Feature extraction

Applications of Machine Learning

Vanessa Gómez Verdejo vanessag@ing.uc3m.es, *Emilio Parrado Hernández* *eparrado@ing.uc3m.es*

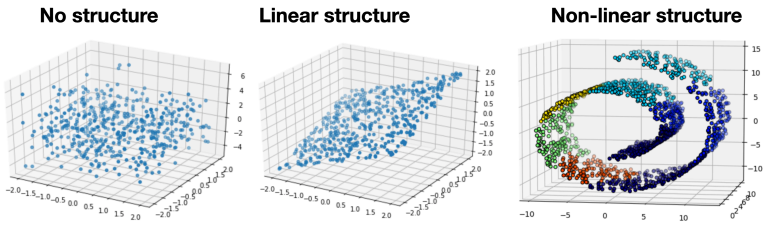
In [17]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

1. Introduction

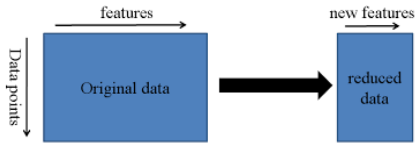
One of the main limitations of the use of machine learning technology in industrial applications is the so-called **Black-Box effect**.

Many machine learning practitioners perceive that they use algorithms that **crunch data and find patterns**. These algorithms are then able to build models upon those patterns and predict more or less accurate outcomes for new (test) data. But these **practitioners can't explain** why the algorithm came up with those predictions.

In this sense, machine learning can also be used to gain insights about the data, in order to **unveiling structure in data**:



Specifically, **feature extraction** methods exploit the data structure to reduce the dimensionality of the input space. Simultaneously, these methods use only relevant data, i.e, remove irrelevant/noisy/correlated components minimizing the loss of *relevant* information, and discover good combinations of input variables (feature engineering).



These characteristics allow us to simplify the ML stage since:

- We reduce the number of parameters in the classifier (alliviating the *curse of dimensionality*)
- We can get to *bend* the input space to better fit our task
- We obtain a compact data representation (crucial for large datasets)

Some notation

Consider we have a dataset with N observations where each data, $\mathbf{x}^{(n)} \in \mathbb{R}^D$, belongs to a D dimensional space. That is, each data is a column vector with D elements:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_D \end{bmatrix}.$$

Besides, for the sake of simplicity, let's consider that these data have zero mean.

On the other hand, for each data, we are going to find a new representation $\mathbf{x}'$ in a lower dimensional space ($K < D$):

$$\mathbf{x}' = \begin{bmatrix} x'_1 \\ \vdots \\ x'_K \end{bmatrix}.$$

This new data representation results from the application of a linear transformation to the original data defined by:

$$\mathbf{x}' = \mathbf{U}^\top \mathbf{x}^\top$$

where $\mathbf{U}$ is a transformation matrix of size $D \times K$:

$$\mathbf{U} = \begin{bmatrix} u_{1,1} \dots u_{1,K} \\ \vdots \ddots \vdots \\ u_{D,1} \dots u_{D,K} \end{bmatrix}$$

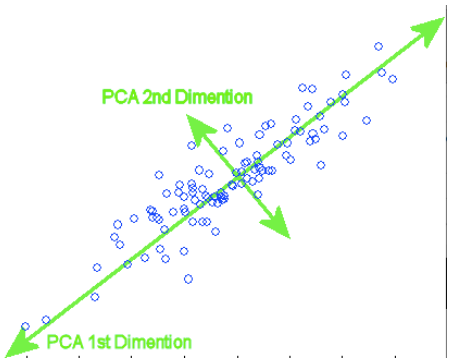
The goal of the **feature extraction** algorithm is to write and solve an optimization

problem that finds out the correct transformation matrix $\mathbf{U}$.

2. Principal Component Analysis (PCA)

Principal Component Analysis can be one of the most mentioned and known data analysis method. It's probably the most advised preprocessing step.

PCA's goal is to obtain a new, reduced set of $K < D$ orthogonal variables, the so-called principal components. The principal components are linear combinations of the input variables, but sorted in decreasing order of variance.



In matrix notation, the purpose of PCA can be written as finding a projection matrix $\mathbf{U}$ and transform the $D \times N$ data matrix $\mathbf{X}$ into the $K \times N$ matrix $\mathbf{X}'$:

$$\mathbf{X}' = \mathbf{U}^\top \mathbf{X}^\top$$

Each column of the matrix $\mathbf{U}$ is a **principal component**, and it will provide a new dimension of the data in the projected space: the principal components act as a vector basis in the projected space.

PCA is **unsupervised**, one does not need targets to compute PCA!!

The PCA algorithm finds the components that maximize the variance contained in the data in a **recursive** way:

- Find the first principal component, $\mathbf{u}_1$, that maximizes the variance of the mapped data: $\mathbf{u}_1^\top \mathbf{x}$ is of maximum variance.
- Deflate the first principal component from the data $\mathbf{x^1} = \mathbf{x} - \mathbf{u}_1^\top \mathbf{x} \mathbf{u}_1$. Deflating = Subtracting the use of the previous eigenvectors to find the second.
- Next, find $\mathbf{u}_2$ so that $\mathbf{u}_2^\top \mathbf{x^1}$ is of maximum variance
- Repeat the deflation of the new principal component from the data and the

determination of the projection $\mathbf{u_j}$that maximizes the variance of the residual $\mathbf{x^{j-1}}$ until obtaining the desired $K < D$ principal components.

2.1 Mathematical formulation

$$\mathbf{U} = \underset{\mathbf{U}}{\operatorname{argmax}}\ \operatorname{Tr}\left\{\mathbf{U}^T\mathbf{X}^T\mathbf{X}\mathbf{U}\right\} = \underset{\mathbf{U}}{\operatorname{argmax}}\ \operatorname{Tr}\left\{\mathbf{U}^T\mathbf{C_{XX}}\mathbf{U}\right\}$$

$$\text{s.t. } \mathbf{U}^T\mathbf{U} = \mathbf{I}$$

Solution:

Let's start considering the projection onto a one-dimensional space ($K = 1$). So, we want to solve the following optimization problem:

$$\mathbf{u}_1 = \underset{\mathbf{u}_1}{\operatorname{argmax}}\ \mathbf{u}_1^\top \mathbf{C_{XX}} \mathbf{u}_1$$

$$\text{s.t. } \mathbf{u}_1^\top \mathbf{u}_1 = 1$$

If we apply Lagrange multipliers, we can include the constraint into functional by means of the lagrange multiplier λ_1 and we arrive to an unconstraint problem

$$\mathbf{u}_1 = \underset{\mathbf{u}_1}{\operatorname{argmax}}\ \mathbf{u}_1^\top \mathbf{C_{XX}} \mathbf{u}_1 + \lambda_1 \left(1 - \mathbf{u}_1^\top \mathbf{u}_1\right)$$

by making its derivate equal to zero, we obtain that the optimum solution has to satisfiy

$$\mathbf{C_{XX}}\mathbf{u}_1 = \lambda_1 \mathbf{u}_1$$

which indicates that $\mathbf{u}_1$ must be an eigenvector of the covariance matrix $\mathbf{C_{XX}}$ and λ_1 is its associated eigenvalue.

Besides, if we left-multiply this expression by $\mathbf{u}_1^\top$ and make use of $\mathbf{u}_1^\top \mathbf{u}_1 = 1$, we obtain that

$$\mathbf{u}_1^\top \mathbf{C_{XX}} \mathbf{u}_1 = \lambda_1$$

that is, the variance of the projected data by the eigenventor $\mathbf{u}_1$ is equal to its associated eigenvalue λ_1. So, we can get the maximum projected variance, with a single projection, if we set this first principal component $\mathbf{u}_1$ as the eigenvector with the largest eigenvalue λ_1. Later, we can define additional principal components in an incremental fashion by choosing each new direction as the one eigenvector with the next highest eigenvalue so that the projected variance is maximized.

So, we can obtain the first K projections of the PCA algorithm by solving the following eigenvalue problem

$$\mathbf{C_{XX}u} = \lambda\mathbf{u}$$

So, the projection matrix, $\mathbf{U}$, consists of the first eigenvectors of $\mathbf{C_{xx}}$ (i.e., those associated with largest eigenvalues)

$$\mathbf{U} = \text{eigs}(\mathbf{C_{XX}})$$

2.2 PCA algorithm

- **Input**: $\mathbf{X}$ ($D \times N$) data matrix, each datum is a column
- **Process**:

  1.- $\mathbf{m} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}^{(n)}$

  2.- $\mathbf{C_{XX}} = \frac{\mathbf{1}}{\mathbf{N}}\sum_{\mathbf{n=1}}^{\mathbf{N}}(\mathbf{x^{(n)}} - \mathbf{m})(\mathbf{x^{(n)}} - \mathbf{m})^{\top}$

  3.- $\mathbf{U, \Lambda = eig(C_{XX})}$

  4.- $\mathbf{X'} = (\mathbf{U_{:K}})^{\top}\mathbf{X}$

- **Output**: Transformed data $\mathbf{X}'$

where eig($\mathbf{A}$) is a function that returns the eigenvectors (columns of $\mathbf{U}$) and eigenvalues (diagonal of Λ) of $\mathbf{A}$ and $\mathbf{U_{:k}}$ means the first k columns of matrix $\mathbf{U}$.

REFERENCE

Jollife I.T. (1986) Principal Component Analysis. Springer-Verlag.

2.3 PCA explains variance

- Each Principal Component **explains an amount of the total variance contained in the data set** equal to the size of the corresponding eigenvalue of $\mathbf{C_{XX}}$. Notice the trace of $\mathbf{C_{XX}}$ (sum of the variances) is the total variance contained in the data and that the sum of the eigenvalues is equal to the trace of a matrix.
- In this sense, the projection of the data on the subset of the first K principal components is the **best reconstruction** of these data with just K components in the sense of maximizing the variance captured by the

reconstruction.

- If the data lies in a **subspace of rank** $K < D$, then the first K principal components are a basis for that subspace and the data is lossless compressed in K components. By lossless we mean without loss of information as these K principal components capture all the variance present in data.
- If the data contains **small amounts of noise**, usually the noise components would reflect in eigenvalues with smaller values, thus removed by throwing away the less principal components.

3. Let's play with data

This section presents some experimental work over a face detection problem where you will be able to test the performance improvements provided by the feature extraction process. Besides, from a qualitative point of view, you will analyze the extracted features by plotting the eigenvectors defining the feature extraction process.

Later, we will analyze the discriminatory capability of the extracted features with a linear SVM classifier. To implement the different approaches we will build on the Scikit-Learn python toolbox.

3.1. Download and prepare the data

This dataset consists of ten different images taken from 40 distinct subjects. For some subjects, the images were taken at different times, varying the lighting, facial expressions (open / closed eyes, smiling / not smiling) and facial details (glasses / no glasses). All the images were taken against a dark homogeneous background with the subjects in an upright, frontal position (with tolerance for some side movement).

The next code includes the lines to download this data set and create the training, validation and test data partitions, as well as normalize them.

In [18]:
```
from sklearn.model_selection import train_test_split
from sklearn.datasets import fetch_olivetti_faces
from sklearn.preprocessing import label_binarize
from sklearn.preprocessing import StandardScaler
```

```
###############################################################################
# Download the data, if not already on disk and load it as numpy array
print('The first time that you download the data it can take a while..
olivetti_people = fetch_olivetti_faces()

# introspect the images arrays to find the shapes (for plotting)
n_samples, h, w = olivetti_people.images.shape

# for machine learning we use the 2 data directly (as relative pixel
# positions info is ignored by this model)
X = olivetti_people.data
n_features = X.shape[1]

# the label to predict is the id of the person
Y = olivetti_people.target
n_classes = np.unique(Y).shape[0]

print("Dataset size information:")
print("n_features: %d" % n_features)
print("n_classes: %d" % n_classes)


###############################################################################
# Preparing the data

# Initialize the random generator seed to compare results
np.random.seed(1)

# Split into a training set and a test set using a stratified k fold

# split into a training and testing set
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.

# split into a training and validation set
X_train, X_val, Y_train, Y_val = train_test_split(X_train, Y_train, te

# Normalizing the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
X_val = scaler.transform(X_val)

# Binarize the labels for supervised feature extraction methods
set_classes = np.unique(Y)
Y_train_bin = label_binarize(Y_train, classes=set_classes)

print("Number of training samples: %d" % X_train.shape[0])
print("Number of validation samples: %d" % X_val.shape[0])
print("Number of test samples: %d" % X_test.shape[0])
```

```
The first time that you download the data it can take a while...
Dataset size information:
n_features: 4096
n_classes: 40
Number of training samples: 200
Number of validation samples: 100
Number of test samples: 100
```

Representing some example faces

Next cells include the `plot_gallery( )` function, which let you plot any of the face images from the data set and in later sections plot the eigenvectors provided by the feature selection process. Besides, as an example, the function is used to plot some training data.

