TP_Unsupervised_1_FaceRecognition

November 25, 2020

0.1 Face recognition

Load the original images present in the files *'YaleB_32x32.mat'*. This is a small part of the freely available Extended Yale Face Database B downloaded from http://www.cad.zju.edu.cn/home/dengcai/Data/FaceData.html. It contains 2414 cropped images resized to 32x32 pixels. Every image is represented as a vector 1x1024 and all images are stacked in a matrix called data. There are 38 subjects with around 64 near frontal images per individual under different illumination conditions. Once loaded and normalised the data, such that the pixels are between 0 and 1, you can plot images using the function *'imshow'*.

1 Goal

The goal of this part is to evaluate the performance of the dimensionality reduction techniques presented this morning for face recognition. We divide the data-set into two parts, training and test. For every dimensionality reduction technique, you will first extract a set of basis images from your training data-set. Then, you will project the test subjects in this new basis and use the nearest neighbor algorithm to evaluate the performance of the dimensionality reduction technique.

Let's load the data.

[1]:
```
if 'google.colab' in str(get_ipython()):
  from google_drive_downloader import GoogleDriveDownloader as gdd
  gdd.
 ↪download_file_from_google_drive(file_id='1rgICXtcIAgDqSoHnNXNZMD_iNABF3RZA',
  dest_path='./YaleB_32x32.mat')
else:
  print('You are not using Colab. Please define working_dir with the absolute␣
 ↪path to the folder where you downloaded the data')

# Please modify working_dir only if you are using your Anaconda (and not Google␣
 ↪Colab)
Working_directory="./"
```

```
You are not using Colab. Please define working_dir with the absolute path to the
folder where you downloaded the data
```

Load the libraries

[2]:
```
import math
import random
import numpy as np
import numpy.matlib
import matplotlib.pyplot as plt
plt.close('all')

from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.decomposition import PCA
from sklearn.decomposition import KernelPCA
from sklearn.decomposition import FastICA
from sklearn.cluster import KMeans
from sklearn.decomposition import NMF


from scipy import linalg as LA
from scipy.stats import ortho_group

from scipy.io import loadmat
```

Here you can copy the functions `pcaLecture`, `KpcaGaussianLecture` and `FastICALecture` implemented in the previous jupyter-notebook.

[3]:
```
# For IMP
def pcaLecture(X):
    '''
    Inputs:
            X: is a [Nxd] matrix. Every row is an observation and every
              column consists of features.
    Outputs:
            Y: is a [Nxd] matrix representing the scores, namely the
            coordinates of X onto the new basis given by the eigenvactors U
            of the covariance matrix of X. Columns are the principal components.

            U: columns are Eigenvectors (sorted from the greatest to the lowest␣
 ↪eigenvalue)

            D: Eigenvalues (sorted from the greatest to the lowest eigenvalue)

            var_explained: percentage of the original variability explained
            by each principal component.
    '''

    N=X.shape[0]
    Xc=X-np.mean(X) # centering
```

```
    D2, Uh = LA.svd(Xc)[1:3] # computation of eigenvectors and eigenvalues␣
↪using SVD
    U=Uh.T
    #De, Ue = LA.eig(np.matmul(Xc.T,Xc)/(N-1))
    Y=np.dot(Xc,Uh) # computation of the scores, use np.dot()
    D=D2**2/(N-1) # computation of the eigenvalues
    tot=np.sum(D)
    var_explained = D*100/tot # computation of explained variance
    return Y,U,D,var_explained
```

[4]:
```
# For IMP
def KpcaGaussianLecture(X,gamma):
    '''
    Inputs:
            X: is a [Nxd] matrix. Every row is an observation and every
            column is a feature.

    Outputs:
            Y: is a [Nxd] matrix representing the scores, namely the
            coordinates of \phi(X) onto the new basis given by the eigenvactors
            of the covariance matrix of \phi(X). Columns are the principal␣
↪components.

            An: columns are Eigenvectors normalised (sorted from the greatest
            to the lowest eigenvalue)

            D: Eigenvalues (sorted from the greatest to the lowest eigenvalue)

            var_explained: percentage of the original variability explained
            by each principal component.

    '''

    # kernel matrix
    def kernel_matrix(X,gamma):
      N=X.shape[0]
      InnerX = np.dot(X,X.T)
      temp1=np.sum(X**2,axis=1).reshape((N,1))
      temp2=np.sum(X**2,axis=1).reshape((1,N))
      Norm1 = np.repeat(temp1,N,axis=1)
      Norm2 = np.repeat(temp2,N,axis=0)
      Norm = Norm1+Norm2-2*InnerX;
      Norm[Norm<1e-10]=0;
      K=np.exp(-Norm/(2*gamma**2))
      return K

    N=X.shape[0]
```

```
    K=kernel_matrix(X,gamma)

    oneN=np.ones((N,N))/N;
    Kc=K-np.mean(K)# center kernel matrix

    # eigenvalue analysis
    D,A=LA.eigh(Kc)
    idx = D.argsort()[::-1]  # reverse order to make 'descend'
    D = np.real(D[idx])
    D[D<0]=1e-18 # make negative eigenvalues positive (and almost 0)
    A = np.real(A[:,idx])

    # variance explained
    tot=np.sum(D)
    var_explained = D/tot # computation of explained variance

    # Normalisation eigenvectors
    # Norm of every eigenvector is 1, we want it to be 1/sqrt(N*eig)

    An=np.copy(A)
    for i in range(N):
        An[:,i]=np.dot(A[:,i],(1/np.sqrt((N-1)*D[i])) )

    Y=np.dot(Kc,An) # computation of the scores, use np.dot()

    return Y,An,D,var_explained
```

[5]:
```
# For IMP
def FastICALecture(X,N_Iter=3000,tol=1e-5,plot_evolution=0,whiten=True):
    '''
    Inputs:
                    X: is a [d x N] matrix. Every column is an observation
                    and every row is a feature.

                    (Optional) N_Iter: maximum number of iterations

                    (Optional) delta: convergence criteria threshold

                    (Optional) plot_evolution: plot evolution of error

     Outputs:
                    S: [d x N] matrix.  Each column is an independent component
                    of the centred and whitened input data X

                    W: [d x d] matrix. It is the demixing matrix. S = W*Xcw
```

```
    '''
    random.seed(42)
    # First derivative of G
    def g(t):
        res = t * np.exp(-(t**2)/2)
        return res

    # Second derivative of G
    def gp(t):
        res = (1 - t**2) * np.exp(-(t**2)/2)
        return res

    # Size of X
    d,N=X.shape

    # Compute sample mean
    mu = X.mean(axis=1, keepdims=True)

    # Center data
    #Xc=X
    Xc=X-np.mean(X)

    # Compute covariance matrix
    C=np.cov(Xc)

    # Whiten data
    Xcw=np.dot(LA.inv(LA.sqrtm(C)),Xc)

    # check if are whitened
    if np.sum(np.eye(d) - np.abs(np.cov(Xcw)))>1e-10:
        raise NameError('Your whitening transformation does not work...')

    # Initialize W
    W = ortho_group.rvs(d) # random orthogonal matrix

    # delta evolution
    k = 0
    delta = np.inf
    evolutionDelta=[]

    while delta > tol and k < N_Iter:

        k = k + 1
        W_old = np.copy(W)

        Wp = np.dot(g(np.dot(W,Xcw)),Xcw.T) - np.dot(np.diagflat(np.dot(gp(np.
↪dot(W,Xcw)),np.ones((N,1)))),W)
```

```
        W = np.dot(LA.inv(LA.sqrtm(np.dot(Wp,Wp.T))),Wp) # W*W'=I
        if np.sum(np.eye(d)-np.abs(np.dot(W,W.T)))>1e-10:
            raise NameError('W should be an orthogonal matrix. Check the computations')

        delta = 1-np.min(np.abs(np.diag(np.dot(W.T,W_old))))
        evolutionDelta.append(delta)

        if k==1 or k%100==0:
            print('Iteration ICA number ', k, ' out of ', N_Iter , ', delta =', delta)


    if k==N_Iter:
        print('Maximum number of iterations reached ! delta = ', delta)
    else:
        print('Convergence achieved ( delta = ', delta, ') in ', k, 'iterations')


    # Independent components
    S = np.dot(W,Xcw)

    if plot_evolution==1:
        plt.figure(figsize=(6, 6))
        plt.plot(range(k),evolutionDelta,'bx--', linewidth=4, markersize=12)
        plt.title('Evolution of error - ICA')
        plt.show()

    return S,W
```

This is a useful function to plot the basis images. Be careful, each row of data is a basis image.

