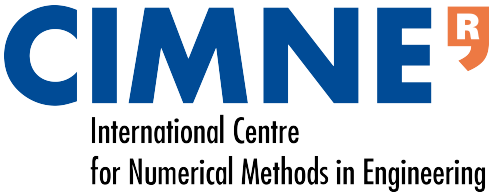
Practical Session 4

Kmeans and PCA

Alex Ferrer Ferre
Antonio Darder Bennassar

July 2022



Contents

1 Introduction

The aim of this session is to test one of the basic unsupervised algorithms (kmeans) and implement its ++ initialization variation, as well as using a singular value decomposition function to perform a PCA analysis.

A kmeans code follows the next scheme:

Algorithm 1 Kmeans++

Require:
X data matrix
k number of clusters
Ensure:
μ_{OPT} optimum parameters

1: $\mu \leftarrow plusplusInitialization(X, k)$
2: $Z \leftarrow cluster\_assig(X, \mu, k)$
3: **while** stopping criteria not met **do**
4: $\quad \mu \leftarrow centroids(X, Z, k)$
5: $\quad Z \leftarrow cluster\_assig(X, \mu, k)$
6: $\mu_{OPT} \leftarrow \mu$

The *cluster_assig* function requires the data and the current centroids to create a matrix of labels for each point. On the other hand *centroids* computes the centroids of a matrix of data "X" with its respective labels "Z".

Regarding PCA we will use a function from numpy to obtain the singular value decomposition of a matrix of data "X". Nevertheless, is important to understand what this function will return. These are the matrices u,s,v which are shown in the equation below. In this equation M and N are the number of rows and columns of X respectively, and K is the smallest between M and N. IMPORTANT notice that v is already transposed, this is how np.linalg.svd() returns this matrix.

$$u = \begin{pmatrix} u_{11} & u_{12} & \dots & u_{1K} \\ u_{21} & u_{22} & \dots & u_{2K} \\ \vdots & \vdots & \ddots & \vdots \\ u_{M1} & u_{M2} & \dots & u_{MK} \end{pmatrix} s = \begin{pmatrix} s_{11} & 0 & \dots & 0 \\ 0 & s_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & s_{KK} \end{pmatrix} v = \begin{pmatrix} v_{11} & v_{12} & \dots & v_{1N} \\ v_{21} & v_{22} & \dots & v_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ v_{K1} & v_{K2} & \dots & v_{KN} \end{pmatrix} \quad (1)$$

$$X \approx u_L \times s_L \times v_L \qquad T \approx u_L \times s_L = X_L \times v_L \quad (2)$$

1.1 The datasets

The dataset used for this session is the iris dataset. Iris is a famous collection of flower measurements (sepal length, sepal width, petal length and petal width,), and contains three families of iris flowers (iris-setosa, iris-versicolor and iris-virginica). Figure 1 shows the correlation matrix for the 4 variables. Besides the iris dataset we will also use a white and black image of a giraffe for PCA analyis (Figure 2).