In [19]:
```
import matplotlib.pyplot as plt
def plot_gallery(images, titles, h, w, n_row=4, n_col=10):
    """Helper function to plot a gallery of portraits"""
    plt.figure(figsize=(1.8 * n_col, 2.4 * n_row))
    plt.subplots_adjust(bottom=0, left=.01, right=.99, top=.90, hspace
    for i in range(images.shape[0]):
        plt.subplot(n_row, n_col, i + 1)
        plt.imshow(images[i].reshape((h, w)), cmap=plt.cm.gray)
        plt.colorbar()
        plt.title(titles[i], size=12)
        plt.xticks(())
        plt.yticks(())
```

In [20]:
```
# As example, we plot a face of each class (or person)
titles = ['class '+str(c) for c in set_classes]
ind_faces = [np.where(Y_train == c)[0][0] for c in set_classes]

plot_gallery(X_train[ind_faces,:], titles, h, w)
```

3.2. Feature extraction with PCA

Here, let's use the `PCA( )` method to find the projections maximizing the variance of the projected data. Next cell obtains:

- The first 100 projection vectors from the training data.
- The projections of training, validation and test data in this new space.
- The dimensions (number of data and number of features) of the projected data.

In [21]:
```
from sklearn.decomposition import PCA

N_feat_max=100


my_pca = PCA(n_components=N_feat_max).fit(X_train)
P_train = my_pca.transform(X_train)
P_val = my_pca.transform(X_val)
P_test = my_pca.transform(X_test)

dim_train = P_train.shape[1]
dim_val = P_val.shape[1]
dim_test = P_test.shape[1]


print('Dimensions of training data are: ' + str(dim_train))
print('Dimensions of validation data are: ' + str(dim_val))
print('Dimensions of test data are: ' + str(dim_test))
```

```
Dimensions of training data are: 100
Dimensions of validation data are: 100
Dimensions of test data are: 100
```

3.3 Analyzing eigenvectors and eigenvalues

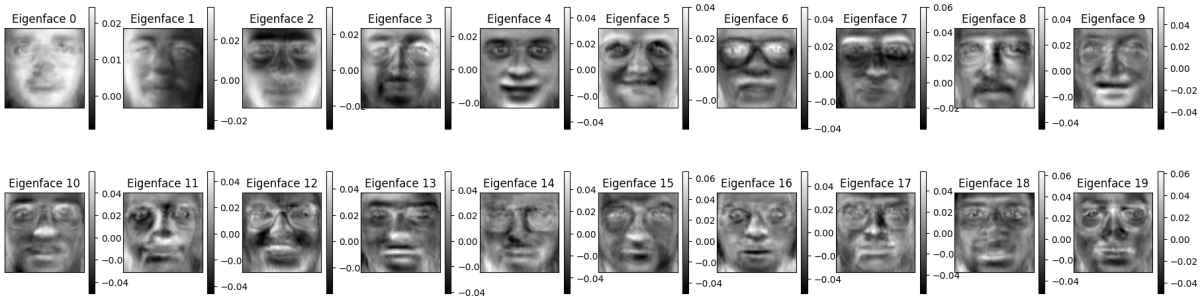
The analysis of eigenvectors in face detection problems is quite common, in fact, they are known as **eigenfaces**. Note that each feature of each transformed instance is the scalar product between the corresponding input face and the corresponding eigenface. Lighter regions of the eigenfaces will contribute positively to generate the new feature, whereas darker regions will contribute negatively.

On the other hand, the **eigenvalues** allow us to know the importance of each principal component (the higher the eigenvalue, the more important the component). Analyzing the eigenvalues we can know which components are the most relevant and, even, know which are unnecessary and how many new

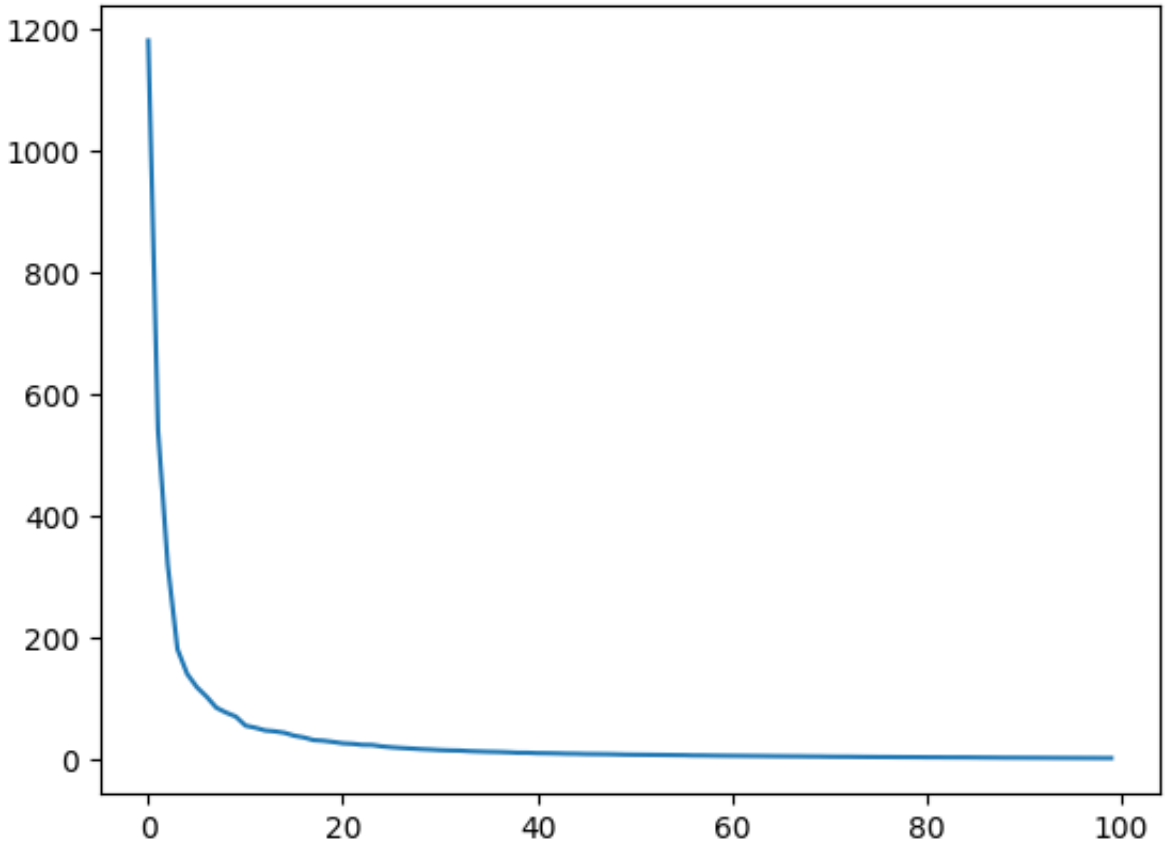
variables we can use without a great loss of information.

Execute the following code cells to plot the first eigenvectors and the evolution of the eigenvalues.

In [22]:
```
# Plot eigenfaces
n_eigenfaces=20
titles = ['Eigenface '+str(num) for num in range(n_eigenfaces)]
eigenfaces = my_pca.components_.reshape((N_feat_max, h, w))
plot_gallery(eigenfaces[:n_eigenfaces,:,:], titles, h, w, n_row=2, n_c
```



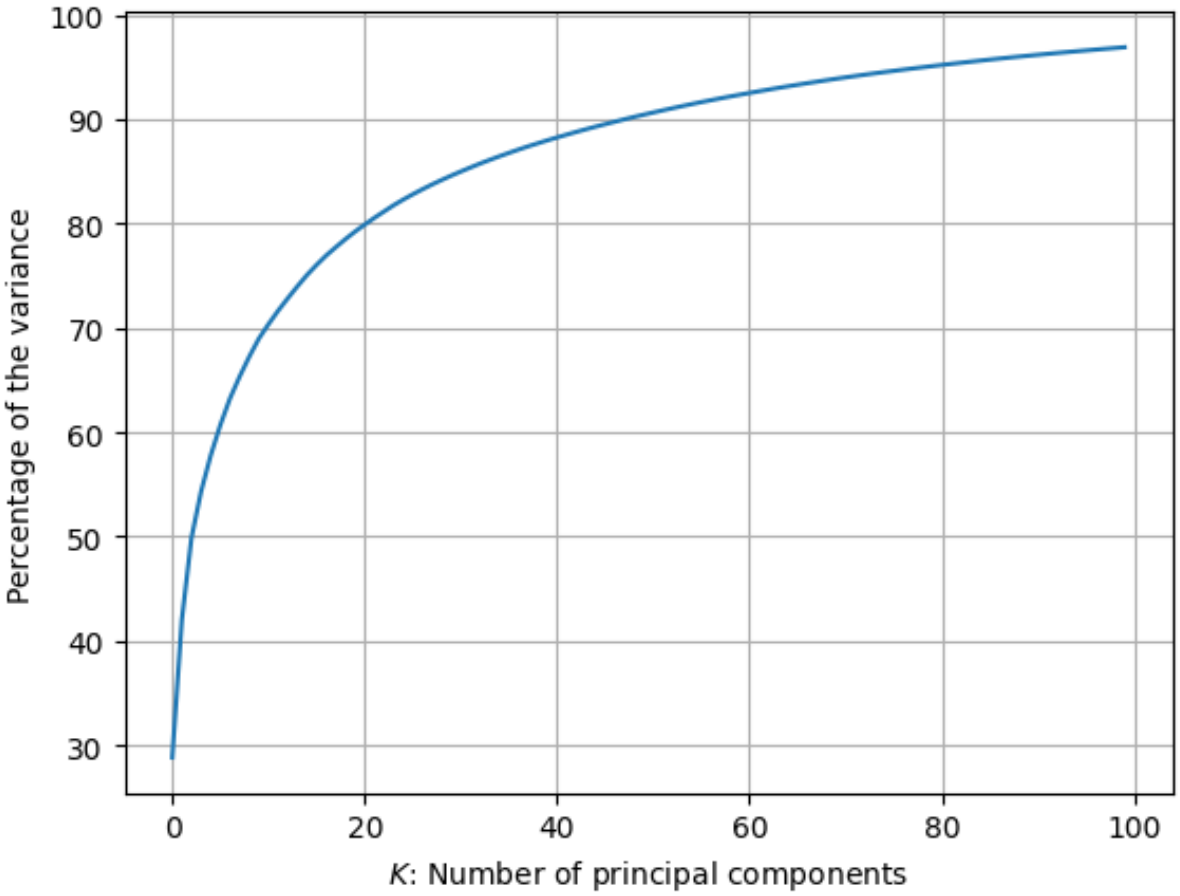
In [23]:
```
# Analyze eigenvalues
eigenvalues = my_pca.explained_variance_
plt.figure()
plt.plot(eigenvalues, label='Evolution eigenvalues')
plt.show()
```



In view of this figure, how many principal components do you think are sufficient to get a good representation of the data and not lose relevant information?