[6]:
```
def plotFaces(data,r,c,ncol=2,N=0,indeces=None,title=None):
    # data: each face is a row in data
    # r,c = number of rows and columns of each image
    # n_col = number of columns for subplots
    # N = random images to plot (used only if indeces is empty)
    # indeces = indeces of images to plot
    # title = title of the plot


    if indeces is None:
        if N==0:
            raise NameError('You should define either N or indeces')
        else:
            print('Use N random subjects')
```

```
            indeces=np.random.randint(0,data.shape[0],(N,1))

    nrow=math.ceil(len(indeces)/ncol)

    fig=plt.figure(figsize=(17, 6))
    plt.suptitle(title, size=16)
    for i, index in enumerate(indeces):
        fig.add_subplot(nrow, ncol, i+1)
        plt.imshow(np.resize(data[index,:],(r,c)).T,origin='upper',cmap='gray')
        plt.xticks(())
        plt.yticks(())
    #plt.subplots_adjust(left=0.01, bottom=0.05, right=0.99, top=0.93, wspace=0.
↪04, hspace=0.0)
```

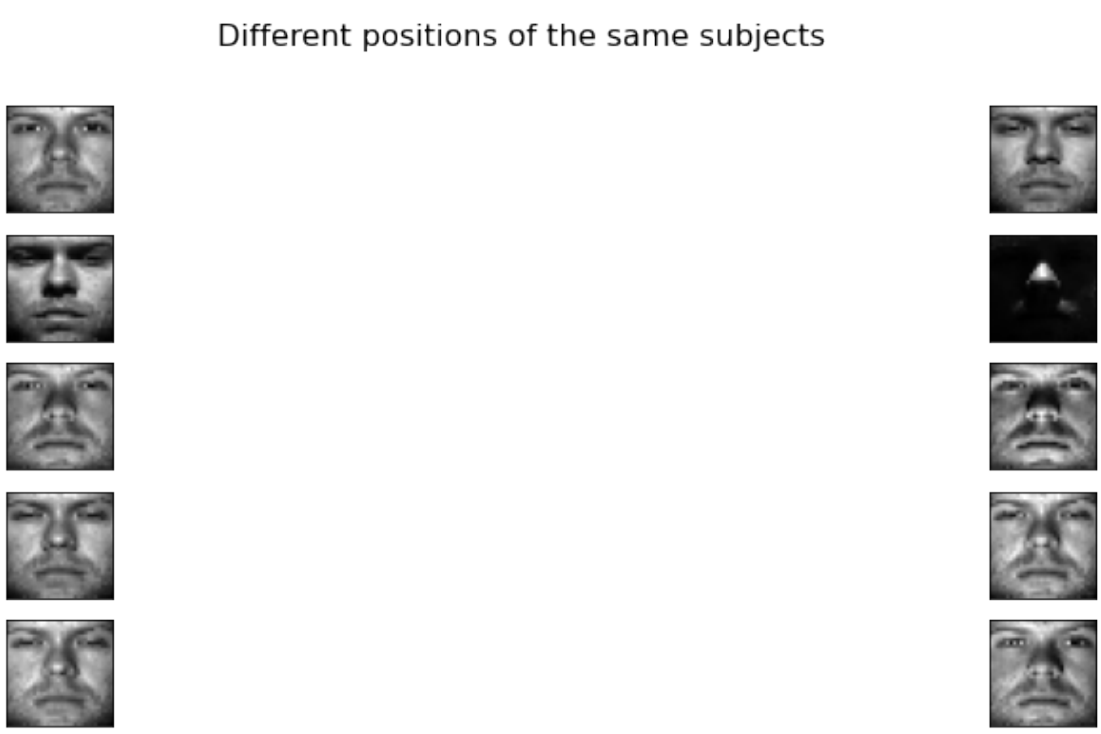
Let's load the data and compute some parameters.

[7]:
```
x = loadmat(Working_directory + './YaleB_32x32.mat')
data=x['fea']
d=data.shape[1] # number of pixels of the images
subjectIndex=x['gnd'] # we have one index per subject
maxValue = np.max(np.max(data)) # max intensity value
data = data/maxValue; # Scale pixels to [0,1]

Ns=len(np.unique(subjectIndex)); # Number subjects
Is=round(len(subjectIndex)/Ns) # Number images per subject (on average, not the␣
↪same number for every subject)
r=int(np.sqrt(d)) # number rows of each image
c=r # number columns of each image, equal to row since images are square

print('There are', data.shape[0], 'facial images and each image has', d,␣
↪'pixels' )
print('There are', Ns, 'different subjects and each subject has on average',␣
↪Is, 'images')
```

```
There are 2414 facial images and each image has 1024 pixels
There are 38 different subjects and each subject has on average 64 images
```

Let's plot first 10 images of different subjects and then 10 images of the same subject but with different positions and illumination conditions

[8]:
```
# Plot data
indexDifferent=np.arange(1,Is*40,Is)
plotFaces(data,r,c,ncol=3,indeces=indexDifferent[0:10],title='Different␣
↪subjects')
indexSame=np.arange(0,10,1)
plotFaces(data,r,c,ncol=2,indeces=indexSame,title='Different positions of the␣
↪same subjects')
```



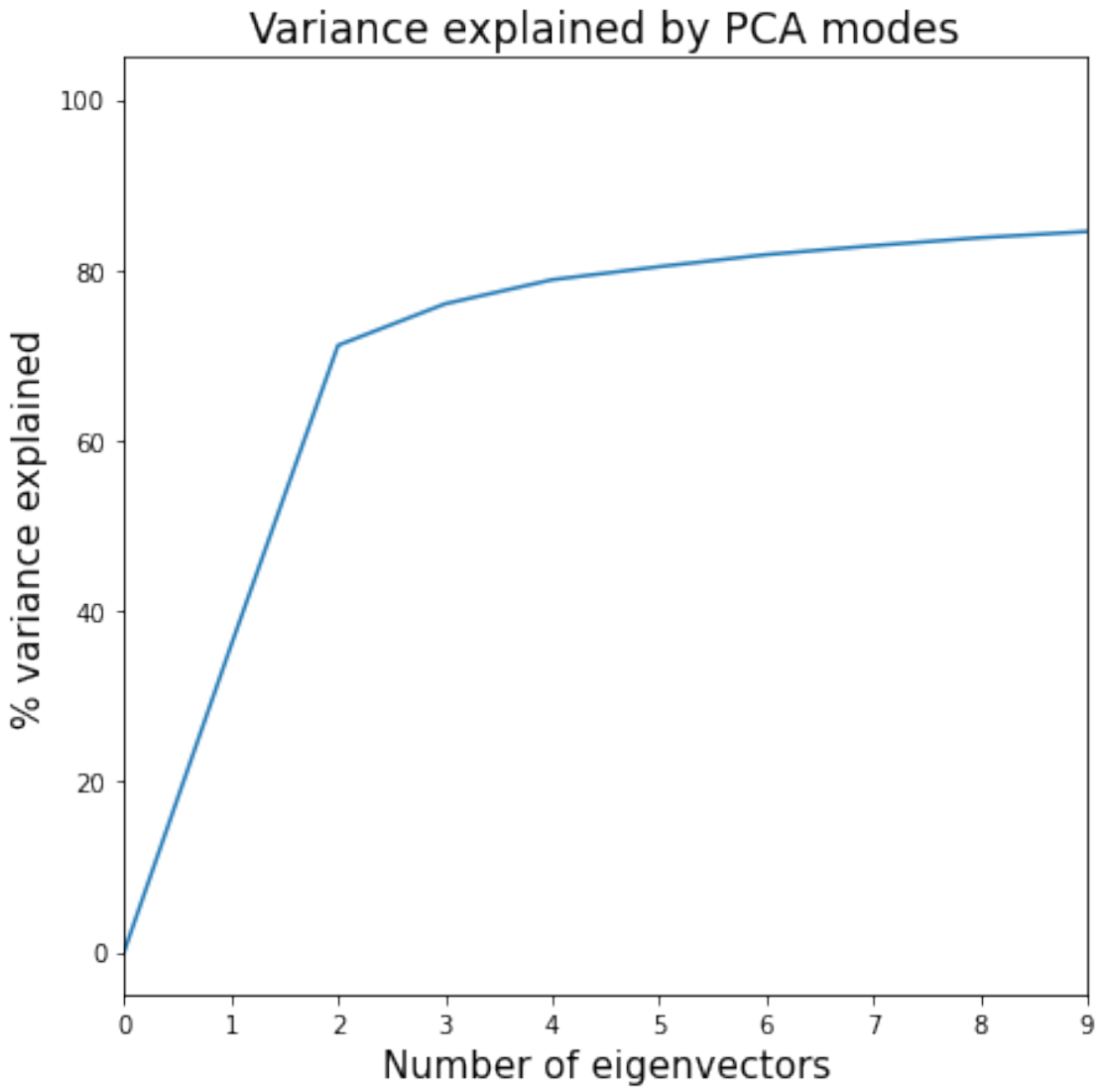
We can now use PCA to investigate the main variations within the data.

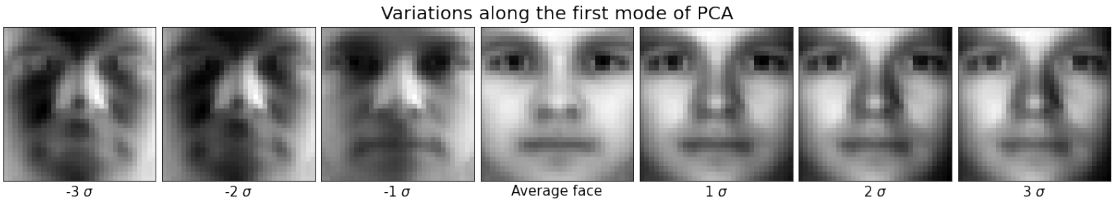
Question: 1. (IMP + IMH) How many modes do you need to explain at least 80% of the variability in the data ? Look at the three main modes and explain which are the main variations in the data.
Answer:

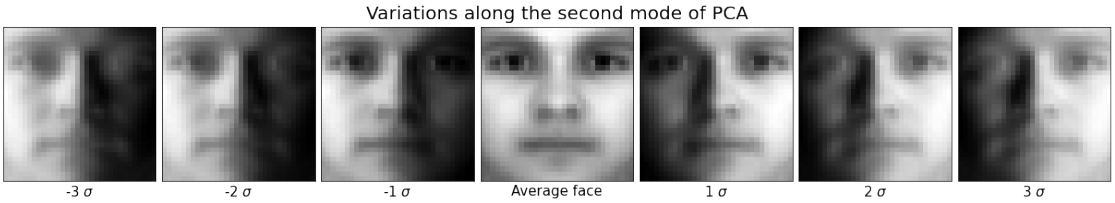
From the first graph, we can see that around 4 to 5 modes will explain 80% of the variance in the data. We will choose 4 since we that after that the modes after 4 do not contribute a lot and we reached a plateau. with the first mode, we can get the general look of the face, the second mode gives us the shadow of the right or left, depends on where the person is looking same for the third mode just regarding up and down shadows.

