nd^2 + d^3), where n is the number of samples and d is the number of features. The dominant term in the time complexity is the computation of the covariance matrix, which takes O(nd^2) time, followed by the computation of the eigenvectors, which takes O(d^3) time using standard methods like the power iteration or QR decomposition. Thus, PCA can be computationally expensive for large datasets, especially when the number of features is high.
2. Incremental PCA: The time complexity of IPCA is O(nd^2 + kd^3), where k is the number of chunks or iterations required to process the entire dataset. The dominant term in the time complexity is the computation of the covariance matrix, which takes O(nd^2) time for each chunk, followed by the computation of the eigenvectors, which takes O(d^3) time using standard methods. Thus, IPCA is

computationally efficient and memory-friendly for large datasets that do not fit into memory.

3. Randomized PCA: The time complexity of Randomized PCA is O(ndr^2 + rd^2), where r is the rank of the low-rank approximation of the covariance matrix. The dominant term in the time complexity is the computation of the low-rank approximation, which takes O(ndr^2) time using random projections, followed by the computation of the eigenvectors of the low-rank approximation, which takes O(rd^2) time using classical PCA. Thus, Randomized PCA is computationally efficient and well-suited for very large datasets.

Overall, the time complexity of PCA and IPCA is similar, with IPCA being more memory-friendly but potentially requiring more iterations. Randomized PCA is generally faster than PCA and IPCA for very large datasets, but may produce slightly different results due to the use of random projections. The choice of method depends on the specific characteristics of the dataset and the computational resources available.

	Accuracy	
	IPCA	RPCA
k = 1	0.94	0.94
k = 3	0.86	0.87
k = 5	0.84	0.84
k = 7	0.79	0.725

- For LDA we used Fisher’s LDA

Fisher's Linear Discriminant Analysis (FLDA) and Linear Discriminant Analysis (LDA) are both used as linear classifiers in machine learning and pattern recognition. Although both methods aim to identify a linear combination of features that best separates different classes of data, FLDA has some advantages over LDA.

Firstly, FLDA is more effective when dealing with only two classes, whereas LDA works better with more than two classes. Secondly, FLDA finds a projection that maximizes the separation between classes while minimizing the variance within each class, making it more effective when the classes are well-separated in the feature space. Thirdly, FLDA can handle datasets with unequal class sizes, unlike LDA which assumes that each class has an equal number of samples. Finally, FLDA is more resistant to outliers than LDA because it minimizes the variance within each class rather than assuming that each class has the same variance.

However, FLDA has some limitations too. It assumes that the data is normally distributed and that the covariance matrix of each class is equal. Also, the calculation of the inverse of the covariance matrix, which is required by FLDA, can be computationally expensive when working with large datasets.

In conclusion, while FLDA is advantageous in certain situations over LDA, the choice between the two methods depends on the specific problem and the characteristics of the data being analyzed.

In terms of time complexity, both Fisher's Linear Discriminant Analysis (FLDA) and Linear Discriminant Analysis (LDA) have similar computational requirements, with a time complexity of O(Nd^2), where N is the number of samples and d is the number of features.

However, the calculation of the inverse of the covariance matrix in FLDA can be computationally expensive, particularly when working with high-dimensional datasets, resulting in a higher time complexity than LDA. LDA, on the other hand, requires the calculation of the mean and covariance matrix of each class, which can be computationally intensive for large datasets, but it doesn't require the inverse of the covariance matrix.

Overall, the time complexity of both methods is relatively similar, but in some cases, LDA may be more computationally efficient than FLDA.

	Accuracy	
	FLDA	LDA
k = 1	0.955	0.945
k = 3	0.935	0.865
k = 5	0.91	0.76
k = 7	0.915	0.715