In [24]:
```
# 1.- Compute an estimation of the variance in data using the training

data_variance = np.sum(np.var(X_train,0))

# 2.- plot the cummulative sum of the sorted eigenvalues divided by th
# 2.1.- express the y-axis in percentage
plt.figure()
plt.plot(100*np.cumsum(eigenvalues)/data_variance, label='Evolution ei
plt.grid()
plt.xlabel('$K$: Number of principal components')
plt.ylabel('Percentage of the variance')
```

Out[24]:
```
Text(0, 0.5, 'Percentage of the variance')
```



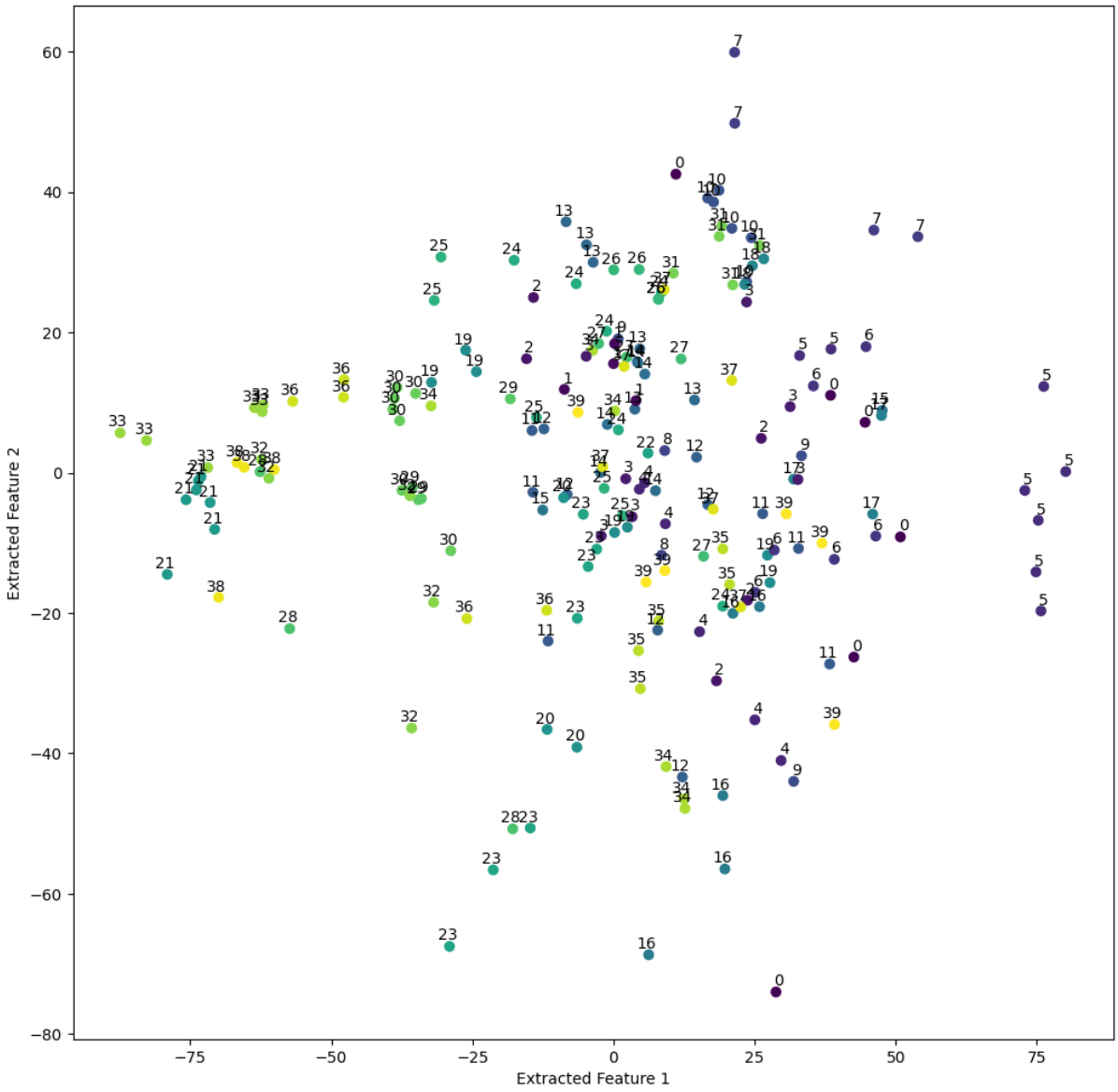
3.4 Dimensionality reduction

The main advantage of PCA lies in the fact that it allows us to reduce the dimensionality of the data. We can even summarize the input data into only 2 or 3 new features that can be used to visualize the data.

The next code cell shows how we can use 2 or 3 of the new features to represent the data of our data set.

In [25]:
```
X_train_pca2 = P_train[:,:2]
plt.figure(figsize=(12,12))
plt.scatter(X_train_pca2[:, 0], X_train_pca2[:, 1], c=Y_train)

for i in range(len(X_train_pca2)):
        plt.annotate(Y_train[i],
                     xy=(X_train_pca2[i,0], X_train_pca2[i,1]),
                     xytext=(5, 2),
                     textcoords='offset points',
                     ha='right',
                     va='bottom')

plt.xlabel('Extracted Feature 1')
plt.ylabel('Extracted Feature 2')
```

Out[25]:
```
Text(0, 0.5, 'Extracted Feature 2')
```



In [26]:
```
from mpl_toolkits.mplot3d import Axes3D
from pylab import figure

X_train_pca3 = P_train[:,:3]
fig = figure(figsize=(12,12)) #plt.figure(figsize=(12,12))
ax = Axes3D(fig)
ax.scatter(X_train_pca3[:, 0], X_train_pca3[:, 1], X_train_pca3[:, 2],

for i in range(len(X_train_pca3)):
  ax.text(X_train_pca3[i,0], X_train_pca3[i,1], X_train_pca3[i,2],  '%

ax.set_xlabel('Extracted Feature 1')
ax.set_ylabel('Extracted Feature 2')
ax.set_zlabel('Extracted Feature 3')
plt.show()
```

```
<Figure size 1200x1200 with 0 Axes>
```

3.5 Reconstruction from the principal components

The Principal Components form a basis that capture the most relevant information within data. Therefore, we can reconstruct the original data as a linear combination of the principal components, precisely weighted by the coordinates of the mapped data in the transformed space:

$$\hat{\mathbf{x}} = \mathbf{U}_{:\mathbf{K}}\mathbf{x}'$$

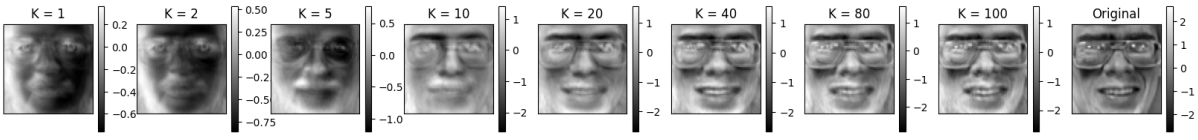
The reconstruction error can be computed as

$$\sqrt{(\mathbf{x} - \hat{\mathbf{x}})^\top (\mathbf{x} - \hat{\mathbf{x}})} =$$

$$\sqrt{\mathbf{x}^\top (\mathbb{I} - \mathbf{U}_{:\mathbf{K}}\mathbf{U}_{:\mathbf{K}}^\top)\mathbf{x}}$$

The following cell plots the result of reconstructing one of the original images with a different number of principal components.

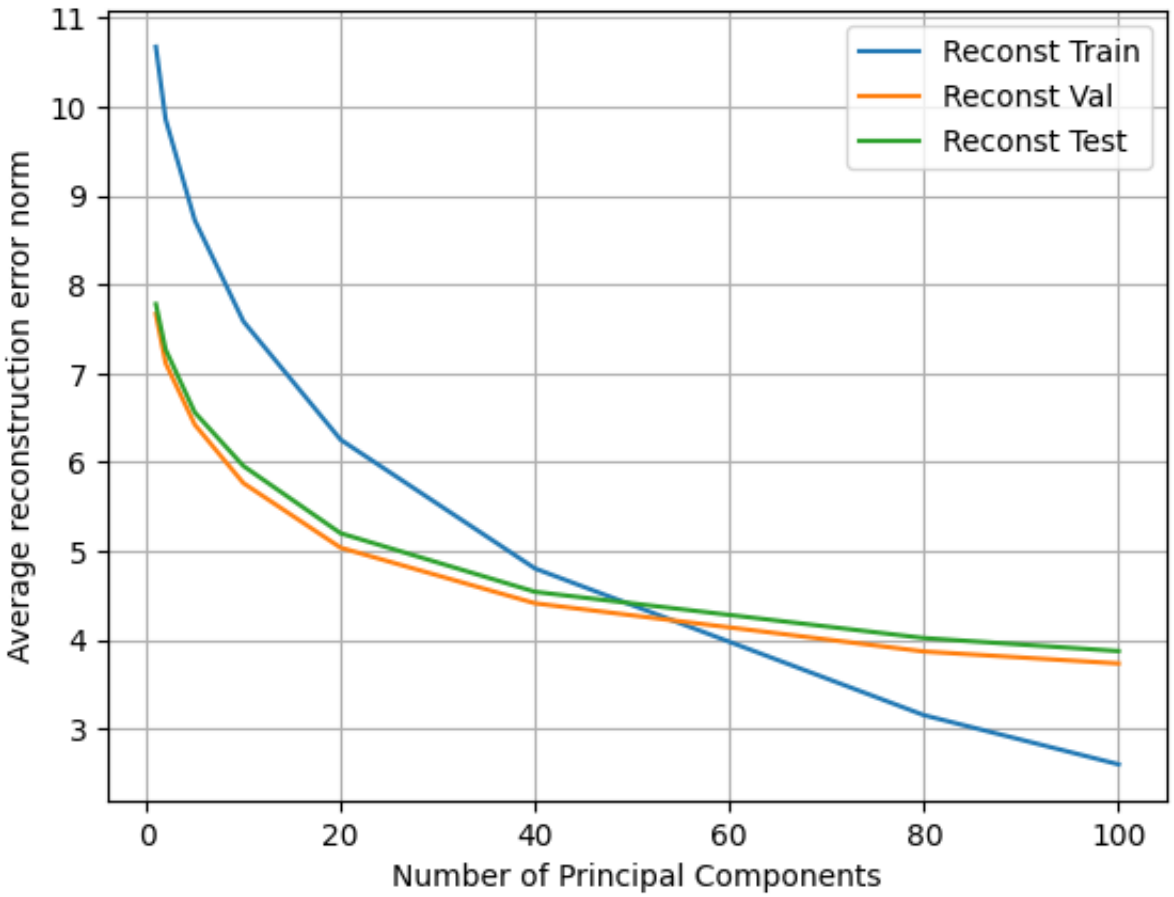
In [27]:
```
subject=27
D = X_train.shape[1]
n_reconst_list=[1,2, 5, 10, 20, 40, 80,  100] # Number of Principal Co
Reconstruct = np.empty((len(n_reconst_list)+1, D))

for ii,n_comp in enumerate(n_reconst_list):
    # Reconstruct image subjet using n_comp PCs
    Reconstruct[ii,:] = P_train[subject,0:n_comp+1].dot(my_pca.compone

# Original Image
Reconstruct[-1,:] = X_train[subject,:].copy()
titles = ['K = '+str(num) for num in n_reconst_list] + ['Original']
Reco = Reconstruct.reshape((len(n_reconst_list)+1, h, w))

plot_gallery(Reco, titles, h, w, n_row=6, n_col=10)
```



Now plot the evolution of the average reconstruction error for the whole training, validation and test sets (one curve per set) as a function of the number of PC used for the reconstruction

In [28]:
```
n_reconst_list=[1,2, 5, 10, 20, 40, 80,  100]
error_train = np.empty(len(n_reconst_list))
error_val = np.empty(len(n_reconst_list))
```