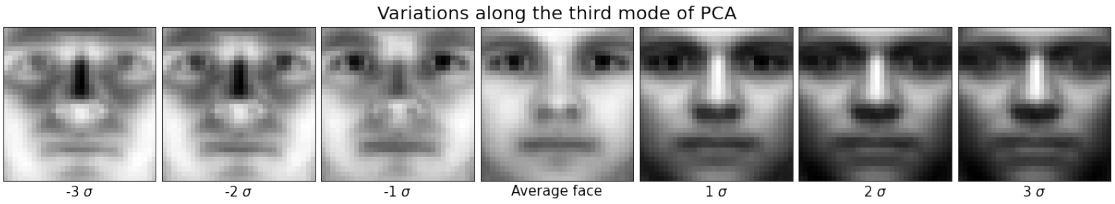
[9]:
```
# Linear interpolation along the first two modes
Xm=data.mean(axis=0) # average face

pca = PCA(random_state=1) # by fixing the random_state we are sure that results␣
↪are always the same
YpcaTrain=pca.fit_transform(data)
UpcaTrain=pca.components_.T # we want PC on columns
var_explained_pca=pca.explained_variance_ratio_
DpcaTrain = (pca.singular_values_)**2/(data.shape[0]-1) # computation of the␣
↪eigenvalues
indices=np.linspace(-3, 3, num=7, dtype=np.int16) # Interpolation indices

# Variance explained by each eigenvector
fig=plt.figure(figsize=(7, 7))
ax=plt.subplot(111)
ax.set_xlim(0, 9)
dim=np.arange(0,10,1)
plt.plot(np.concatenate(([0], np.cumsum(var_explained_pca)*100)))
plt.xticks(dim)
plt.xlabel('Number of eigenvectors',fontsize=15)
plt.ylabel('% variance explained',fontsize=15)
plt.title('Variance explained by PCA modes',fontsize=17)

## First mode
fig=plt.figure(figsize=(17, 3))
plt.suptitle('Variations along the first mode of PCA', size=20)
for i, index in enumerate(indices):
  image = Xm + index * np.sqrt(DpcaTrain[0]) * UpcaTrain[:,0]
  fig.add_subplot(1, len(indices), i+1)
  plt.imshow(np.resize(image,(r,c)).T,origin='upper',cmap='gray')
  if index != 0:
    plt.xlabel(r'%i $\sigma$' %index, fontsize=15)
  else:
    plt.xlabel('Average face', fontsize=15)
  plt.xticks(())
  plt.yticks(())
  plt.subplots_adjust(left=0.01, bottom=0.05, right=0.99, top=0.93, wspace=0.
↪04, hspace=0.0)


## Second mode
```

```
fig=plt.figure(figsize=(17, 3))
plt.suptitle('Variations along the second mode of PCA', size=20)
for i, index in enumerate(indices):
  image = Xm + index * np.sqrt(DpcaTrain[1]) * UpcaTrain[:,1]
  fig.add_subplot(1, len(indices), i+1)
  plt.imshow(np.resize(image,(r,c)).T,origin='upper',cmap='gray')
  if index != 0:
    plt.xlabel(r'%i $\sigma$' %index, fontsize=15)
  else:
    plt.xlabel('Average face', fontsize=15)
  plt.xticks(())
  plt.yticks(())
  plt.subplots_adjust(left=0.01, bottom=0.05, right=0.99, top=0.93, wspace=0.
 ↪04, hspace=0.0)

## Third mode
fig=plt.figure(figsize=(17, 3))
plt.suptitle('Variations along the third mode of PCA', size=20)
for i, index in enumerate(indices):
  image = Xm + index * np.sqrt(DpcaTrain[2]) * UpcaTrain[:,2]
  fig.add_subplot(1, len(indices), i+1)
  plt.imshow(np.resize(image,(r,c)).T,origin='upper',cmap='gray')
  if index != 0:
    plt.xlabel(r'%i $\sigma$' %index, fontsize=15)
  else:
    plt.xlabel('Average face', fontsize=15)
  plt.xticks(())
  plt.yticks(())
  plt.subplots_adjust(left=0.01, bottom=0.05, right=0.99, top=0.93, wspace=0.
 ↪04, hspace=0.0)
```


Variance explained by PCA modes
% variance explained
Number of eigenvectors
0
20
40
60
80
100
0
1
2
3
4
5
6
7
8
9


Variations along the first mode of PCA
-3 σ
-2 σ
-1 σ
Average face
1 σ
2 σ
3 σ





We can now move to evaluate the performance of the dimensionality reduction techniques presented this morning for face recognition. We first divide the data-set into two parts, training (80%) and test (20%) in a stratified way (subjects are divided in a balanced way between the two parts).

We will see in the next lecture why we need to divide into training and test sets. For now, just know that we will use the first set to train our algorithm and the second set to test the performance of our algorithm on new, unseen data.

For every dimensionality reduction technique, you will first extract a set of basis images from your training data-set. Then, you will project the test subjects in this new basis and use the nearest neighbor algorithm to evaluate the performance of the dimensionality reduction technique. For each test sample, the nearest neighbor algorithm simply looks for the closest training sample and then assigns the same label (i.e. index of subject).

[10]:
```
Xtrain, Xtest, Id_Train, Id_Test =␣
 ↪train_test_split(data,subjectIndex,test_size=0.20,stratify=subjectIndex,␣
 ↪random_state=44)
Xctest=Xtest-np.mean(Xtest,axis=0) # centering
Xctrain=Xtrain-np.mean(Xtrain,axis=0) # centering
```

As first idea, we could simply use the pixel intensities as features. This is basically like using the original data, without dimensionality reducton techniques.

[11]:
```
## Use the pixel intensities to find the correct subject for the test images
NN=KNeighborsClassifier(n_neighbors=1)
NN.fit(Xctrain,Id_Train.ravel())
print('By using the pixel intensities, we use ', Xctrain.shape[1], ' features')
print('Percentage of correct answer using the pixel intensities is ', NN.
 ↪score(Xctest,Id_Test))
```

```
By using the pixel intensities, we use  1024  features
Percentage of correct answer using the pixel intensities is  0.7412008281573499
```

2 PCA

You will first use PCA. Compute the scores Y_{train}, eigenvectors U_{train} and eigenvalues D_{train} of the training set. The eigenvectors U_{train} represent the basis images and they are usually called *'Eigenfaces'*. Then, project both training and test data onto the eigenvectors that explain 99% of the variability of the training set L^{99}_{train}. You will obtain two vectors of scores, $Y^{99}_{train} = X_{train}L^{99}_{train}$ and $Y^{99}_{test} = X_{test}L^{99}_{test}$, which you will use for evaluating the performance of the algorithm. Use the function `KNeighborsClassifier` to test the performance.

Practical Questions:

1. (IMP+IMH) Use either the scikit-learn implementation or yours (better!) to compute the PCA for the training data-set. Comment the eigenfaces. Do they seem "real" ?

Answers:

1. Most of them look like a real face or negative of the picture of the face, but there are some of them do not look human.

Theoretical Questions:

1. (IMP+IMH) In your opinion, why do we need to center the data before computing a PCA ? If you want, you can use the previous toy examples to answer this question.
2. (IMP+IMH) Let X be the original data, a matrix $[N, d]$, and Y the scores of a PCA keeping all eigenvectors, which means that Y is also a matrix $[N, d]$. Are X and Y equal ? If not, why ? What would you use (generally speaking) in a machine learning problem ? Why ?
3. (IMP) Let x_p and x_q be two row-vectors representing two images, U an *orthogonal* matrix whose columns are the eigenvectors of X and $y_p = x_pU$, $y_q = x_qU$, check that $x_px_q^T = y_py_q^T$. This shows that $Y = XU$ is a linear transformation that preserves inner products.
4. (IMP) Let C be the covariance matrix of X and $C = UDU^T$ its eigen decomposition. Show that the covariance matrix of $Y = XU$ is D.

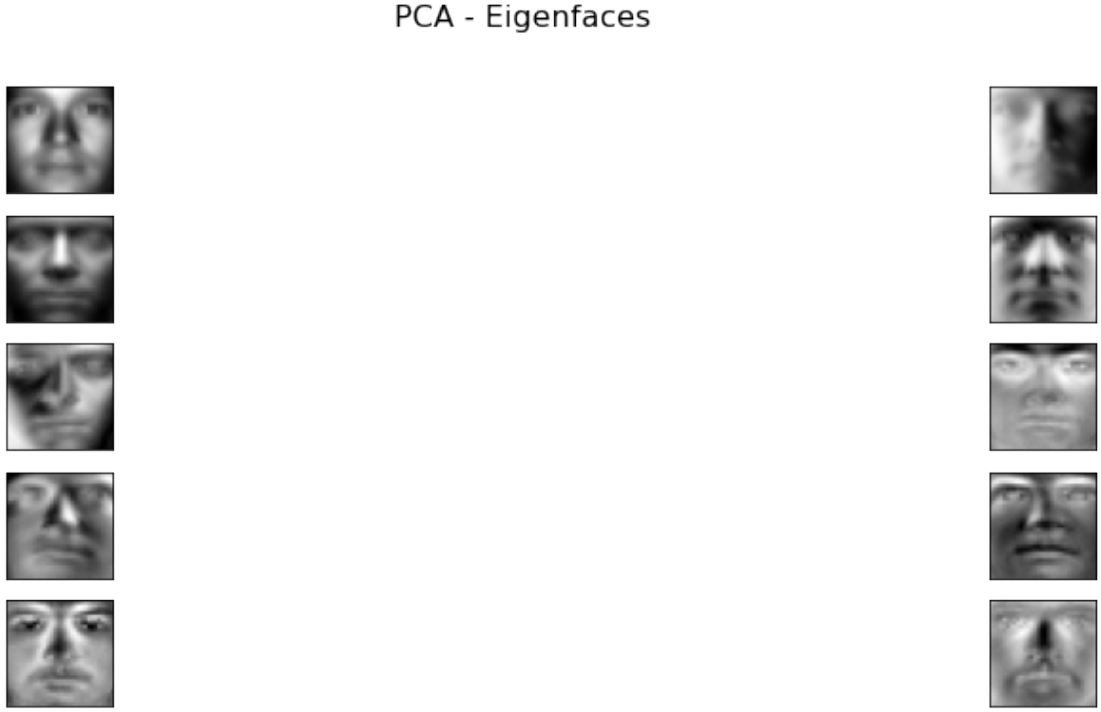
Answers:

1. We need to center the data before applying PCA because the Principal Components are computed based on the sum of squared distances from 0. This means that applying PCA on uncentered data will not capture variance within the data, but rather the distance of the dataset from the origin.
2. X and Y are not equal, but they do represent the same data. Y is the representation of X in its eigenbasis. for machine learning I would use the Y since each dimension represents a specific, discriminatory variation axis and the machine learning algorithm will have a better results with this data.
3. Orthogonal matrices needs to follow $UU^T = I$, I is the identity matrix therefore we can easily go from the known $y_q^T = (x_qU)^T = U^Tx_q^T$ to $y_py_q^T = x_pUU^Tx_q^T = x_pIx_q^T = x_px_q^T$ which gives us $y_py_q^T = x_px_q^T$
4. So we got $Cov(X) = C = UDU^T = \frac{X^TX}{N-1}$ from that we can get $Cov(Y) = \frac{Y^TY}{N-1} = \frac{(XU)^T(XU)}{N-1} = \frac{U^TX^TXU}{N-1}$ and from the relationship listed above $\frac{X^TX}{N-1} = UDU^T$ and $U^TU = I_d$ $Cov(Y) = I_dDI_d = D$

[12]:
```
## PCA (scikit-learn implementation)
```

```
pca = PCA(random_state=1) # by fixing the random_state we are sure that results␣
 ↪are always the same
YpcaTrain=pca.fit_transform(Xtrain)
UpcaTrain=pca.components_.T # we want PC on columns
var_explained_pca=pca.explained_variance_ratio_

# Threshold defined as 99% of the variability
Threshold_PCA = 0.99
CumulativePca=np.cumsum(var_explained_pca)
indexPCA=np.argwhere(CumulativePca>Threshold_PCA)
PCAComp=indexPCA[0][0]

print('PCA uses ', PCAComp, ' features')

# Selection of the eigenvectors
Yr_train_PCA=YpcaTrain[:,:PCAComp]
Ur_train_PCA=UpcaTrain[:,:PCAComp]

# Computation of the test scores using the eigenvectors computed with the
# training data-set
Yr_test_PCA=np.dot(Xctest,Ur_train_PCA)

# Plot the Eigenfaces
plotFaces(UpcaTrain.T,r,c,ncol=2,indeces=np.arange(0,10,1),title='PCA -␣
 ↪Eigenfaces')

# Score
NN.fit(Yr_train_PCA,Id_Train.ravel())
print('Percentage of correct answer using PCA is ', NN.
 ↪score(Yr_test_PCA,Id_Test))
```

```
PCA uses  232  features
Percentage of correct answer using PCA is  0.7101449275362319
```



[13]:
```
# For IMP
## PCA (your implementation)
YpcaTrain,UpcaTrain,_,var_explained_pca=pcaLecture(Xtrain)
UpcaTrain=UpcaTrain.T
# Threshold defined as 99% of the variability
Threshold_PCA = 99
CumulativePca=np.cumsum(var_explained_pca)
indexPCA=np.argwhere(CumulativePca>Threshold_PCA)
PCAComp=indexPCA[0][0]

print('PCA uses ', PCAComp, ' features')

# Selection of the eigenvectors
Yr_train_PCA=YpcaTrain[:,:PCAComp]
Ur_train_PCA=UpcaTrain[:,:PCAComp]

# Computation of the test scores using the eigenvectors computed with the
# training data-set
Yr_test_PCA=np.dot(Xctest,Ur_train_PCA)

# Plot the Eigenfaces
plotFaces(UpcaTrain,r,c,ncol=2,indeces=np.arange(0,10,1),title='PCA -␣
↪Eigenfaces')
```

```
# Score
NN.fit(Yr_train_PCA,Id_Train.ravel())
print('Percentage of correct answer using PCA is ', NN.
 ↪score(Yr_test_PCA,Id_Test))
```

```
PCA uses  224  features
Percentage of correct answer using PCA is  0.6625258799171843
```

3 KPCA

In this section, we are going to do exactly the same procedure as before but using Kernel-PCA with a Gaussian kernel. Remember that we need to compute and center the test kernel matrix $[\mathbf{K}]_{ij} =< \phi(x_i) - \frac{1}{N}\sum_{s=1}^{N}\phi(x_s), \phi(x_j) - \frac{1}{N}\sum_{s=1}^{N}\phi(x_s) >$ and that, once computed the basis vectors in the training set $\{\alpha_i\}$, we can compute the score for a test sample t using the following equation:

$$y_i(t) = \sum_{j=1}^{N} a_{ij} < \phi(t) - \frac{1}{N}\sum_{s=1}^{N}\phi(x_s), \phi(x_j) - \frac{1}{N}\sum_{s=1}^{N}\phi(x_s) >= \sum_{j=1}^{N} a_{ij}\tilde{k}(t, x_j)$$

Answer the following questions:

Questions:

1. (IMP+IMH) Look for the best gamma value
2. (IMP+IMH) Why the basis vectors $\{\alpha_i\}$ are not plotted as in PCA ?
3. (IMP+IMH) Is it worth it, in your opinion, to compute PCA and/or KPCA ? Why not using the original pixel intensities ? Please consider the following aspects in your answer:

performance, computational time, number of features, and interpretability of the results.

4. (IMP - Optional) Create a new function `Kpca_poly_lecture` where you change the kernel to $k(x, y) =< x, y >^d$. Evaluate the performance of this new kernel.

Answers:

1. Best gamma is 3.5 with 64.8% accuracy.
2. In Kernel PCA is not in the same space of the original data and therefore it's not easy to access and plot.
3. Computing Kernel PCA or PCA allows us to reduce the dimensionality of the data but still to have a good accuracy as the pixel intensity analysis. The computational time and complexity will be lower. as well as see the result using eigenfaces you can easily compare to pixel intensity analysis.

In my opinion, computing K-PCA and/or PCA allows us to reduce the number of features needed to achieve similar accuracy levels as the pixel intensity analysis. Reducing the number of features allows for faster and less computationally intense analysis, and it allows a human operator to look at the results and interpret them with much greater ease: by looking at eigenfaces we can distinguish the features that make up each dimension, compared to looking at pixel intensities an dheatmaps, that would give fairly obscure an darbitrary axes to the human eye.

[14]:
```
# Kernel-PCA (scikit-learn implementation)
## choose a gamma value
gamma=3.5
#

Kpca = KernelPCA(kernel='rbf', gamma=gamma, remove_zero_eig=True,␣
 ↪random_state=1)
YKpca=Kpca.fit_transform(Xctrain)
DKpca=Kpca.lambdas_
AnKpca=Kpca.alphas_

# variance explained
tot=np.sum(np.real(DKpca))
varexplKpca = DKpca/tot # computation of explained variance

# Threshold defined as 99% of the variability
Threshold_KPCA = 0.99
CumulativeKPca=np.cumsum(varexplKpca)
indexKPCA=np.argwhere(CumulativeKPca>Threshold_KPCA)
KPCAComp=indexKPCA[0][0]

# Selection of the eigenvectors
Yr_train_KPCA=YKpca[:,:KPCAComp]
Anr_train_KPCA=AnKpca[:,:KPCAComp]

# Construction matrix K for test
N = Xctrain.shape[0]
M = Xctest.shape[0]
```

```
InnerX = np.dot(Xctest,Xctrain.T)
tempTrain=np.sum(Xctrain**2,axis=1).reshape((1,N))
tempTest=np.sum(Xctest**2,axis=1).reshape((M,1))
NormTrain2 = np.repeat(tempTrain,M,axis=0)
NormTest2 = np.repeat(tempTest,N,axis=1)
Norm = NormTest2+NormTrain2-2*InnerX
Norm[Norm<1e-10]=0
Ktest=np.exp(-Norm/(2*gamma**2))

# Centering kernel test matrix
oneN=np.ones((N,N))/N
oneM=np.ones((M,M))/M
KcTest=Ktest-np.dot(oneM,Ktest)-np.dot(Ktest,oneN) + np.dot(np.
 ↪dot(oneM,Ktest),oneN) # center kernel matrix

# Computation of the test scores using the eigenvectors computed with the␣
 ↪training data-set
Yr_test_KPCA=np.dot(KcTest,Anr_train_KPCA)

print('KPCA uses ', Yr_train_KPCA.shape[0], ' features')


# Score
NN.fit(Yr_train_KPCA,Id_Train.ravel())
print('Percentage of correct answer using KPCA is ', NN.
 ↪score(Yr_test_KPCA,Id_Test.ravel()))
```

```
KPCA uses  1931  features
Percentage of correct answer using KPCA is  0.6480331262939959
```