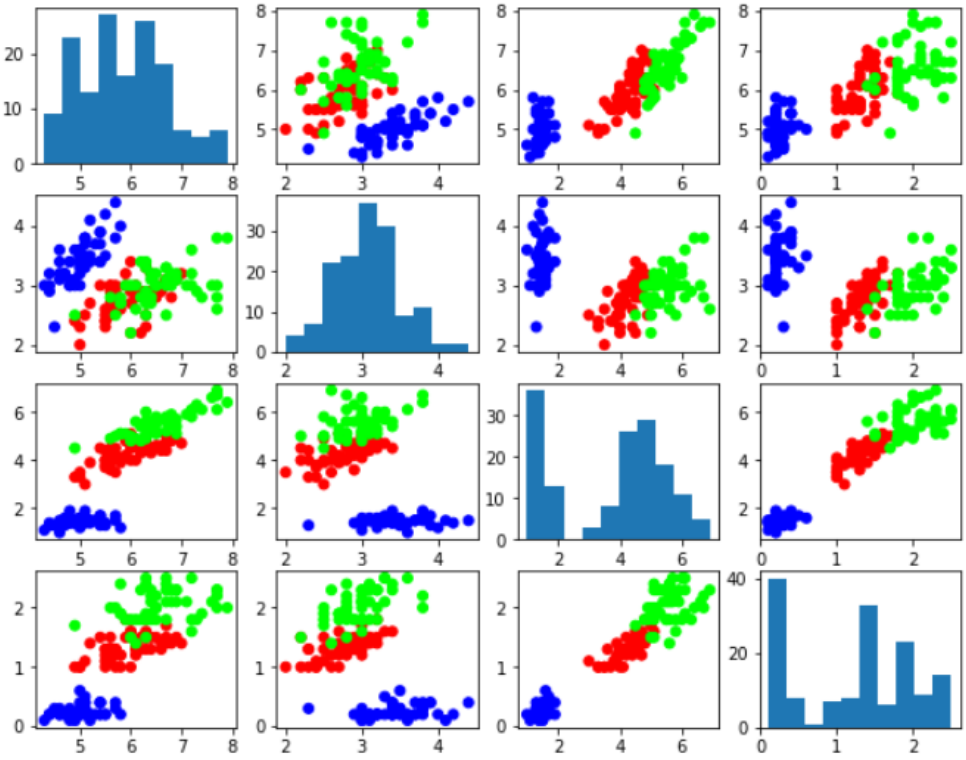


Figure 1 Iris dataset

Figure 2 Black and white giraffe

2 Code

2.1 Step 1: Import the libraries

As always import the libraries:

```
# Important libraries
import csv
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
import math
```

2.2 Step 2: Defining a class to create usable data

Like the other practical sessions, the first step after importing the libraries is to import the data with the given class Data and plot it.

```
class Data:
  def __init__(self,fileName):
    self.loadData(fileName)
    color = self.y@np.arange(1, self.y.shape[1]+1, 1, dtype=int).T
    self.color = color

  def loadData(self,fileName):
    # Uses a csv file to create a numpy array
    with open(fileName, newline='') as csvfile:
      reader = csv.reader(csvfile, delimiter=',', quotechar='|')
      data = list(reader)
    data = np.array(data)
    self.X = np.array(data[1:np.shape(data)[0],0:-1],dtype='float64')
    ydata = np.array(data[1:np.shape(data)[0],-1],dtype='int32')
    self.y = np.zeros((len(ydata),np.amax(ydata)-np.amin(ydata)+1))
    for i in range(self.y.shape[0]):
      for j in range(self.y.shape[1]):
        if ydata[i] == j+1:
            self.y[i,j] = 1

  def SplitData(self,testRatio):
    # Shuffles the data and splits data in train and test
    M      = self.X.shape[0]
```

```
    ntrain = int((1-testRatio)*M)
    idx    = np.arange(M)
    np.random.shuffle(idx)
    self.Xtrain = self.X[idx[0:ntrain],:]
    self.Xtest  = self.X[idx[ntrain:M],:]
    self.Ytrain = self.y[idx[0:ntrain],:]
    self.Ytest  = self.y[idx[ntrain:M],:]

  def plotCorrelationMatrix(self):
    # Plots the variables two by two in matrix of correlation
    nF = self.X.shape[1]
    figure, axes = plt.subplots(nrows=nF, ncols=nF)
    plt.gcf().set_size_inches(10, 8)
    for i in range(nF):
      for j in range(nF):
        if i == j:
          axes[i,j].hist(self.X[:,j])
        else:
          axes[i,j].scatter(self.X[:,j],self.X[:,i],c=self.color,cmap=
              cm.brg)
    plt.show()

  def plotData(self,i,j):
    # Plots Xi vs Xj
    plt.scatter(self.X[:,i],self.X[:,j],c=self.color,cmap=cm.brg)

def testAccuracy(Z,Y):
  cont = 0
  for i in range(Y.shape[0]):
    for j in range(Y.shape[1]):
      if Z[i,j] == np.amax(Z[i,:]):
        p = j
      if Y[i,j] == np.amax(Y[i,:]):
        t = j
    if p == t:
      cont += 1
  TA = cont/Y.shape[0]
  return TA
```

```
data = Data('P4_kmeans_iris.csv')
data.SplitData(0.2)
data.plotCorrelationMatrix()
Xtr = data.Xtrain[:,[0,2]]
Ytr = data.Ytrain
Xte = data.Xtest[:,[0,2]]
Yte = data.Ytest
```