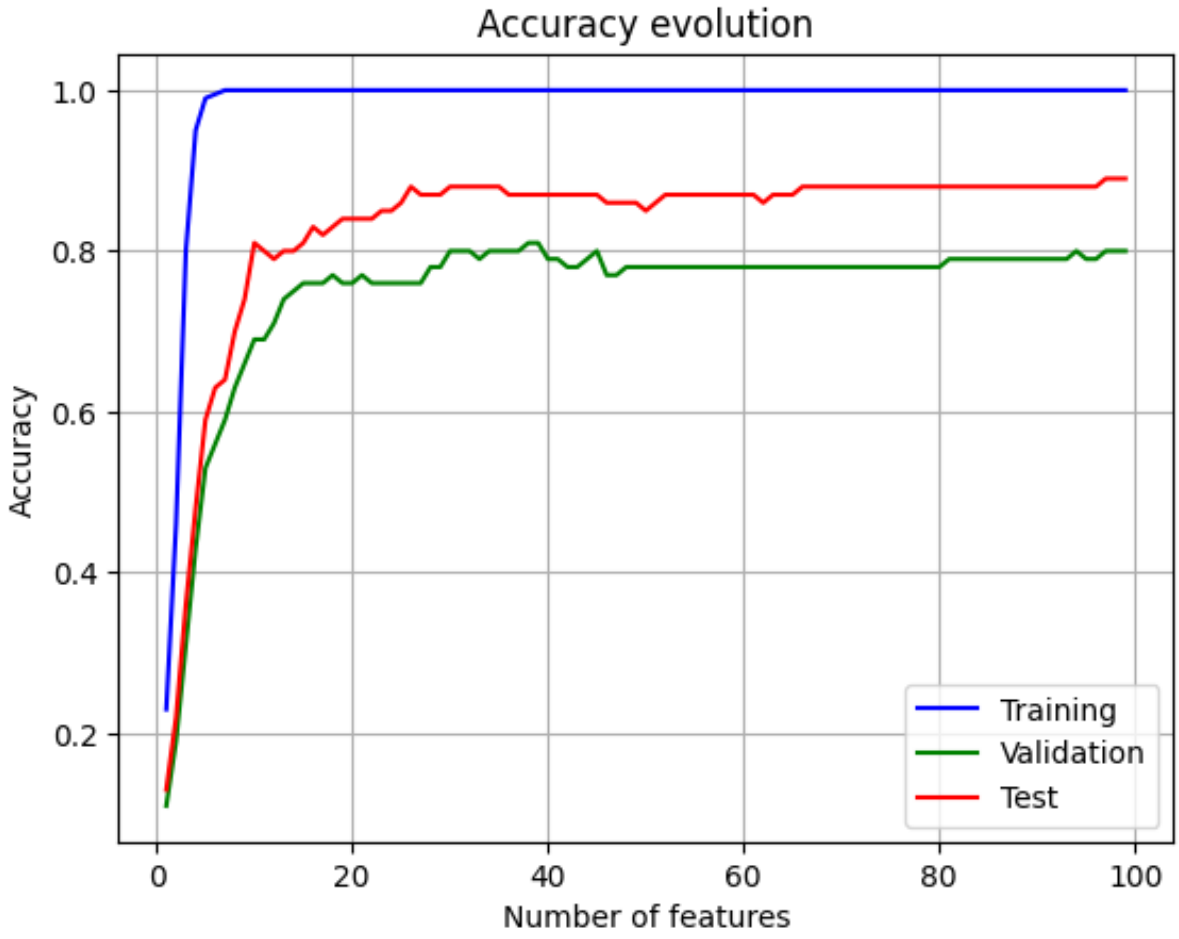
```
error_test = np.empty(len(n_reconst_list))

for ii,n_comp in enumerate(n_reconst_list):
    Reconstruct_train = P_train[:,0:n_comp+1].dot(my_pca.components_[0
    error_train[ii] = np.mean(np.linalg.norm(X_train-Reconstruct_train
    Reconstruct_val = P_val[:,0:n_comp+1].dot(my_pca.components_[0:n_c
    error_val[ii] = np.mean(np.linalg.norm(X_val-Reconstruct_val,axis=
    Reconstruct_test = P_test[:,0:n_comp+1].dot(my_pca.components_[0:n
    error_test[ii] = np.mean(np.linalg.norm(X_test-Reconstruct_test,ax

plt.figure()
plt.plot(n_reconst_list, error_train, label='Reconst Train')
plt.plot(n_reconst_list, error_val, label='Reconst Val')
plt.plot(n_reconst_list, error_test, label='Reconst Test')
plt.xlabel('Number of Principal Components')
plt.ylabel('Average reconstruction error norm')
plt.legend()
plt.grid()
```



3.6 Performance evaluation

In this section you will use a linear SVM to evaluate the discriminatory capability of the extracted features.

In [29]:
```
from sklearn import svm
from sklearn.model_selection import GridSearchCV

def SVM_accuracy_evolution(X_train_t, Y_train, X_val_t, Y_val, X_test_
    """Compute the accuracy of training, validation and test data for

    Args:
        X_train_t (numpy dnarray): training data projected in the new
        Y_train (numpy dnarray): labels of the training data (number d
        X_val_t (numpy dnarray): validation data projected in the new
        Y_val (numpy dnarray): labels of the validation data (number d
        X_test_t (numpy dnarray): test data projected in the new featu
        Y_test (numpy dnarray): labels of the test data (number data x
        rang_feat: range with different number of features to be evalu
    """

    # Define the model to train a liner SVM
    clf = svm.SVC(kernel='linear')

    acc_tr = np.empty(X_train_t.shape[1]-1)
    acc_val = np.empty(X_train_t.shape[1]-1)
    acc_test = np.empty(X_train_t.shape[1]-1)
    for i in rang_feat:
        # Train SVM classifier
        clf.fit(X_train_t[:,:i], Y_train)

        # Compute train, val and test accuracies and save in acc_tr, a
        acc_tr[i-1] =  clf.score(X_train_t[:,:i], Y_train)
        acc_val[i-1] = clf.score(X_val_t[:,:i], Y_val)
        acc_test[i-1] = clf.score(X_test_t[:,:i], Y_test)

    return acc_tr, acc_val,acc_test

# Run the function with the pca extracted features  and plot the resul
rang_feat = range(1,P_train.shape[1])
[acc_tr, acc_val, acc_test] = SVM_accuracy_evolution(P_train,
                                                     Y_train,
                                                     P_val,
                                                     Y_val,
                                                     P_test,
                                                     Y_test, rang_feat


plt.figure()
plt.plot(range(1,P_train.shape[1]),acc_tr, "b", label="train")
plt.plot(range(1,P_train.shape[1]),acc_val, "g", label="validation")
plt.plot(range(1,P_train.shape[1]),acc_test, "r", label="test")
plt.xlabel("Number of features")
plt.ylabel("Accuracy")
plt.title('Accuracy evolution')
plt.legend(['Training', 'Validation', 'Test'], loc = 4)
plt.grid()
plt.show()
```



3.7 Select the optimum number of features

We can use the validation accuracy vector to obtain optimum number of features and provide the test error for this number of features.

In [30]:
```
K_opt = np.argmax(acc_val)
print('Number optimum of features: %d' %(K_opt))
print("The optimum test accuracy is  %2.2f%%" %(acc_test[K_opt]*100))
```

```
Number optimum of features: 37
The optimum test accuracy is  87.00%
```

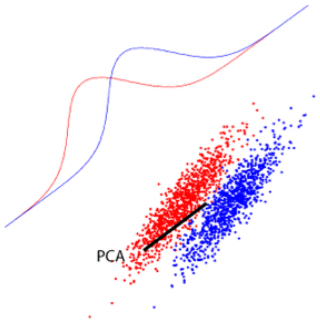
3.8. Limitations in PCA formulation

Remember PCA is an **unsupervised** algorithm!!!!

Consider the example of the figure where we have a binary classification problem.

- Which direction will PCA consider as the most relevant one?
- If we had to extract only one projection, which is the most relevant for the task?

- Clearly, when dealing with supervised problems, we should consider the labels to obtain good features

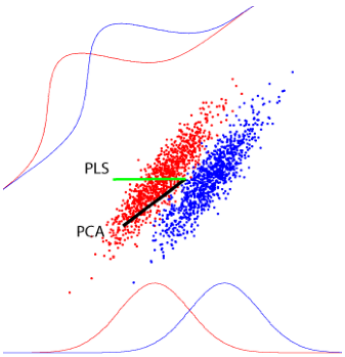


Solution: we can find in the literature supervised feature extraction approaches, such as, Partial Least Squares (PLS), Orthogonal Partial Least Squares (OPLS) and Canonical Correlation Analysis (CCA). These methods try to maximize either the covariance or correlation between the projected data and the labels, in this way, try to find the proyections that mostly align with the targets.

4. Supervised MVA

4.1 Partial Least Squares (PLS)

Goal: Find the projections of the input and output data with maximum covariance (Maximizing the variance between input and target (output) Value):



$$\begin{aligned}\mathbf{U},\mathbf{V} = & \underset{\mathbf{U},\mathbf{V}}{\operatorname{argmax}} \operatorname{Tr}\left\{\mathbf{U}^T\mathbf{X}^T\mathbf{Y}\mathbf{V}\right\} \\ = & \underset{\mathbf{U},\mathbf{V}}{\operatorname{argmax}} \operatorname{Tr}\left\{\mathbf{U}^T\mathbf{C}_{\mathbf{XY}}\mathbf{V}\right\} \\ & \text{s.t. } \mathbf{U}^T\mathbf{U} = \mathbf{V}^T\mathbf{V} = \mathbf{I}\end{aligned}$$

Two matricies, input space and output space Which leads to

$$\mathbf{U},\mathbf{V} = \operatorname{svd}(\mathbf{C}_{\mathbf{XY}})$$

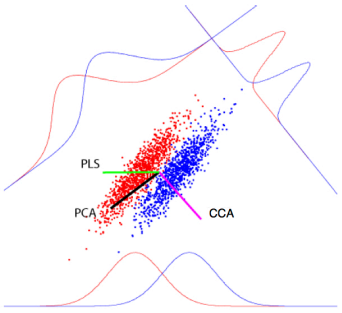
The maximum number of projections is limited by the number of output classes minus one.

4.2 Canonical Correlation Analysis (CCA)

Goal: Find the directions of maximum correlation between input and output data

$$\mathbf{u},\mathbf{v} = \underset{\mathbf{u},\mathbf{v}}{\operatorname{argmax}} \frac{\left(\mathbf{u}^T\mathbf{C}_{\mathbf{XY}}\mathbf{v}\right)^2}{\mathbf{u}^T\mathbf{C}_{\mathbf{XX}}\mathbf{u}\mathbf{v}^T\mathbf{C}_{\mathbf{YY}}\mathbf{v}}$$

$$\mathbf{U}, \mathbf{V} = \underset{\mathbf{U},\mathbf{V}}{\operatorname{argmax}} \operatorname{Tr}\left\{\mathbf{U}^T \mathbf{C_{XY}} \mathbf{V}\right\}$$
$$\text{s.t.} \quad \mathbf{U}^T \mathbf{C_{XX}} \mathbf{U} = \mathbf{V}^T \mathbf{C_{YY}} \mathbf{V} = \mathbf{I}$$



SOLUTION This problem can be solved as a generalized eigenvalue problem. If we follow a similar procedure to PCA, we can start considering the projection onto a one-dimensional space ($K = 1$):

$$\mathbf{u}_1, \mathbf{v}_1 = \underset{\mathbf{u}_1,\mathbf{v}_1}{\operatorname{argmax}} \mathbf{u}_1^\top \mathbf{C_{XY}} \mathbf{v}_1$$
$$\text{s.t.} \;\; \mathbf{u}_1^\top \mathbf{C_{XX}} \mathbf{u}_1 = \mathbf{v}_1^\top \mathbf{C_{YY}} \mathbf{v}_1 = 1$$

Applying Lagrange multipliers:

$$\mathbf{u}_1, \mathbf{v}_1 = \underset{\mathbf{u}_1,\mathbf{v}_1}{\operatorname{argmax}} \mathbf{u}_1^\top \mathbf{C_{XY}} \mathbf{v}_1 + \lambda_1 \left(1 - \mathbf{u}_1^\top \mathbf{C_{XX}} \mathbf{u_1}\right) + \gamma_1 \left(1 - \mathbf{v}_1^\top \mathbf{C_{YY}} \mathbf{v_1}\right)$$

whose solution is given by the following system of generalized eigen value problems:

$$\mathbf{C_{XY}} \mathbf{v}_1 = \lambda_1 \mathbf{C_{XX}} \mathbf{u_1}$$
$$\mathbf{C_{XY}}^\top \mathbf{u}_1 = \gamma_1 \mathbf{C_{YY}} \mathbf{v_1}$$

which can be converted to stardard eigenvalue problems by multiplying each equation, respectively, by $\mathbf{C_{XX}^{-1}}$ and $\mathbf{C_{YY}^{-1}}$.