[15]:
```
# For IMP
## KPCA (your implementation)
YKpca, AnKpca, DKpca, varexplKpca = KpcaGaussianLecture(Xctrain,gamma=4)
# Threshold defined as 99% of the variability
Threshold_KPCA = 0.99
CumulativeKPca=np.cumsum(varexplKpca)
indexKPCA=np.argwhere(CumulativeKPca>Threshold_KPCA)
KPCAComp=indexKPCA[0][0]

# Selection of the eigenvectors
Yr_train_KPCA=YKpca[:,:KPCAComp]
Anr_train_KPCA=AnKpca[:,:KPCAComp]

# Construction matrix K for test
N = Xctrain.shape[0]
M = Xctest.shape[0]
InnerX = np.dot(Xctest,Xctrain.T)
```

```
tempTrain=np.sum(Xctrain**2,axis=1).reshape((1,N))
tempTest=np.sum(Xctest**2,axis=1).reshape((M,1))
NormTrain2 = np.repeat(tempTrain,M,axis=0)
NormTest2 = np.repeat(tempTest,N,axis=1)
Norm = NormTest2+NormTrain2-2*InnerX
Norm[Norm<1e-10]=0
Ktest=np.exp(-Norm/(2*gamma**2))

# Centering kernel test matrix
oneN=np.ones((N,N))/N
oneM=np.ones((M,M))/M
KcTest=Ktest-np.dot(oneM,Ktest)-np.dot(Ktest,oneN) + np.dot(np.
 ↪dot(oneM,Ktest),oneN) # center kernel matrix

# Computation of the test scores using the eigenvectors computed with the␣
 ↪training data-set
Yr_test_KPCA=np.dot(KcTest,Anr_train_KPCA)

print('KPCA uses ', Yr_train_KPCA.shape[0], ' features')


# Score
NN.fit(Yr_train_KPCA,Id_Train.ravel())
print('Percentage of correct answer using KPCA is ', NN.
 ↪score(Yr_test_KPCA,Id_Test.ravel()))
```

```
KPCA uses  1931  features
Percentage of correct answer using KPCA is  0.7329192546583851
```

4 ICA

In the next section you will evaluate ICA. Every image x_i can be seen as a linear combination of basis images. ICA can be used in two different ways for face recognition. We can look for a set of statistically independent basis images s_j (first architecture) or for a set of statistically independent coefficients a_j (second architecture).

In the first architecture, we compute $X' = A'S'$, where every row of X' is an image and the columns are pixels. Images are considered as random variables and we look for a set of statistically independent basis images contained in the rows of S'.

In the second architecture, we transpose the previous setting computing $X'' = A''S''$, where every column of X'' is an image and rows are pixels. In this case, we consider the pixels as random variables and we look for a set of statistically independent coefficients contained in the rows of S and a set of basis images in the columns of A.

Instead than using the original training data X as input matrix, we are going to use the eigenvectors (first architecture) or the scores (second architecture) computed with PCA, namely $Y = XL$ (same notation as in the slides of the lecture). In this way, we reduce the computational time since the number of eigenvectors that account for 99% of the variance of the training images (columns of L)

is definitely lower than the number of pixels (columns of X). If you want, you can of course use the original data but it will take much more time to converge.

For the first architecture we will use L^T as input matrix. In fact, we can notice that the PCA approximation of the matrix X_{train}, containing an image in every row, can be written as $\tilde{X} = YL^T$. If we use L^T as input in the ICA algorithm we obtain $L^T = AS$, thus it follows that $\tilde{X} = YW^TS$ (since $A = W^{-1} = W^T$). The basis images are contained in the rows of S and the coefficients used for evaluating the performance are instead contained in the rows of $Y_{train}W^T$ for the training set and in $Y_{test}W^T$ for the test set.

For the second architecture, we will instead use Y^T as input matrix thus obtaining $Y^T = AS$. Remember that in the second architecture we want to apply the ICA algorithm to the transpose of X_{train}, namely $X^T = AS$. We can notice that, given the PCA transformation $Y = XL$, one can write $X \approx YL^T$ which entails $X^T \approx LY^T = LAS = LW^TS$. The columns of LW^T contain the basis images whereas the columns of S contain the statistically independent coefficients used to test the performance of the algorithm. The coefficients for the test set are in the columns of $S_{test} = W_{train}Y_{test}^T$.

NB: Here we used $X = X_c$ which means centered face images

Questions:

1. (IMP+IMH) Look at the results of the two architecures. Which one is better ?
2. (IMP+IMH) Looking at the basis images, in which case do they seem more 'real' ?
3. (IMP - Optional) Implement the two architecture using your own imlementation of ICA (`FastICALecture`) instead than the one of scikit-learn (Be careful at the input data...scikit-learn wants a [observations, features] matrix) **Answers:**
4. I think the second architecture look better although both of them see to lack a main feature of the face
5. still looks more real in second architecture but still, something looks wired about it, and it's hard to recognize each face

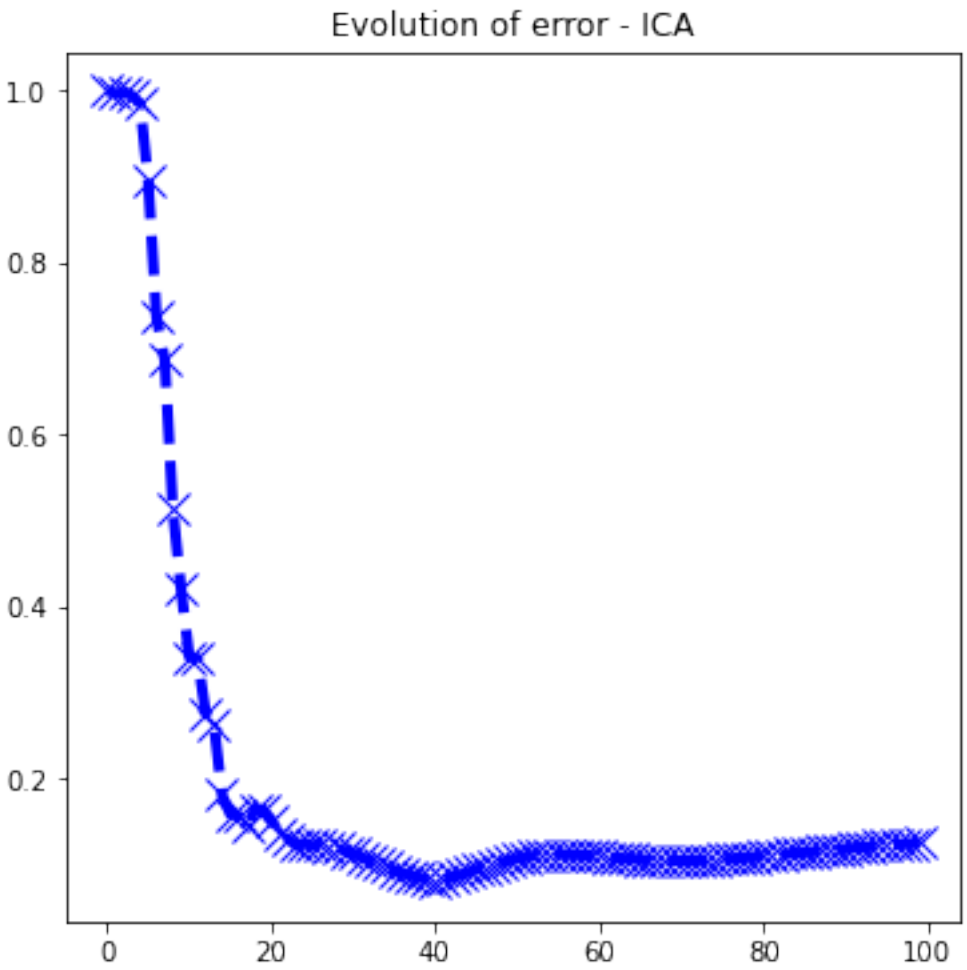
[16]:
```
#ICA - first architecture (scikit-learn implementation)
pca = PCA(random_state=1) # by fixing the random_state we are sure that results␣
 ↪are always the same
YpcaTrain=pca.fit_transform(Xtrain)
UpcaTrain=pca.components_.T # we want PC on columns
var_explained_pca=pca.explained_variance_ratio_

# We use the PCA projection to speed up results
# Threshold defined as 99% of the variability
Threshold_PCA = 0.99
CumulativePca=np.cumsum(var_explained_pca)
indexPCA=np.argwhere(CumulativePca>Threshold_PCA)
PCAComp=indexPCA[0][0]

# Selection of the eigenvectors
Yr_train_PCA=YpcaTrain[:,:PCAComp]
```

```
Ur_train_PCA=UpcaTrain[:,:PCAComp]
Yr_test_PCA=np.dot(Xctest,Ur_train_PCA)

ICA= FastICA(whiten=True, fun='exp', max_iter=30000, algorithm='parallel',␣
 ↪tol=1e-4)
Yica=ICA.fit_transform(Ur_train_PCA)
Sica=Yica.T
Aica=ICA.mixing_
Wica=ICA.components_
Y_test_ICA= np.dot(Yr_test_PCA, Aica)
Y_train_ICA = np.dot(Yr_train_PCA,Aica)

print('ICA uses ', Y_train_ICA.shape[0], ' features')

# Plot the Eigenfaces
plotFaces(Sica,r,c,ncol=2,indeces=np.arange(0,10,1),title='ICA - first␣
 ↪architecture')

# Score
NN.fit(Y_train_ICA,Id_Train.ravel())
print('Percentage of correct answer using ICA arch.1 is ', NN.
 ↪score(Y_test_ICA,Id_Test.ravel()))
```