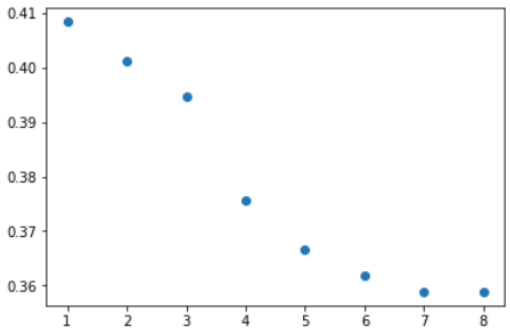
2.3 Step 3: Define the Kmeans algorithm

Next we will define the functions *cluster_assig*, *centroids*, and *plusplusInitialization* which will be the core for the function *Kmeans*.

```
# Define a function that given a matrix of data X and a vector of
    centroids mu, assigns each point of X to one cluster
def cluster_assig(X,mu,k):
  #Complete
  return Z

# Define a function that given a matrix of points X, and its labels Z
    computes the centroids of the clusters
def centroids(X,Z,k):
  #Complete
  return mu

# Define a function that given the number of clusters wanted and the
    matrix of points X, uses the kmeans++ initialization technique to
    compute the initial centroids
def plusplusInitialization(X,k):
  #Complete
  return mu

# Using the previous functions, create one that iterates and updates
    the centroids, it also returns a history of the cost function
def kmeans(X,k):
  #Complete
  return mu_new, error
```

Using the function *Kmeans* we can find the optimum centroids for the iris dataset and use these to compute the labels for the test. Run the script below to see the results.

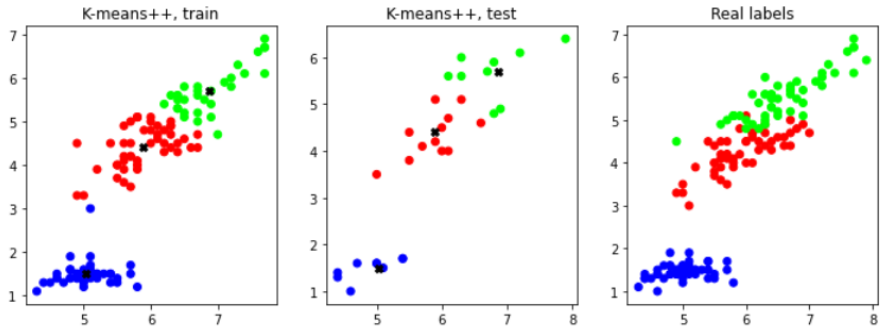
```
[76] k = 3
     mu,err = kmeans(Xtr,k)
     plt.scatter(np.arange(1,len(err)+1),err)
     plt.show()
```




```
[61] # Use this line to swap rows on mu until matching the true labels
     mu[[0,1,2],:] = mu[[2,0,1],:]
```

```
Z   = cluster_assig(Xtr,mu,k)
Zte = cluster_assig(Xte,mu,k)
color = Z@np.arange(1, Z.shape[1]+1, 1, dtype=int).T
colorte = Zte@np.arange(1, Z.shape[1]+1, 1, dtype=int).T
plt.figure(figsize=(12,4))

plt.subplot(1,3,1)
plt.title('K-means++, train')
plt.scatter(Xtr[:,0],Xtr[:,1],c=color,cmap=cm.brg)
plt.scatter(mu[:,0],mu[:,1],color="black",marker="X")

plt.subplot(1,3,2)
plt.title('K-means++, test')
plt.scatter(Xte[:,0],Xte[:,1],c=colorte,cmap=cm.brg)
plt.scatter(mu[:,0],mu[:,1],color="black",marker="X")

plt.subplot(1,3,3)
plt.title('Real labels')
data.plotData(0,2)
TA = testAccuracy(Z,Ytr)
TAe = testAccuracy(Zte,Yte)

print('Train accuracy = ',TA)
print('\nTest accuracy = ',TAe)
plt.show()
```