Somme comments:

- As many extracted features as output classes minus one.
- It is usually applied to obtain a common space to work with input and output features
- For classification purposes:
  - It tends to outperform PLS approaches
  - It's equivalent to LDA as feature extractor

4.3 Linear Discriminat Analysis (LDA) as feature extractor

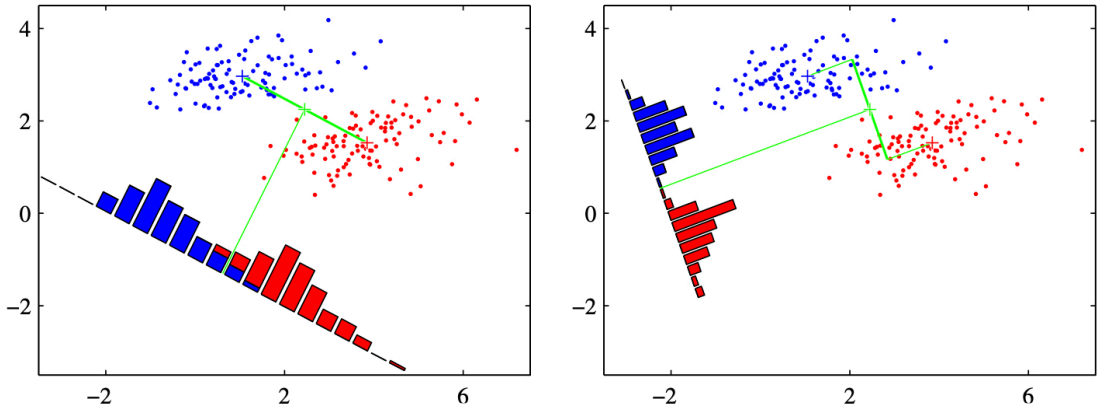
To finish reviewing basic methods for feature extraction, there is a very popular method closely related to logistic regression: Fisher's linear discriminant function analysis (LDA). This method is actually a classifier, but it can be used for feature extraction.

The method consists of projecting each piece of data onto the real line by means of a linear operation

$$z = \mathbf{w}^T \mathbf{x}$$

and determine the class according to z is greater or less than a certain threshold w_0, which is calculated optimally assuming that the data in each class is Gaussian distributed.

The criterion for determining $\mathbf{w}$ is to maximise the separability of the data of both classes once projected. The following is a good illustration of this. This figure is taken from C. Bishop's book Pattern Recognition and Machine Learning, Chapter 4.1.4.



As explained in the above chapter, under certain assumptions, the vector $\mathbf{w}$ that maximises the separability between classes is calculated as:

$$\mathbf{w}_{LDA} = \mathbf{\Sigma}^{-1}(\mathbf{m}_1 - \mathbf{m}_0)$$

where

$$\mathbf{m}_1 = \frac{1}{N_1} \sum_{n \in \mathcal{C}_1} \mathbf{x}_n$$

$$\mathbf{m}_0 = \frac{1}{N_1} \sum_{n \in \mathcal{C}_0} \mathbf{x}_n$$

is the mean vector of the points $\mathbf{x}$ belonging to each class, and

$$\mathbf{\Sigma} = \sum_{n \in \mathcal{C}_1} (\mathbf{x}_n - \mathbf{m}_1)(\mathbf{x}_n - \mathbf{m}_1)^T + \sum_{n \in \mathcal{C}_0} (\mathbf{x}_n - \mathbf{m}_0)(\mathbf{x}_n - \mathbf{m}_0)^T$$

This method is provided in Sklearn with the LinearDiscriminantAnalysis library,

which allows implementing both the binary and multi-class classification cases. LDA naturally extends to the latter case by finding a Gaussian for each class and separating the means by maximum Mahalanobis distance.

5. Performance evaluation of supervised feature extraction: PLS, CCA and LDA

In this section, we are going to analyze the advantages of using supervised the above feature extraction techniques.

5.1. Evaluation of Partial Least Squares (PLS)

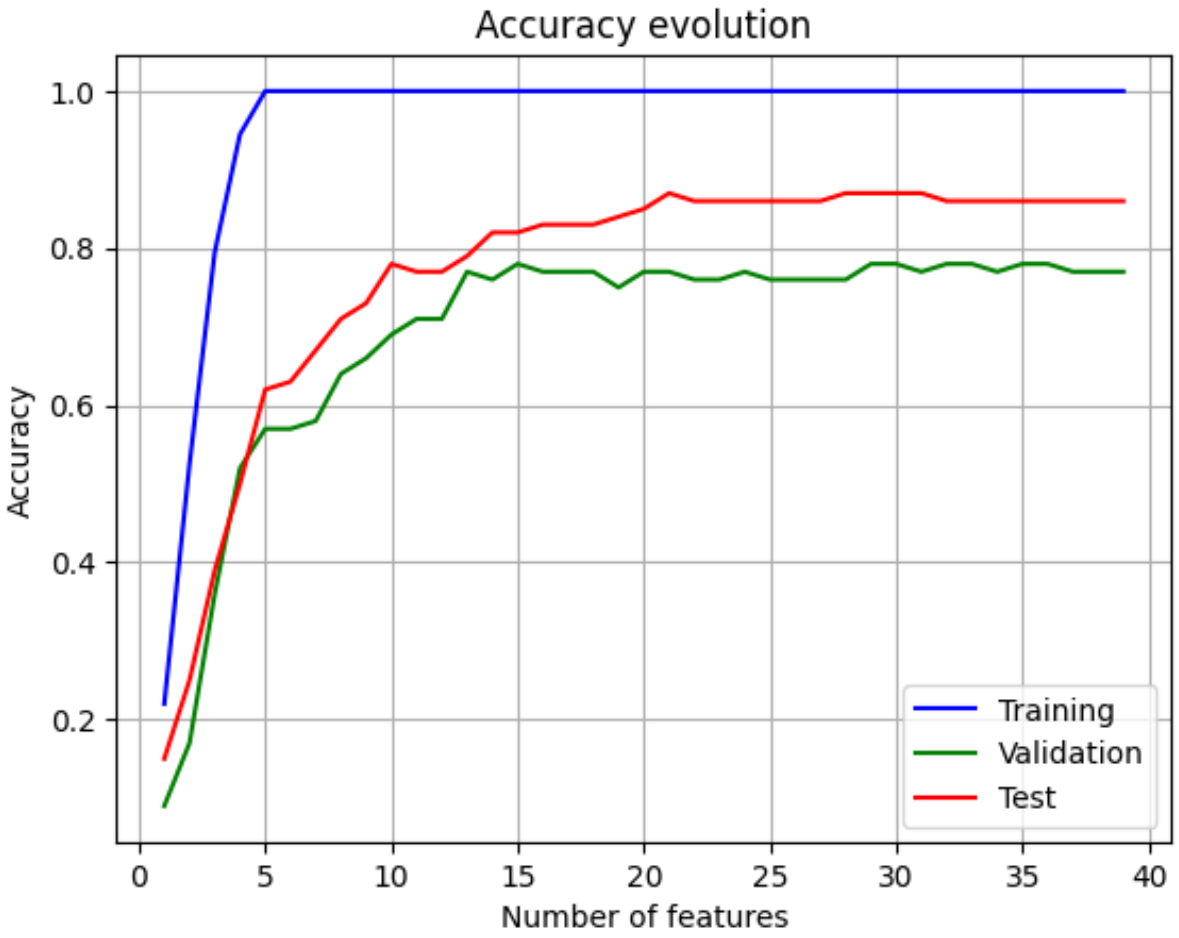
Let's start computing the PLS approach with the method `PLSSVD( )` and then we will:

1. Obtain the first eigenvectors from the training data and obtain the projections of training, validation and test data. Note that in this case we can only obtain as many new projections as number of categories.
2. Compute the SVM accuracy provided by different number of extracted features and plot it. We will use the `SVM_accuracy_evolution( )` and `plot_accuracy_evolution( )` functions of the previous section.
3. Obtain the optimum number of projected features and its corresponding test accuracy.

Note: to work with the supervised feature extraction methods, we have to use the binarized label vector (`Y_train_bin`); whereas, to train the linear SVM we can go on using the standard label codification (`Y_train` , `Y_val` and `Y_test`)

In [31]:
```
from sklearn.cross_decomposition import PLSSVD

N_feat_max = n_classes # As many new features as classes
# 1. Obtain PLS projections
pls = PLSSVD(n_components=N_feat_max)
pls.fit(X_train, Y_train_bin)
P_train_pls = pls.transform(X_train)
P_val_pls = pls.transform(X_val)
P_test_pls = pls.transform(X_test)


# 2. Compute and plot accuracy evolution
rang_feat = np.arange(1, N_feat_max, 1)
[acc_tr, acc_val, acc_test] = SVM_accuracy_evolution(P_train_pls, Y_tr
```

```
plt.figure()
plt.plot(range(1, P_train_pls.shape[1]),acc_tr, "b", label="train")
plt.plot(range(1, P_train_pls.shape[1]),acc_val, "g", label="validatio
plt.plot(range(1, P_train_pls.shape[1]),acc_test, "r", label="test")
plt.xlabel("Number of features")
plt.ylabel("Accuracy")
plt.title('Accuracy evolution')
plt.legend(['Training', 'Validation', 'Test'], loc = 4)
plt.grid()
plt.show()

# 3. Find the optimum number of features
pos_max = np.argmax(acc_val)
num_opt_feat = rang_feat[pos_max]
test_acc_opt = acc_test[pos_max]

print('Number optimum of features: %d' %(num_opt_feat))
print("The optimum test accuracy is  %2.2f%%" %(100*test_acc_opt))
```



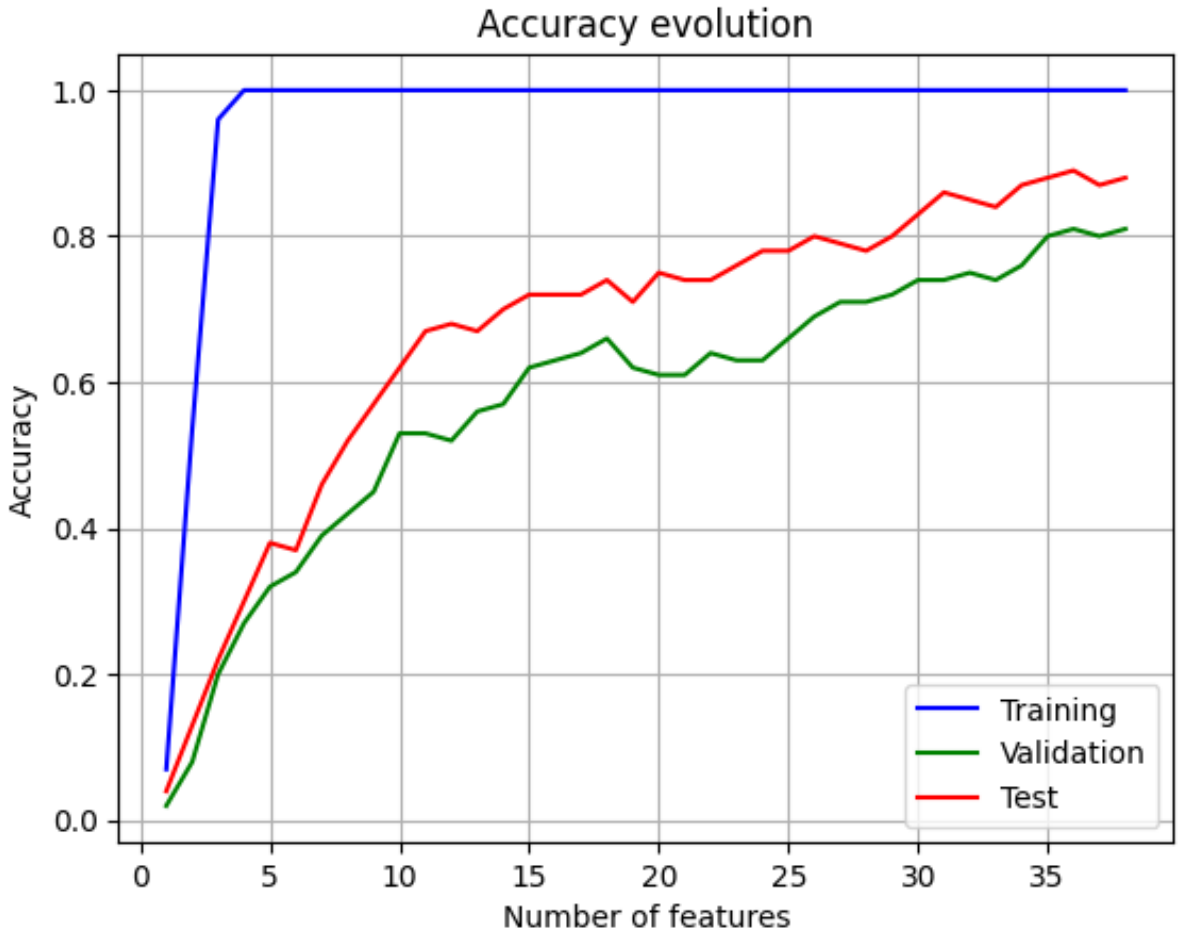

```
Number optimum of features: 15
The optimum test accuracy is  82.00%
```

5.2. Evaluation of Canonical Correlation Analysis (CCA)

In the next cell, let's repeat the previous steps to obtain the accuracy evolution with the number of features, but using the CCA algorithm.