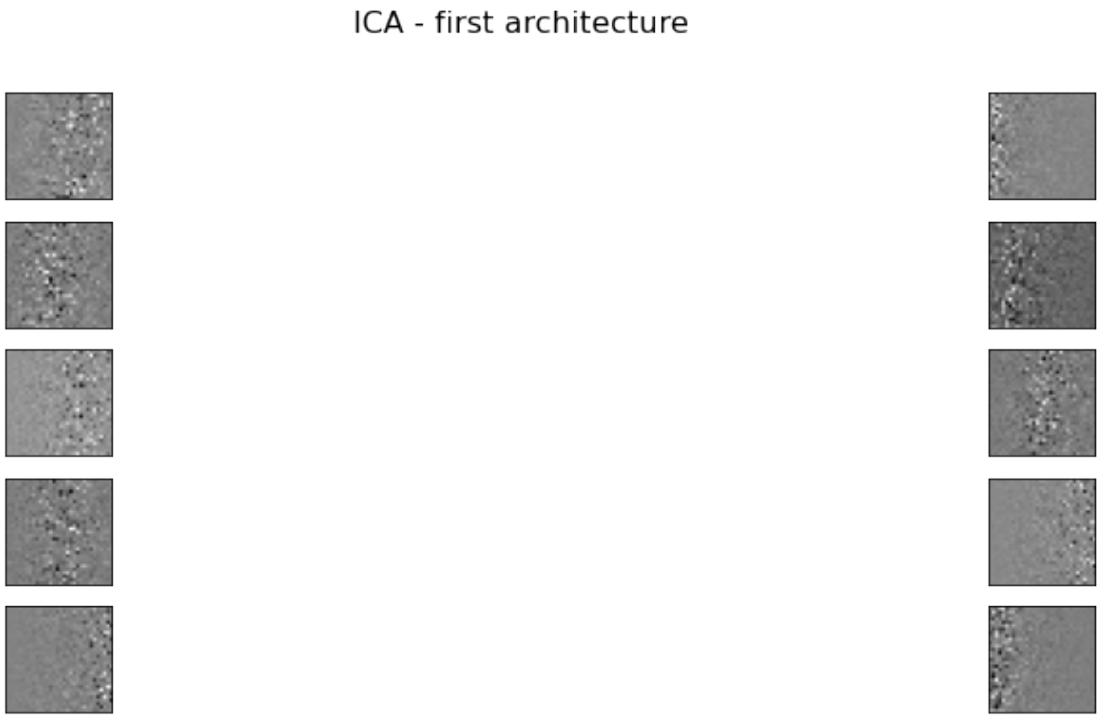
```
ICA uses  1931  features
Percentage of correct answer using ICA arch.1 is  0.7908902691511387
```

[17]:
```
# For IMP
# ICA - First architecture (your implementation)
YpcaTrain,UpcaTrain,_,var_explained_pca=pcaLecture(Xtrain)
UpcaTrain=UpcaTrain.T
# Threshold defined as 99% of the variability
Threshold_PCA = 99
CumulativePca=np.cumsum(var_explained_pca)
indexPCA=np.argwhere(CumulativePca>Threshold_PCA)
PCAComp=indexPCA[0][0]

print('PCA uses ', PCAComp, ' features')

# Selection of the eigenvectors
Yr_train_PCA=YpcaTrain[:,:PCAComp]
Ur_train_PCA=UpcaTrain[:,:PCAComp]

# Computation of the test scores using the eigenvectors computed with the
# training data-set
Yr_test_PCA=np.dot(Xctest,Ur_train_PCA)

Sica,Wica = FastICALecture(Ur_train_PCA.T,N_Iter=100,tol=1e-4,plot_evolution=1)
Aica = LA.pinv(Wica)
Y_test_ICA= np.dot(Yr_test_PCA, Aica)
Y_train_ICA = np.dot(Yr_train_PCA,Aica)

print('ICA uses ', Y_train_ICA.shape[0], ' features')

# Plot the Eigenfaces
# plotFaces(Sica.T,r,c,ncol=2,indeces=np.arange(0,10,1),title='ICA - first␣
 ↪architecture')
plotFaces(Sica,r,c,ncol=2,indeces=np.arange(0,10,1),title='ICA - first␣
 ↪architecture')

# Score
NN.fit(Y_train_ICA,Id_Train.ravel())
print('Percentage of correct answer using ICA arch.1 is ', NN.
 ↪score(Y_test_ICA,Id_Test.ravel()))
```

```
PCA uses  224  features
Iteration ICA number  1  out of  100 , delta =  0.999526766505901
Iteration ICA number  100  out of  100 , delta =  0.12505309429582745
Maximum number of iterations reached ! delta =  0.12505309429582745
```



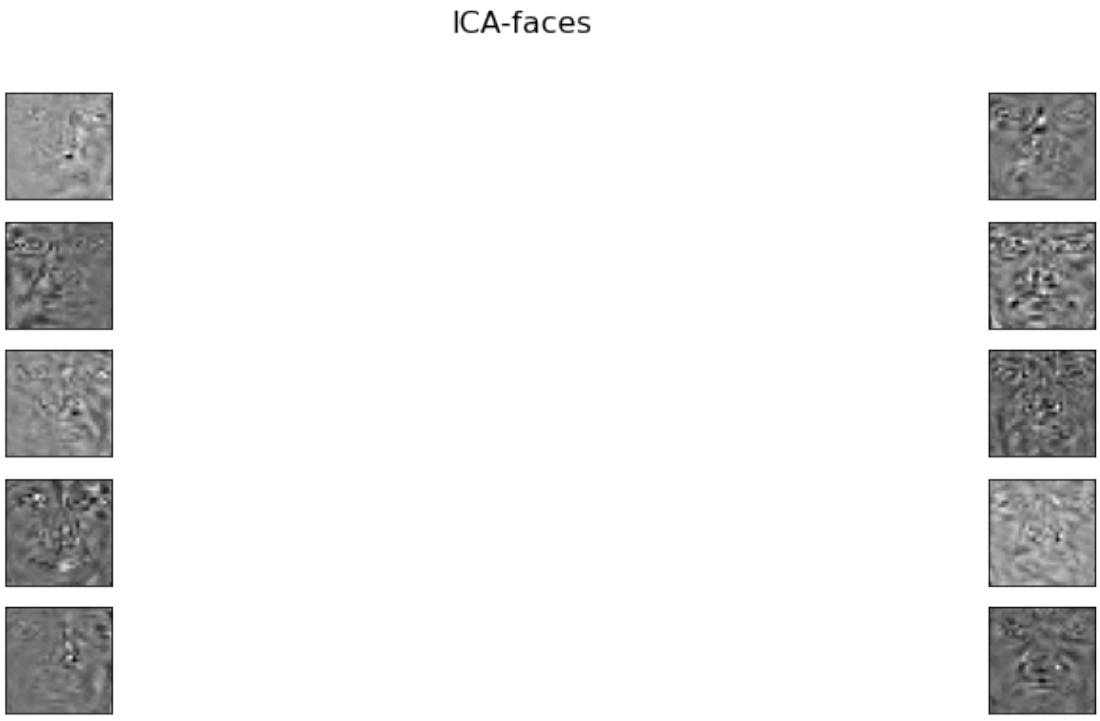

```
ICA uses  1931  features
Percentage of correct answer using ICA arch.1 is  0.6625258799171843
```



[18]:
```
## ICA
# Second architecture (scikit-learn implementation)
pca = PCA(random_state=1) # by fixing the random_state we are sure that results␣
 ↪are always the same
YpcaTrain=pca.fit_transform(Xtrain)
UpcaTrain=pca.components_.T # we want PC on columns
var_explained_pca=pca.explained_variance_ratio_

# Threshold defined as 99% of the variability
Threshold_PCA = 0.99
CumulativePca=np.cumsum(var_explained_pca)
indexPCA=np.argwhere(CumulativePca>Threshold_PCA)
PCAComp=indexPCA[0][0]

# Selection of the eigenvectors
Yr_train_PCA=YpcaTrain[:,:PCAComp]
Ur_train_PCA=UpcaTrain[:,:PCAComp]
Yr_test_PCA=np.dot(Xctest,Ur_train_PCA)

ICA= FastICA(whiten=True, fun='exp', max_iter=30000, tol=1e-4,␣
 ↪algorithm='parallel', random_state=1)
Yica=ICA.fit_transform(Yr_train_PCA)
S_train_ICA=Yica.T
```

```
W_train_ICA=ICA.components_

ICAFAces=np.dot(Ur_train_PCA,W_train_ICA.T)
Y_train_ICA=S_train_ICA
Y_test_ICA=np.dot(W_train_ICA,Yr_test_PCA.T)

# Plot the ICA-faces
plotFaces(ICAFAces.T,r,c,ncol=2,indeces=np.arange(0,10,1),title='ICA-faces')

print('ICA uses ', Y_train_ICA.shape[0], ' features')

# Score ICA

NN.fit(Y_train_ICA.T,Id_Train.ravel())
print('Percentage of correct answer using ICA is ', NN.score(Y_test_ICA.
 ↪T,Id_Test.ravel()))
```

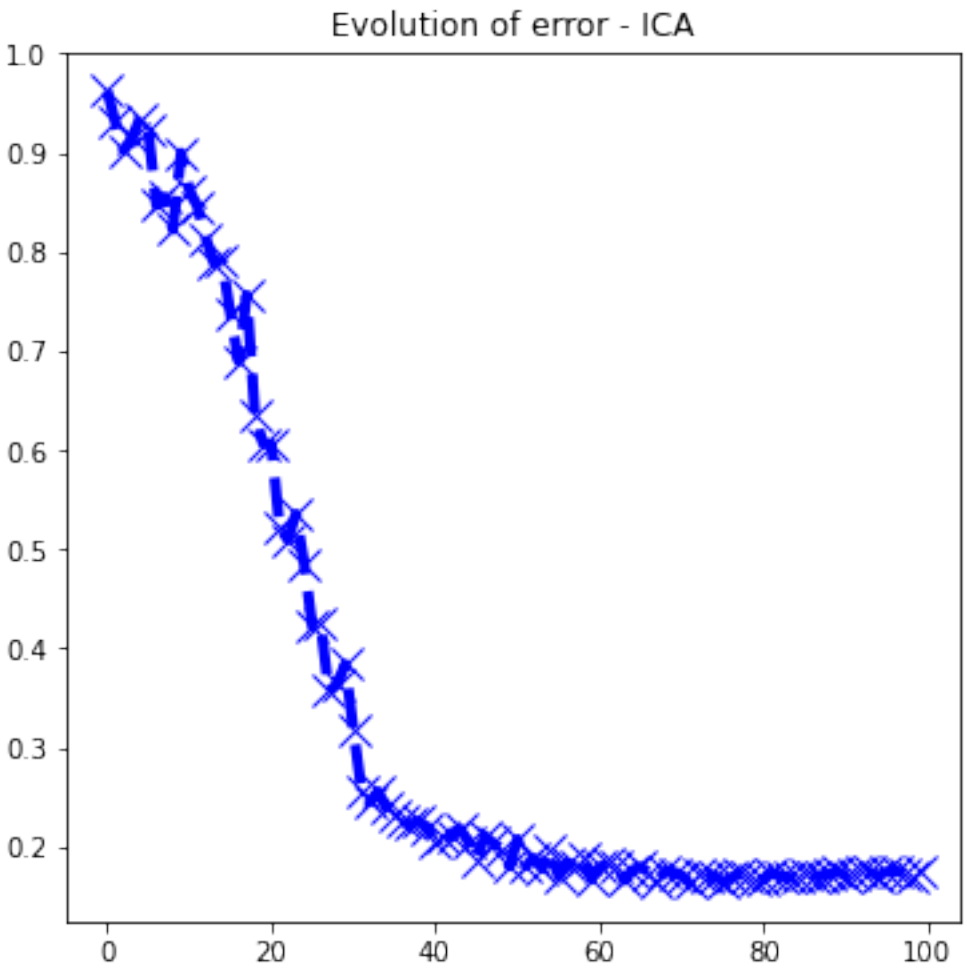
```
ICA uses  232  features
Percentage of correct answer using ICA is  0.9213250517598344
```