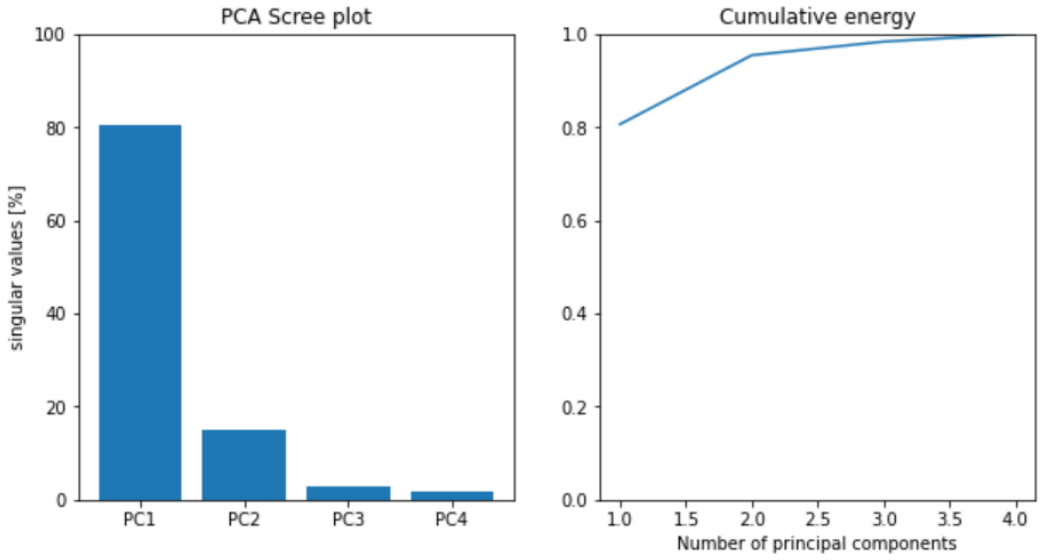
```
Train accuracy =  0.8739495798319328

Test accuracy =  0.8666666666666667
```



2.4 Step 4: Use svd to make a scree plot and a cumulative energy plot

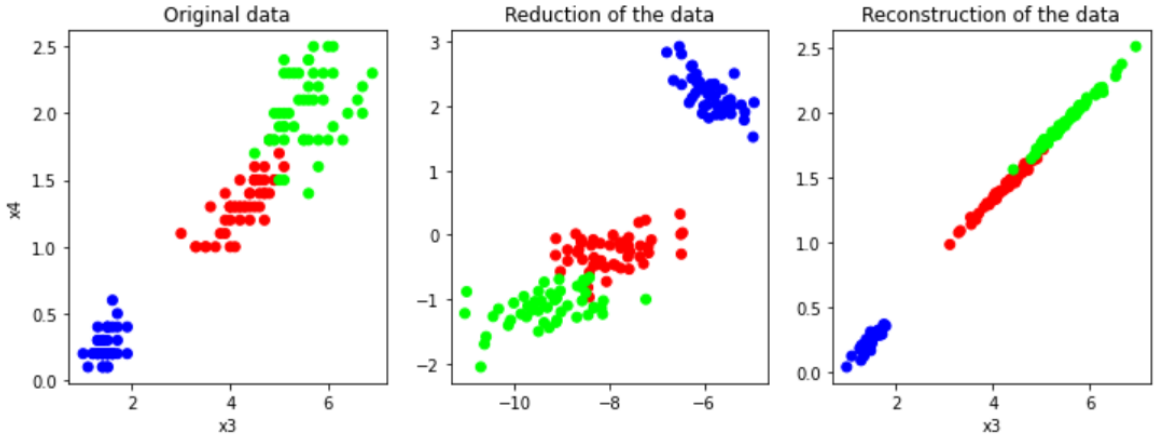
Next we will move to PCA analysis. For this task we will use $np.linalg.svd(X, full\_matrices = False)$, full matrices are not required so we can put that as False. Develop a small script applying svd to the iris data and plot the cumulative energy along with a scree plot.