To implement it, one are going to use the method CCA().

In [32]:
```
from sklearn.cross_decomposition import CCA

N_feat_max = n_classes-1 # As many new features as classes minus 1
# 1. Obtain CCA projections
cca = CCA(n_components=N_feat_max)
cca.fit(X_train, Y_train_bin)
P_train_cca = cca.transform(X_train)
P_val_cca = cca.transform(X_val)
P_test_cca = cca.transform(X_test)


# 2. Compute and plot accuracy evolution
rang_feat = np.arange(1, N_feat_max, 1)
[acc_tr, acc_val, acc_test] = SVM_accuracy_evolution(P_train_cca, Y_tr


plt.figure()
plt.plot(range(1,P_train_cca.shape[1]),acc_tr, "b", label="train")
plt.plot(range(1,P_train_cca.shape[1]),acc_val, "g", label="validation
plt.plot(range(1,P_train_cca.shape[1]),acc_test, "r", label="test")
plt.xlabel("Number of features")
plt.ylabel("Accuracy")
plt.title('Accuracy evolution')
plt.legend(['Training', 'Validation', 'Test'], loc = 4)
plt.grid()
plt.show()
# 3. Find the optimum number of features
pos_max = np.argmax(acc_val)
num_opt_feat = rang_feat[pos_max]
test_acc_opt = acc_test[pos_max]

print('Number optimum of features: %d' %(num_opt_feat))
print("The optimum test accuracy is  %2.2f%%" %(100*test_acc_opt))
```



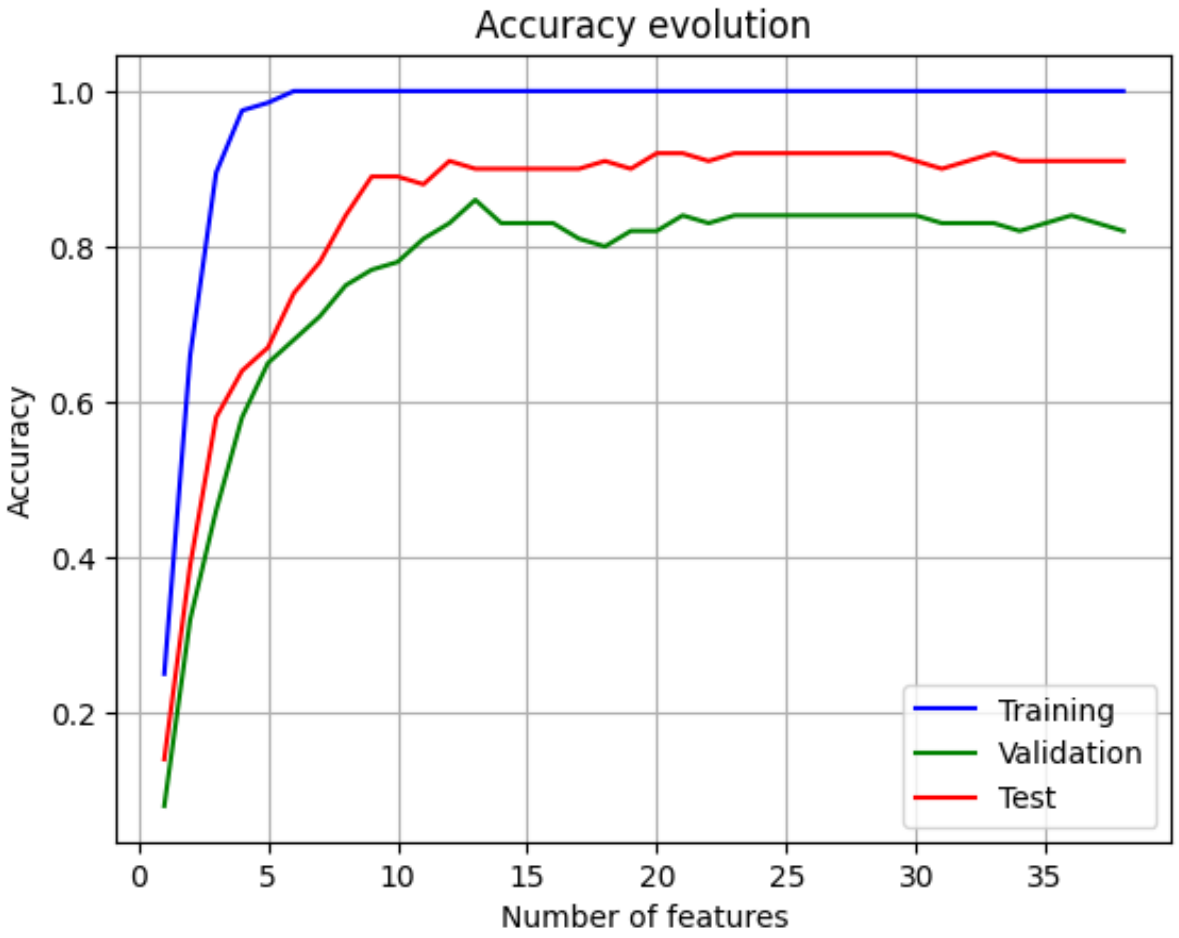

```
Number optimum of features: 36
The optimum test accuracy is  89.00%
```

5.3 Evaluation of Linear Discrimination Analysis (LDA)

Now, let's repeat the performance analysis que the LDA() method. Remember that, in classification problems, CCA is equivalent to LDA, so you should obtain similar results to those of the previous section.

In [33]:
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

N_feat_max = n_classes-1 # As many new features as classes minus 1
# 1. Obtain LDA or CCA projections
lda = LinearDiscriminantAnalysis(n_components=N_feat_max)
lda.fit(X_train, Y_train)
P_train_lda = lda.transform(X_train)
P_val_lda = lda.transform(X_val)
P_test_lda = lda.transform(X_test)

# 2. Compute and plot accuracy evolution
rang_feat = np.arange(1, N_feat_max, 1)
[acc_tr, acc_val, acc_test] = SVM_accuracy_evolution(P_train_lda, Y_tr
```

```
plt.figure()
plt.plot(range(1,P_train_lda.shape[1]),acc_tr, "b", label="train")
plt.plot(range(1,P_train_lda.shape[1]),acc_val, "g", label="validation
plt.plot(range(1,P_train_lda.shape[1]),acc_test, "r", label="test")
plt.xlabel("Number of features")
plt.ylabel("Accuracy")
plt.title('Accuracy evolution')
plt.legend(['Training', 'Validation', 'Test'], loc = 4)
plt.grid()
plt.show()
# 3. Find the optimum number of features
pos_max = np.argmax(acc_val)
num_opt_feat = rang_feat[pos_max]
test_acc_opt = acc_test[pos_max]

print('Number optimum of features: %d' %(num_opt_feat))
print("The optimum test accuracy is  %2.2f%%" %(100*test_acc_opt))
```




```
Number optimum of features: 13
The optimum test accuracy is  90.00%
```

Some useful references

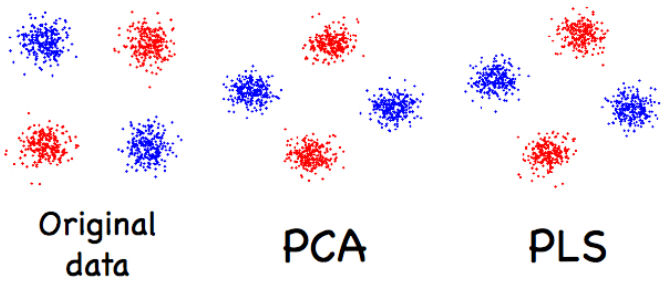
If you want to go deeper into these methods, these papers can be useful for you:

- Wold S., Albano C., Dunn W.J., Edlund U., Esbensen K., Geladi P., Hellberg S., Johansson E., Lindberg W. and Sjostrom M. (1984). Multivariate data analysis in chemistry Chemometrics, Mathematics and Statistics in Chemistry Reidel Publishing Company p. 17.
- Geladi P. (1988). Notes on the history and nature of partial least squares (PLS) modelling. Journal of Chemometrics 2, 231–246.
- Hardoon D.R., Szedmak S. and Shawe-Taylor J. (2003). Canonical correlation analysis: An overview with application to learning methods. Technical report.
- Worsley K, Poline J, Friston K and Evans. A. (1998). Characterizing the response of pet and fMRI data using multivariate linear models (MLM). Neuroimage 6, 305–319.

5. Kernel Multivariate Analysis

5.1 Motivation of KMVA

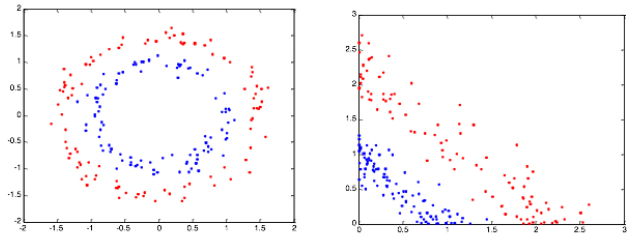
Previous MVA models are **linear** methods, so they are simple, easy to understand, robust to overfitting problems. However, it lacks expressive power.



Solution: no linear extension by means of kernel methods!!!

5.2 Kernel methods: review

- Idea: Project data onto a another space (usually with a dimensionality larger than that of the original space)



... and run a linear algorithm run in this *feature space*. The outcome of the linear

model in *feature space* turns out to be a non-linear model in the original input space.

Some useful results:

- **Kernel Trick**: It is possible to compute inner products in many ∞-dimensional space:

  $$k(\mathbf{x}, \mathbf{y}) = \langle \boldsymbol{\phi}(\mathbf{x}), \boldsymbol{\phi}(\mathbf{y}) \rangle$$

  So, if a linear algorithm can be reformulated in terms of inner products, we can replace these dot products by kernel functions.
- **Representer Theorem**: states that the solutions of certain optimization problems can be written as a linear combination of the training samples

  $$\mathbf{u} = \sum_{n=1}^{N} a^{(n)} \boldsymbol{\phi}(\mathbf{x}^{(n)}) = \boldsymbol{\Phi}\mathbf{a}$$

  where the vector $\mathbf{a} = \left[a^{(1)}, \ldots, a^{(N)}\right]^\top$ contains the dual variables which are indicating the weight that takes each data to represent the solution and $\boldsymbol{\Phi} = \left[\boldsymbol{\phi}(\mathbf{x})^{(1)}, \ldots, \boldsymbol{\phi}(\mathbf{x})^{(N)}\right]$ is a matrix with the data mapped in the feature space, one column per data instance.

5.3 Kernel Principal Component Analysis (KPCA): formulation

KPCA aims to find the projections that maximize the variance of the data in **feature space**. This means to perform the PCA algorithm with matrix $\boldsymbol{\Phi}$:

Obtain the eigen spectrum of the covariance matrix in Feature Space $\boldsymbol{\Sigma}$.

- **Eigenvalue Problem**

  $$\boldsymbol{\Sigma}\mathbf{u} = \lambda\mathbf{u} \Rightarrow \frac{1}{N}\boldsymbol{\Phi}\boldsymbol{\Phi}^\top\mathbf{u} = \lambda\mathbf{u}$$

- **Eigenvectors are linear combinations of the data samples**:

  $$\mathbf{u} = \boldsymbol{\Phi}\mathbf{a}$$

  $$\frac{1}{N}\boldsymbol{\Phi}\boldsymbol{\Phi}^\top\boldsymbol{\Phi}\mathbf{a} = \lambda\boldsymbol{\Phi}\mathbf{a}$$

- Multiply the above equation by $\boldsymbol{\Phi}^\top$

$$\frac{1}{N}\mathbf{\Phi}^\top\mathbf{\Phi}\mathbf{\Phi}^\top\mathbf{\Phi}\mathbf{a} = \lambda\mathbf{\Phi}^\top\mathbf{\Phi}\mathbf{a}$$

- Remember the kernel matrix can be written as $K = \mathbf{\Phi}^\top\mathbf{\Phi}$

$$\frac{1}{N}\mathbf{K^2a} = \lambda\mathbf{Ka}$$

- Multiplication by $\mathbf{K^{-1}}$

$$\frac{1}{N}\mathbf{Ka} = \lambda\mathbf{a}$$

The previous result states that if $\mathbf{u}$ is an eigenvector of the covariance matrix in feature space, $\mathbf{\Sigma}$, then $\mathbf{a}$, the vector with the dual coefficients of $\mathbf{u}$ in terms of the data samples, is an eigenvector of $\mathbf{K}$.

Moreover, if λ is the eigenvalue of associated to $\mathbf{u}$, then $N\lambda$ is the eigenvalue that corresponds to $\mathbf{a}$.

This leads to the following **algorithm to compute the KPCA**

- **Input:** $\mathbf{K}$, kernel matrix
- **Process:**
  1. **Center** the kernel $\mathbf{K} \leftarrow \mathbf{K} - \frac{\mathbf{1}}{\mathbf{N}}\mathbf{K11}^\top - \frac{\mathbf{1}}{\mathbf{N}}\mathbf{11}^\top\mathbf{K} + \frac{\mathbf{1}}{\mathbf{N^2}}\mathbf{11}^\top\mathbf{K11}^\top$
  2. $\mathbf{A, \Lambda}$ = eig($\mathbf{K}$)
  3. $\mathbf{Z} = \mathbf{KA_{:K}}$
- **Output:** $\mathbf{Z}$, $N \times K$ matrix with the projection of all the samples on the K principal components in feature space

Z_{ik} is the projection of $\boldsymbol{\phi}(\mathbf{x}_i)$ on the k-th principal component.

$$Z_{ik} = \boldsymbol{\phi}(\mathbf{x}_i)^\top\mathbf{\Phi}\mathbf{a}_k = (\mathbf{\Phi}^\top)_i\mathbf{\Phi}\mathbf{a}_k = \mathbf{K_i a_k}$$

REFERENCE

Scholkopf B., Smola A. and Muller K.R. (1998). Non linear component analysis as kernel eigenvalue problem. Neural Computation 10, 1299–1319.

KPCA provides with more dimensions

- In the discussion of PCA we showed that when data lies in a subspace of

smaller rank than the number of data feature, **PCA is able to optimally reduce the number of dimensions** in the projected data and recover that subspace

- With KPCA the intuition of the resulting number of dimensions of the projected data flows in the opposite direction. Remember the Kernel Principal Components are the eigenvectors of the Kernel Matrix. Therefore the **upper limit of the number of Kernel Principal Components is the number of data,** N.
- Intuitively, KPCA can give as many Principal Components as training samples, providing the used kernel yields a full rank kernel matrix.
- For instance, in the case of an RBF kernel with a spread parameter γ large enough (narrow Gaussians), the kernel matrix will have full rank. The kernel matrix can be regarded as a new data matrix in which the coordinates of each point are the evaluations of the kernel function centred on each training sample.

5.4 Kernel Partial Least Square (KPLS)

Review linear formulation

$$\mathbf{U}, \mathbf{V} = \underset{\mathbf{U},\mathbf{V}}{\operatorname{argmax}} \operatorname{Tr}\left\{\mathbf{U}^T\mathbf{X}^T\mathbf{Y}\mathbf{V}\right\} \qquad \text{s.t. } \mathbf{U}^T\mathbf{U} = \mathbf{V}^T\mathbf{V} = \mathbf{I}$$

Applying a similar procedure to KPCA, we get the following **kernel formulation**

$$\mathbf{A}, \mathbf{V} = \underset{\mathbf{A},\mathbf{V}}{\operatorname{argmax}} \operatorname{Tr}\left\{\mathbf{A}^T\mathbf{K}\mathbf{Y}\mathbf{V}\right\} \qquad \text{s.t. } \mathbf{A}^T\mathbf{K}\mathbf{A} = \mathbf{V}^T\mathbf{V} = \mathbf{I}$$

which solution is given by

$$\mathbf{A}, \mathbf{V} = \operatorname{svd}(\mathbf{K}\mathbf{Y})$$

5.5 Kernel Canonical Correlation Analysis (KCCA)

Review linear formulation

$$\begin{aligned}\mathbf{U}, \mathbf{V} = \quad & \underset{\mathbf{U},\mathbf{V}}{\operatorname{argmax}} \operatorname{Tr}\left\{\mathbf{U}^T\mathbf{X}^T\mathbf{Y}\mathbf{V}\right\} \\ & \text{s.t. } \mathbf{U}^T\mathbf{X}^T\mathbf{X}\mathbf{U} = \mathbf{V}^T\mathbf{Y}^T\mathbf{Y}\mathbf{V} = \mathbf{I}\end{aligned}$$

Then, we can obtain the following **kernel formulation**

$$\begin{aligned}\mathbf{A}, \mathbf{V} = & \underset{\mathbf{A},\mathbf{V}}{\operatorname{argmax}} \operatorname{Tr}\left\{\mathbf{A}^T\mathbf{K}\mathbf{Y}\mathbf{V}\right\} \\ & \text{s.t. } \mathbf{A}^T\mathbf{K}\mathbf{K}\mathbf{A} = \mathbf{V}^T\mathbf{Y}^T\mathbf{Y}\mathbf{V} = \mathbf{I}\end{aligned}$$

6. Playing with KPCA

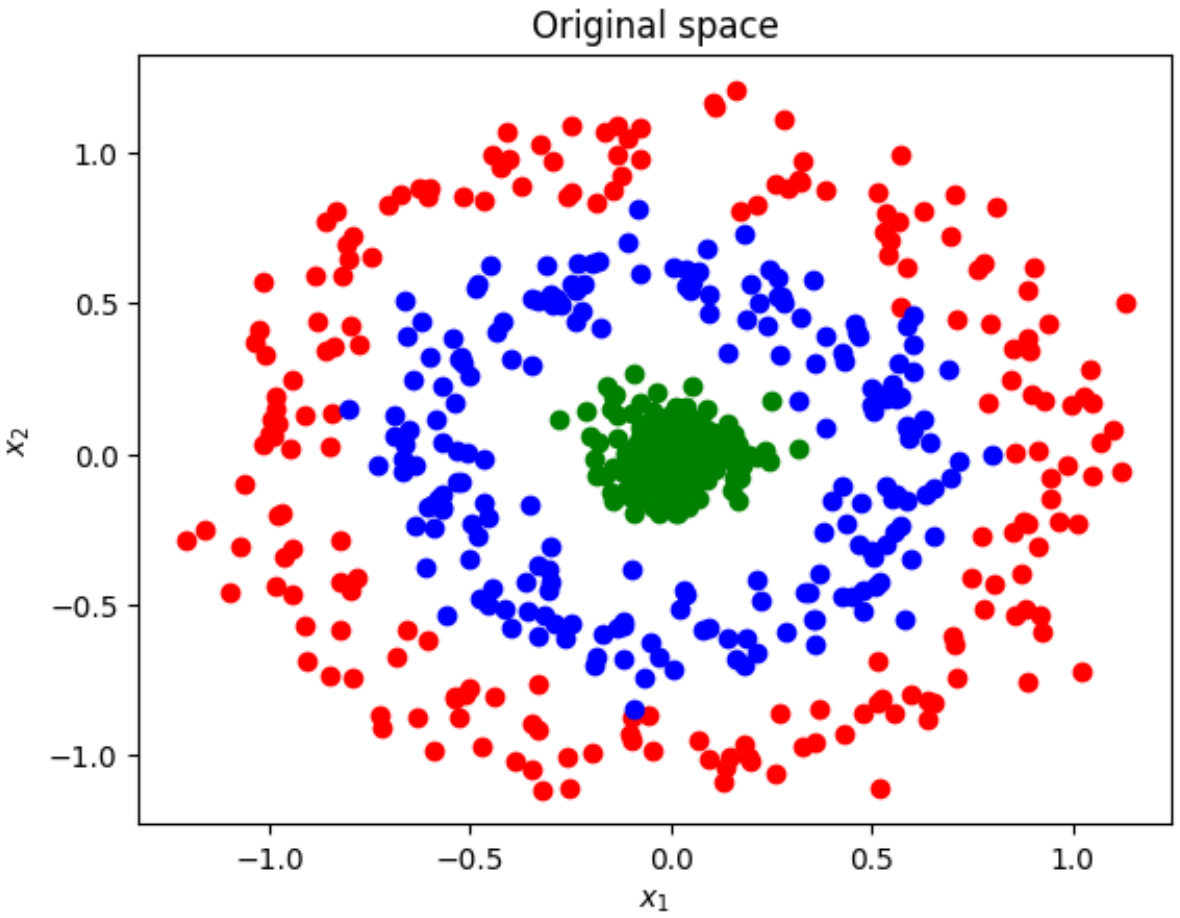
6.1 Creating a toy problem

The following code let you generate a bidimensional problem consisting of thee circles of data with different radius, each one associated to a different class.

As expected from the geometry of the problem, the classification boundary is not linear, so we will able to analyze the advantages of using no linear feature extraction techniques to transform the input space to a new space where a linear classifier can provide an accurate solution.

In [34]:
```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import label_binarize
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_circles
import matplotlib.pyplot as plt


np.random.seed(0)
X, Y = make_circles(n_samples=400, factor=.6, noise=.1)

X_c2 = 0.1*np.random.randn(200,2)
Y_c2 = 2*np.ones((200,))

X= np.vstack([X,X_c2])
Y= np.hstack([Y,Y_c2])

plt.figure()
plt.title("Original space")
reds = Y == 0
blues = Y == 1
green = Y == 2

plt.plot(X[reds, 0], X[reds, 1], "ro")
plt.plot(X[blues, 0], X[blues, 1], "bo")
plt.plot(X[green, 0], X[green, 1], "go")
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
```

```
plt.show()

# split into a training and testing set
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.

# Normalizing the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Binarize the labels for supervised feature extraction methods
set_classes = np.unique(Y)
Y_train_bin = label_binarize(Y_train, classes=set_classes)
Y_test_bin = label_binarize(Y_test, classes=set_classes)
```



6.2 KPCA in sklearn

To extend the previous PCA feature extraction approach to its non-linear version, we can use of KernelPCA( ) function.

To analyze the advantages of the non linear feature extraction, let's compare it with its linear version. So, let's start computing both linear and kernelized versions of PCA. Next code obtains the variables (P_train, P_test) and (P_train_k,

P_test_k) which contain, respectively, the projected data of the linear PCA and the KPCA.

To start to work, let's compute a maximum of two new projected features and consider a Radial Basis Function kernel (RBF) with gamma (the kernel parameter) fixed to 1.

In [35]:
```
from sklearn.decomposition import PCA, KernelPCA

N_feat_max=2
gamma_value = 1   #Analyze different gamma values

# Learn a linear PCA of two components
# Transform the training set onto the new components: X_train -> P_tra
# Transform the test set onto the new components: X_test -> P_test

pca = PCA(n_components=N_feat_max).fit(X_train)
P_train = pca.transform(X_train)
P_test = pca.transform(X_test)

# Learn a kernel PCA of two components with an RBF kernel and spread p
# Transform the training set onto the new components: X_train -> P_tra
# Transform the test set onto the new components: X_test -> P_test_k

kpca = KernelPCA(n_components=N_feat_max, kernel='rbf', gamma=gamma_va
kpca.fit(X_train)
P_train_k = kpca.transform(X_train)
P_test_k = kpca.transform(X_test)
```

Now, let's evaluate the discriminatory capability of the projected data (both linear and kernelized ones) feeding with them a linear SVM and measuring its accuracy over the test data using either the linear PCA projected data or the KPCA ones.

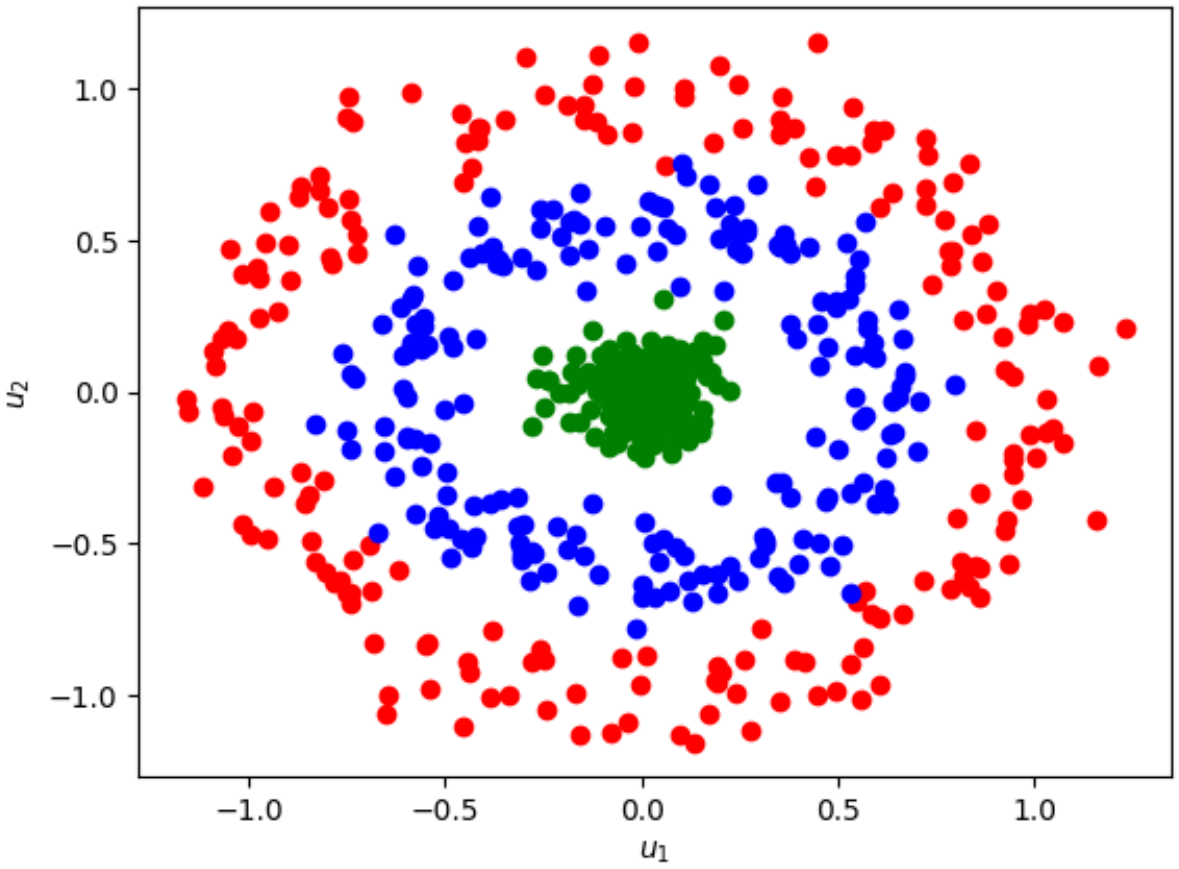
In [36]:
```
# Define SVM classifier
from sklearn import svm
clf = svm.SVC(kernel='linear')


clf.fit(P_train, Y_train)
acc_tr =  clf.score(P_train, Y_train)
acc_tst = clf.score(P_test, Y_test)

clf.fit(P_train_k, Y_train)
acc_tr_k =  clf.score(P_train_k, Y_train)
acc_tst_k = clf.score(P_test_k, Y_test)

print("Linear PCA")
print("----------")
print("Accuracy training set: {0:.2f}".format(acc_tr*100))
print("Accuracy test set: {0:.2f}".format(acc_tst*100))
```

```
print("")
print("Kernel PCA")
print("----------")
print("Accuracy training set: {0:.2f}".format(acc_tr_k*100))
print("Accuracy test set: {0:.2f}".format(acc_tst_k*100))
print("")
```

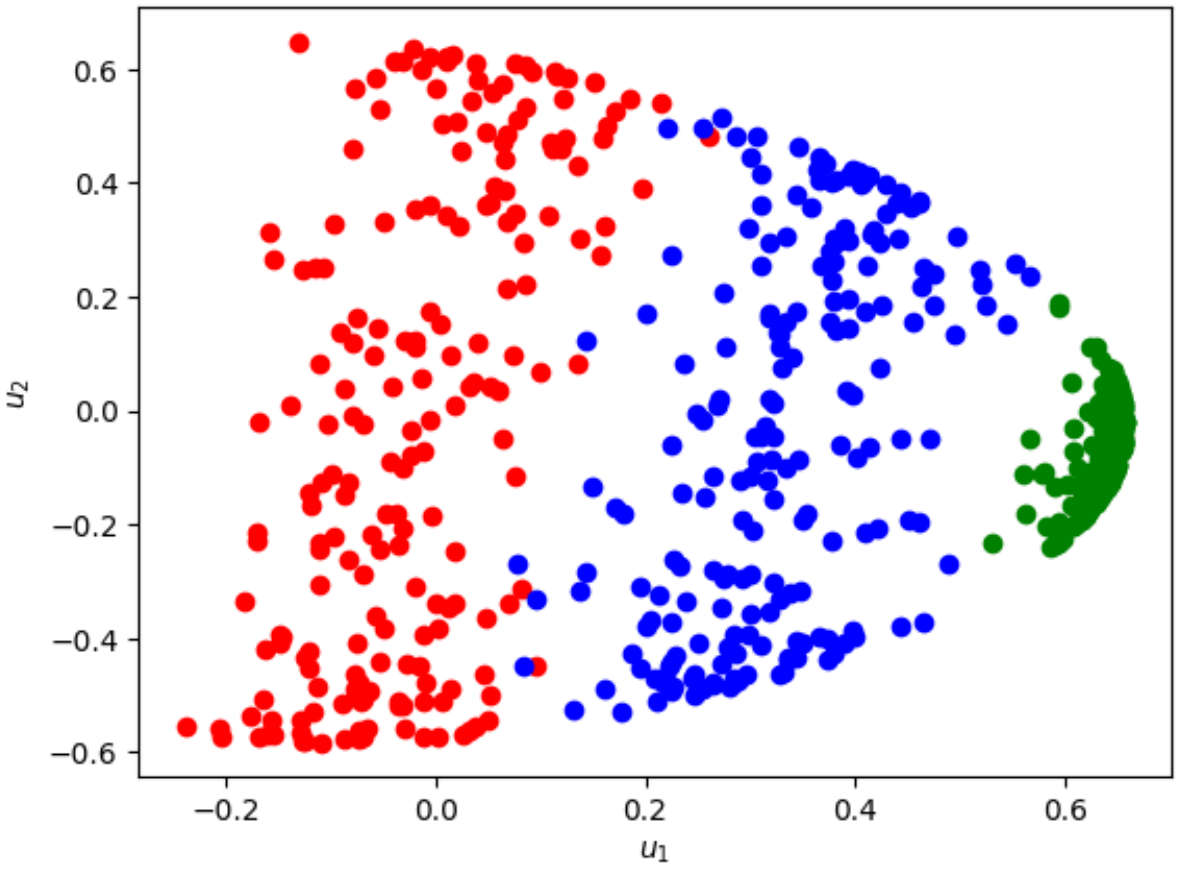
```
Linear PCA
----------
Accuracy training set: 36.44
Accuracy test set: 24.00

Kernel PCA
----------
Accuracy training set: 94.89
Accuracy test set: 95.33
```

Finally, next code plots the projected data

In [37]:
```
P_ = pca.transform(X)
Pk_=kpca.transform(X)
plt.plot(P_[reds, 0], P_[reds, 1], "ro")
plt.plot(P_[blues, 0], P_[blues, 1], "bo")
plt.plot(P_[green, 0], P_[green, 1], "go")
plt.xlabel("$u_1$")
plt.ylabel("$u_2$")
plt.show()
```



In [38]:
```
plt.plot(Pk_[reds, 0], Pk_[reds, 1], "ro")
plt.plot(Pk_[blues, 0], Pk_[blues, 1], "bo")
plt.plot(Pk_[green, 0], Pk_[green, 1], "go")
plt.xlabel("$u_1$")
plt.ylabel("$u_2$")
plt.show()
```



6.3 Implementation of Kernel MVA approaches

Until now, we have only used the KPCA approach because is the only not linear feature extraction method that it is included in Scikit-Learn.

However, if we compare linear and kernel versions of MVA approaches, we could extend any linear MVA approach to its kernelized version. In this way, we can use the same methods reviewed for the linear approaches and extend them to its non-linear fashion calling it with the training kernel matrix, instead of the training data, and the method would learn the dual variables, instead of the eigenvectors.

The following table relates both approaches:

	Linear	Kernel
Input data	$\mathbf{X}$	$\mathbf{K}$
Variables to compute (fit)	Eigenvectors ($\mathbf{U}$)	Dual variables ($\mathbf{A}$)
Projection vectors	$\mathbf{U}$	$\mathbf{U} = \mathbf{\Phi}^T\mathbf{A}$ (cannot be computed)

Project data (transform)	$\mathbf{X}' = \mathbf{U}^T\mathbf{X}^T$	$\mathbf{X}' = \mathbf{A}^T\Phi\Phi^T$ $= \mathbf{A}^T\mathbf{K}$

Computing and centering kernel matrix

Let's start this section computing the different kernel matrix that we need to train and evaluate the different feature extraction methods. For this examplo, we are going to consider a Radial Basis Function kernel (RBF), where each element of the kernel matrix is given by

$$k(x_i, x_j) = \exp(-\gamma(x_i - x_j)^2)$$

In particular, we need to compute two kernel matrix:

- Training data kernel matrix (`K_tr`) where the RBF is compute pairwise over the training data. The resulting matrix dimension is of $N_{tr} \times N_{tr}$, being N_{tr} the number of training data.
- Test data kernel matrix (`K_test`) where the RBF is compute between training and test samples, i.e., in RBF expression the data x_i belongs to test data whereas x_j belongs to training data. The resulting matrix dimension is of $N_{test} \times N_{tr}$, being N_{test} and N_{tr} the number of test and training data, respectively.

Let's use the rbf_kernel( ) function to compute the `K_tr` and `K_test` kernel matrix. In tis case, let's fix the kernel width value (`gamma`) to 1.

In [39]:
```
# Computing the kernel matrix
from sklearn.metrics.pairwise import rbf_kernel

g_value = 1

# Compute the kernel matrix (use the X_train matrix, before dividing i
K_tr = rbf_kernel(X_train, gamma = g_value)
K_test  = rbf_kernel(X_test, X_train,  gamma = g_value)
```

After compute these kernel matrix, they have to be centered (in the same way that we remove the mean when we work over the input space). For this purpose, next code provides you the function `center_K()` to remove the mean of both K_tr and K_test matrix.

In [40]:
```
def center_K(K):
    """Center a kernel matrix K, i.e., removes the data mean in the fe

    Args:
```

```
        K: kernel matrix
    """
    size_1,size_2 = K.shape;
    D1 = K.sum(axis=0)/size_1
    D2 = K.sum(axis=1)/size_2
    E = D2.sum(axis=0)/size_1

    K_n = K + np.tile(E,[size_1,size_2]) - np.tile(D1,[size_1,1]) - np
    return K_n


# Center the kernel matrix
K_tr_c = center_K(K_tr)
K_test_c = center_K(K_test)
```

Alternative KPCA formulation

Next, code lines obtain a KPCA implementaion using the linear PCA function and the kernel matrix as input data.

In [41]:
```
from sklearn.decomposition import PCA
from sklearn.decomposition import KernelPCA
from sklearn import svm

# Defining parameters
N_feat_max = 2

## PCA method (to complete)
# 1. Train PCA with the kernel matrix and project the data
pca_K2 = PCA(n_components=N_feat_max)
pca_K2.fit(K_tr_c)
P_train_k2 = pca_K2.transform(K_tr_c)
P_test_k2 = pca_K2.transform(K_test_c)

# 2. Evaluate the projection performance
clf = svm.SVC(kernel='linear')
clf.fit(P_train_k2, Y_train)
print('Test accuracy with PCA with a kenel matrix as input: %2.2f' %(c


## KPCA method (for comparison purposes)
# 1. Train KPCA and project the data
pca_K = KernelPCA(n_components=N_feat_max, kernel="rbf", gamma=1)
pca_K.fit(X_train)
P_train_k = pca_K.transform(X_train)
P_test_k = pca_K.transform(X_test)

# 2. Evaluate the projection performance
clf = svm.SVC(kernel='linear')
clf.fit(P_train_k, Y_train)
print('Test accuracy with KPCA: %2.2f' %(clf.score(P_test_k, Y_test)))
```

```
Test accuracy with PCA with a kenel matrix as input: 0.95
Test accuracy with KPCA: 0.95
```

Similar kernel implementations can be obtained for PLS and CCA approaches, but we leave this work for you in the next homework.