[19]:
```
# For IMP
## ICA - second architecture (your implementation)
YpcaTrain,UpcaTrain,_,var_explained_pca=pcaLecture(Xtrain)
UpcaTrain=UpcaTrain.T
```

```
# Threshold defined as 99% of the variability
Threshold_PCA = 99
CumulativePca=np.cumsum(var_explained_pca)
indexPCA=np.argwhere(CumulativePca>Threshold_PCA)
PCAComp=indexPCA[0][0]

print('PCA uses ', PCAComp, ' features')

# Selection of the eigenvectors
Yr_train_PCA=YpcaTrain[:,:PCAComp]
Ur_train_PCA=UpcaTrain[:,:PCAComp]

# Computation of the test scores using the eigenvectors computed with the
# training data-set
Yr_test_PCA=np.dot(Xctest,Ur_train_PCA)

S_train_ICA,W_train_ICA = FastICALecture(Yr_train_PCA.
 ↪T,N_Iter=100,tol=1e-4,plot_evolution=1)

ICAFAces=np.dot(Ur_train_PCA,W_train_ICA)
Y_train_ICA=S_train_ICA
Y_test_ICA=np.dot(W_train_ICA,Yr_test_PCA.T)

# Plot the ICA-faces
plotFaces(ICAFAces.T,r,c,ncol=2,indeces=np.arange(0,10,1),title='ICA-faces')

print('ICA uses ', Y_train_ICA.shape[0], ' features')

# Score ICA

NN.fit(Y_train_ICA.T,Id_Train.ravel())
print('Percentage of correct answer using ICA is ', NN.score(Y_test_ICA.
 ↪T,Id_Test.ravel()))
```

```
PCA uses  224  features
Iteration ICA number  1  out of  100 , delta =  0.962061629336125
Iteration ICA number  100  out of  100 , delta =  0.1757210766010151
Maximum number of iterations reached ! delta =  0.1757210766010151
```




```
ICA uses  224  features
Percentage of correct answer using ICA is  0.028985507246376812
```

5 NNMF

Here you will test Non-negative Matrix factorization. The basis images of the training are in the matrix W_{train} and the scores (or coefficients) to test the performance in H_{train}. The test scores are computed as $H_{test} = W_{train}^{-1} X_{test}$.

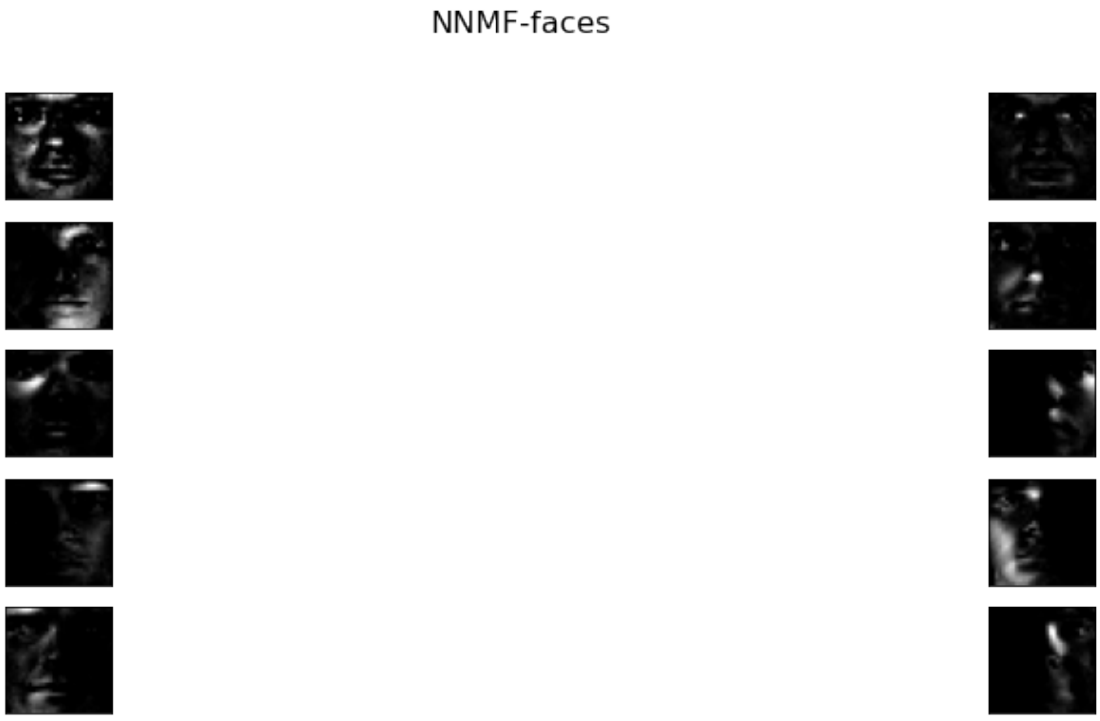
Question

1. (IMP+IMH) Plot the basis images and compare them with respect to the basis images obtained using PCA and ICA. What can you say ?
2. (IMP+IMH) What about the performances of NNMF, i.e. computational time and classification accuracy ? Is it better or worse than the other methods ? Why ?
3. (IMP) Implement your own implementation in `NNMFLecture` following the lecture slides. Complete the missing lines (`XXXXXXXXXX`)

[20]:
```
# NNMF (scikit-learn implementation)
Ncomponents=100 # choose a number of componenents (r in the slides)
model = NMF(init='random', solver='mu', n_components=Ncomponents,␣
 ↪tol=1e-3,max_iter=300, random_state=0)
WtrainNNMF = model.fit_transform(Xtrain.T)
HtrainNNMF = model.components_

plotFaces(WtrainNNMF.T,r,c,ncol=2,indeces=np.arange(0,10,1),title='NNMF-faces')
```

```
Htest_nnmf = np.dot(LA.pinv(WtrainNNMF),Xtest.T)

print('NNMF uses ', Ncomponents, ' features')

# Score
NN.fit(HtrainNNMF.T,Id_Train.ravel())
print('Percentage of correct answer using NNMF is ', NN.score(Htest_nnmf.
 ↪T,Id_Test))
```

```
NNMF uses  100  features
Percentage of correct answer using NNMF is  0.8737060041407867
```



[21]:
```
# For IMP
def NNMFLecture(X,r=None,N_Iter=1000,tolerance=1e-3,plot_evolution=1):
    '''
    Inputs:
    %           X: is a [dxN] matrix. Every column (x) is an observation and␣
 ↪every
    %           row consists of features.
    %
    %           r: size of the matrices W and H
    %
    %           (Optional) N_Iter: maximum number of iterations
    %
```