These are the result you should obtain:



Using the matrices obtained u,s and v we can plot the data in the reduced dimension as well as reconstruct it after the truncation. Develop a small script which makes these plots.

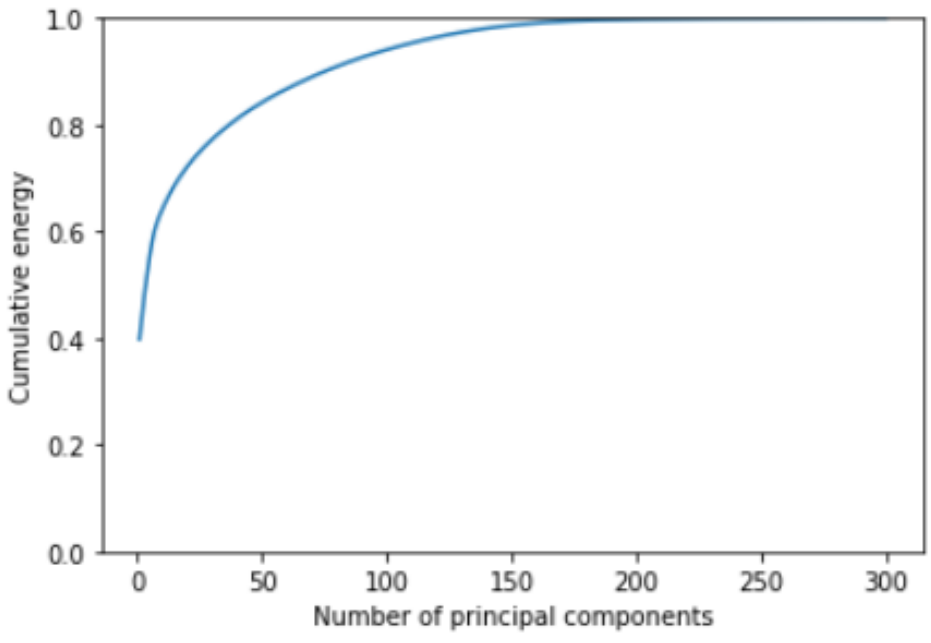
You should obtain the following:



2.5 Step 5: PCA for dimensionality reduction in images

Finally we will use PCA to "compress" images. We can use *image* from matplotlib to plot the images. The cumulative energy for the image of the giraffe can be obtained like the following:

```
from matplotlib import image

def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.2989, 0.5870, 0.1140])

img = image.imread('giraffe.jpg')
g = rgb2gray(img)
G = np.array(g)
u,s,v = np.linalg.svd(G, full_matrices=False)
rc = s/np.sum(s)
plt.plot(np.arange(1,len(rc)+1),np.cumsum(rc))
plt.xlabel("Number of principal components")
plt.ylabel("Cumulative energy")
plt.ylim((0,1))
plt.show()
```



As we can see with the first 150 components the variance of the data is really low. Now using the matrices u,s,v we can cut the data to different number of components and reconstruct it to plot the giraffe and see how it looks. Develop a small script which plots the giraffe image with different number of components.

The results should be similar to the ones below