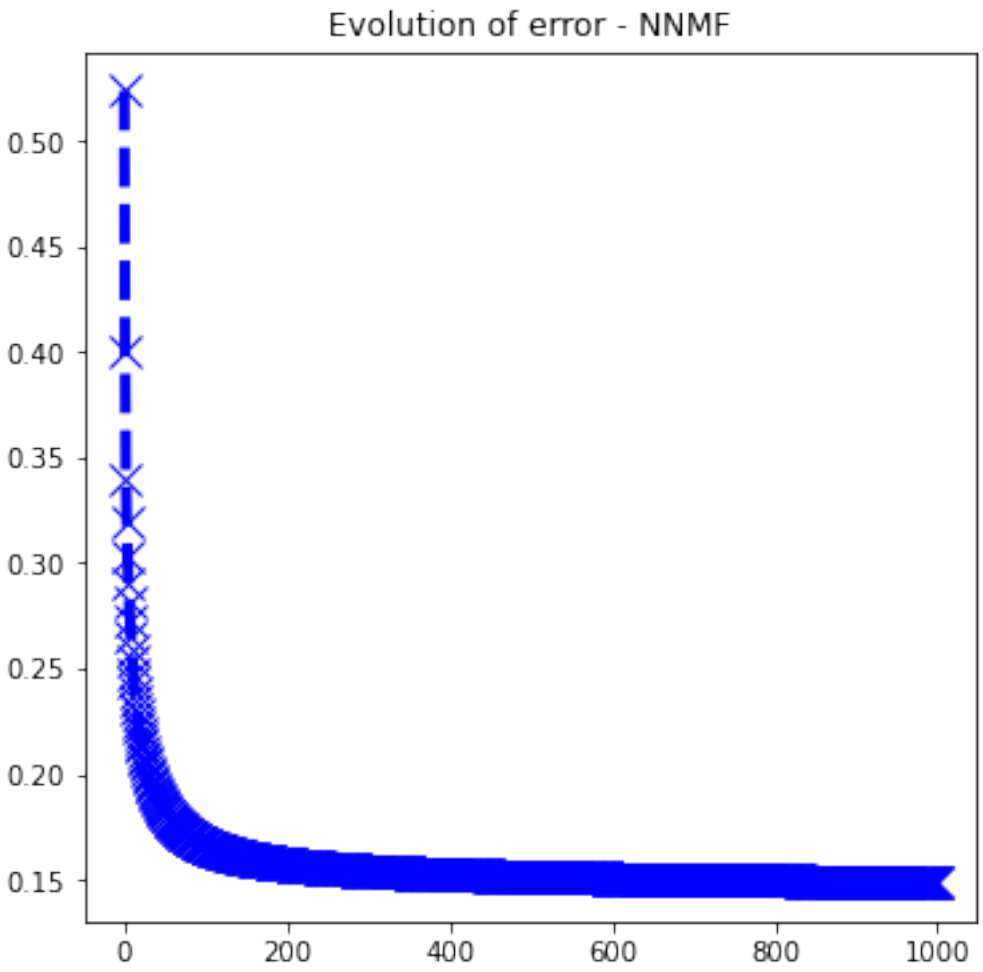
```
    %           (Optional) tolerance: convergence criteria threshold
    %
    %           (Optional) plot_evolution: plot evolution convergence criteria
    %
    % Outputs:
    %           W: is a [d x r] matrix containing the basis images in its
    %           columns
    %
    %           H: is a [r x N] matrix containing the loadings (h) in its␣
↪columns
    %           of the linear combination: x=Wh
    %
  '''
    if r is None:
        r=X.shape[0]

    # Test for positive values
    if np.min(X) < 0:
        raise NameError('Input matrix X has negative values !')

    # Size
    d,N=X.shape

    # Initialization
    W=np.random.rand(d,r)
    H=np.random.rand(r,N)

    # parameters for convergence
    k = 0
    delta = np.inf
    eps=np.finfo(float).eps
    evolutionDelta=[]

    while delta > tolerance and k < N_Iter:

        # multiplicative method
        for i in range(20):
            XtW = np.dot(W.T, X)
            HWtW = np.dot(W.T.dot(W), H ) + eps
            H *= XtW
            H /= HWtW
            #W = np.divide(XXXXXXXX + eps)

        XH = X.dot(H.T)
        WHtH = W.dot(H.dot(H.T)) + eps
        W *= XH
        W /= WHtH
```

```
        #H = np.divide(XXXXXXXXXX + eps)

        # Convergence indices
        k = k + 1
        diff=X-np.dot(W,H)
        delta = LA.norm(diff,'fro') / LA.norm(X,'fro') #␣
↪sqrt(trace(diff'*diff)) / sqrt(trace(X'*X))
        evolutionDelta.append(delta)

        if k==1 or k%100==0:
            print('Iteration NNMF number ', k, ' out of ', N_Iter , ', delta =␣
↪', delta, ', error (norm delta): ', LA.norm(diff))

    if k==N_Iter:
        print('Maximum number of iterations reached ! delta = ', delta)
    else:
        print('Convergence achieved ( delta = ', delta, ') in ', k, '␣
↪iterations')

    if plot_evolution==1:
        plt.figure(figsize=(6, 6))
        plt.plot(range(k),evolutionDelta,'bx--', linewidth=4, markersize=12)
        plt.title('Evolution of error - NNMF')
        plt.show()

    return W,H
```

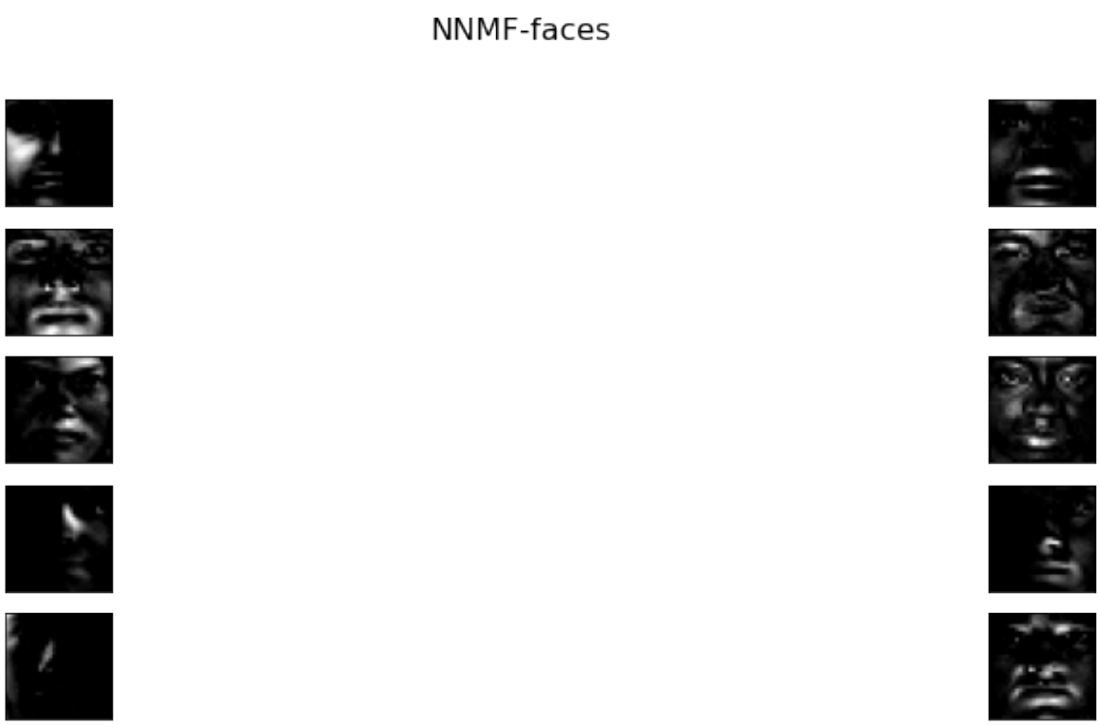
[22]:
```
# For IMP
# NNMF (test your own implementation)
Ncomponents=100 # choose a number of componenents (r in the slides)
#model = NMF(init='random', solver='mu', n_components=Ncomponents,␣
↪tol=1e-3,max_iter=300, random_state=0)
WtrainNNMF,HtrainNNMF = NNMFLecture(Xtrain.
↪T,r=Ncomponents,N_Iter=1000,tolerance=1e-3,plot_evolution=1)

plotFaces(WtrainNNMF.T,r,c,ncol=2,indeces=np.arange(0,10,1),title='NNMF-faces')

Htest_nnmf = np.dot(LA.pinv(WtrainNNMF),Xtest.T)

print('NNMF uses ', Ncomponents, ' features')

# Score
NN.fit(HtrainNNMF.T,Id_Train.ravel())
print('Percentage of correct answer using NNMF is ', NN.score(Htest_nnmf.
↪T,Id_Test))
```

```
Iteration NNMF number  1  out of  1000 , delta =  0.5234993370961607 , error
```

```
(norm delta):  269.4748841388608
Iteration NNMF number  100  out of  1000 , delta =  0.16618000171078148 , error
(norm delta):  85.5422987841964
Iteration NNMF number  200  out of  1000 , delta =  0.1573324495667057 , error
(norm delta):  80.98796046896105
Iteration NNMF number  300  out of  1000 , delta =  0.15422315036011935 , error
(norm delta):  79.38742731815402
Iteration NNMF number  400  out of  1000 , delta =  0.1525780781852839 , error
(norm delta):  78.54061510216755
Iteration NNMF number  500  out of  1000 , delta =  0.15150414314675903 , error
(norm delta):  77.9877996550947
Iteration NNMF number  600  out of  1000 , delta =  0.15070609449727382 , error
(norm delta):  77.57699862418968
Iteration NNMF number  700  out of  1000 , delta =  0.15010056436902983 , error
(norm delta):  77.26529782614041
Iteration NNMF number  800  out of  1000 , delta =  0.1496089291121879 , error
(norm delta):  77.01222519645773
Iteration NNMF number  900  out of  1000 , delta =  0.14920882063466784 , error
(norm delta):  76.80626660590677
Iteration NNMF number  1000  out of  1000 , delta =  0.14887986058604663 , error
(norm delta):  76.63693215845505
Maximum number of iterations reached ! delta =  0.14887986058604663
```




```
NNMF uses  100  features
Percentage of correct answer using NNMF is  0.8571428571428571
```



Here, we check that using the original data for ICA is definitely too long. In scikit-learn we directly select the number of components p. However, results are less satisfactory than using PCA as before or too long.

[ ]:
```
# Shuffle data randomly and use only 300 components (p=300)
indeces=np.arange(data.shape[0]) # Integers from 0 to N-1
np.random.shuffle(indeces)
sample=data[indeces,:]

#first architecture (scikit-learn implementation)
ICA= FastICA(whiten=True, fun='exp', max_iter=30000, algorithm='parallel',␣
 ↪tol=1e-4, random_state=1, n_components=300)
Yica=ICA.fit_transform(sample)
Sica=Yica.T
Aica=ICA.mixing_
Wica=ICA.components_

# Plot the Eigenfaces
plotFaces(Sica,r,c,ncol=2,indeces=np.arange(0,10,1),title='ICA basis images␣
 ↪(first architecture)')
```

[ ]:
```
#second architecture (scikit-learn implementation)
ICA= FastICA(whiten=True, fun='exp', max_iter=30000, algorithm='parallel',␣
 ↪tol=1e-4, random_state=1, n_components=300)
```

```
Yica=ICA.fit_transform(data.T)
Sica=Yica.T
Aica=ICA.mixing_
Wica=ICA.components_

# Plot the Eigenfaces
plotFaces(Aica.T,r,c,ncol=2,indeces=np.arange(0,10,1),title='ICA basis images␣
 ↪(second architecture)')
```

[ ]: