Practical Session Report
Advanced Machine Learning:
Kernel Methods

Josselin Marnat & Valentin Benozillo

November 10, 2017

Contents

1 Kernel-PCA

Our own implementation of PCA and Kernel-PCA are respectively available in `PCA.py`, `kPCA.py`.

1.1 Implementation

1.1.1 PCA

```
def PCA(X,n_components):
    # data normalization
    X_centered = X - np.mean(X)
    X_normalized = X_centered / np.std(X)

    # covariance matrix
    cov = np.cov(X_normalized.T)

    # eigen values & vectors
    eigen_values, eigen_vectors = np.linalg.eig(cov)

    # ordering the eigen vectors by decreasing values
    eig_vals_order = np.argsort(eigen_values)[::-1]
    eigen_vectors_decr = eigen_vectors[:,eig_vals_order]

    # creating the n_components PCs over X
    PCs = np.dot(X,eigen_vectors_decr[:,0:n_components])

    return(PCs)
```

1.1.2 K-PCA : kernels definition

```
def linear(x,y,sigma):
    return np.dot(x,y)

# gaussian rbf kernel
def gaussian(x,y,sigma):
    n = np.linalg.norm(x-y)
    # return np.exp(- (n * n) / (2 * sigma * sigma))
    return np.exp(-sigma * n * n)

def poly(x,y,sigma):
    d = np.dot(x,y)
    #return np.pow(d,2)
    return (d + 1) * (d + 1)

def kernel(X,function,sigma):
    n = np.size(X,0)
    K = np.zeros([n,n])

    for i in range(n):
        for j in range(n):
            K[i,j] = function(X[i,:],X[j,:],sigma)

    # centering K
    Kn = np.ones([n,n])/n
    K = K - Kn * K - K * Kn + Kn * K * Kn

    return(K)
```

1.1.3 K-PCA

```
def k_pca(X,y,function,sigma):

    # data normalization
    X_centered = X - np.mean(X)
    X_normalized = X_centered / np.std(X)
```

```
    K = kernel(X_normalized,function,sigma)

    # covariance matrix
    cov = np.cov(K.T)

    # eigen values & vectors
    # which are NOT ALWAYS ordered descreasingly
    eigen_values, eigen_vectors = np.linalg.eig(cov)
    eig_vals_order = np.argsort(eigen_values)[::-1]
    eigen_vectors = eigen_vectors[:,eig_vals_order]

    # PCA in ONE dimension
    X_new = np.dot(K,eigen_vectors[:,0])
    plt.scatter(x=X_new[y==0],y=np.ones(len(X_new[y==0])),color='blue',alpha=.5)
    plt.scatter(x=X_new[y==1],y=np.ones(len(X_new[y==1])),color='red',alpha=.5)
    plt.show()


    # PCA in TWO dimensions
    X_new = np.dot(K,eigen_vectors[:,0:2])
    plt.scatter(x=X_new[y==0,0],y=X_new[y==0,1],color='blue',alpha=.5)
    plt.scatter(x=X_new[y==1,0],y=X_new[y==1,1],color='red',alpha=.5)
    plt.show()


    # PCA in THREE dimensions
    X_new = np.dot(K,eigen_vectors[:,0:3])
    X_new = X_new.astype('float64')
    fig = plt.figure()
    asub = fig.add_subplot(111,projection='3d')
    ax = X_new[y==0,0]
    bx = X_new[y==0,1]
    cx = X_new[y==0,2]
    ay = X_new[y==1,0]
    by = X_new[y==1,1]
    cy = X_new[y==1,2]
    asub.scatter(ax,bx,cx,c='blue',marker='o',alpha=.5)
    asub.scatter(ay,by,cy,c='red',marker='o',alpha=.5)
    plt.show()

    return K
```

1.2 Moons

1.2.1 Sample

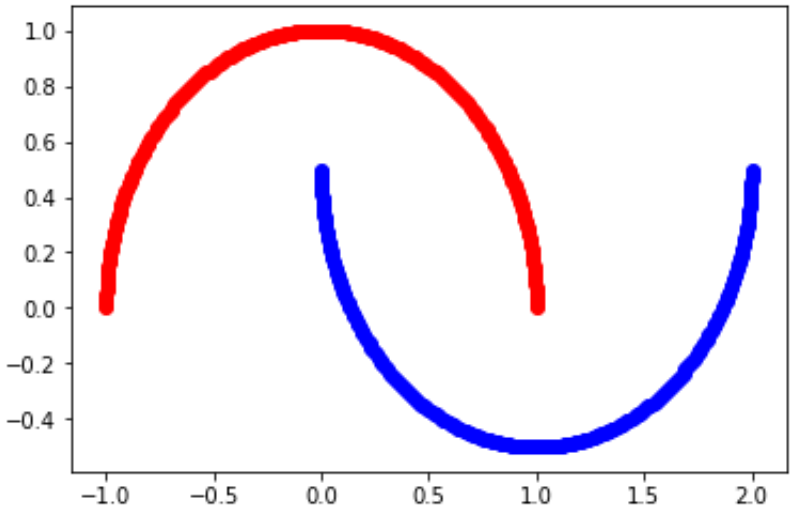

Figure 1: Moons sample

1.2.2 PCA

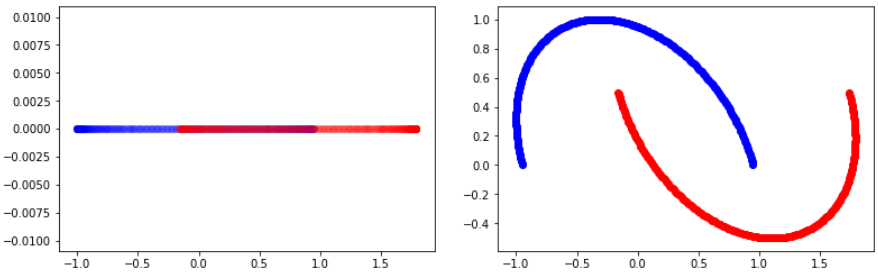

Figure 2: PCA results in one & two dimensions

```
Processing time for PCA : 0.35378100000000146 seconds
```

The classic PCA didn't work well on moons.

1.2.3 K-PCA : Linear

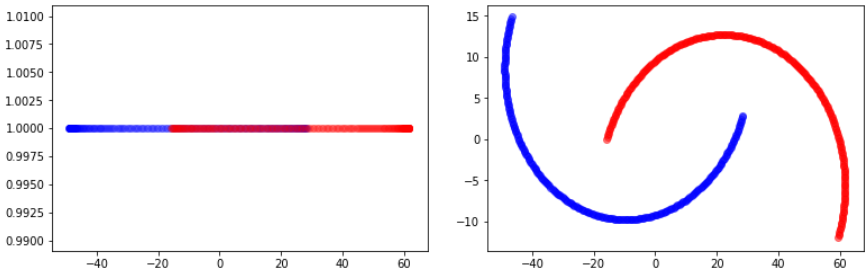

Figure 3: K-PCA results using linear kernel in one & two dimensions

```
Processing time (moons linear) :  1.903731 seconds
```

Obviously, the linear kernel didn't work too. But the processing time is considerably higher.

1.2.4 K-PCA : Gaussian

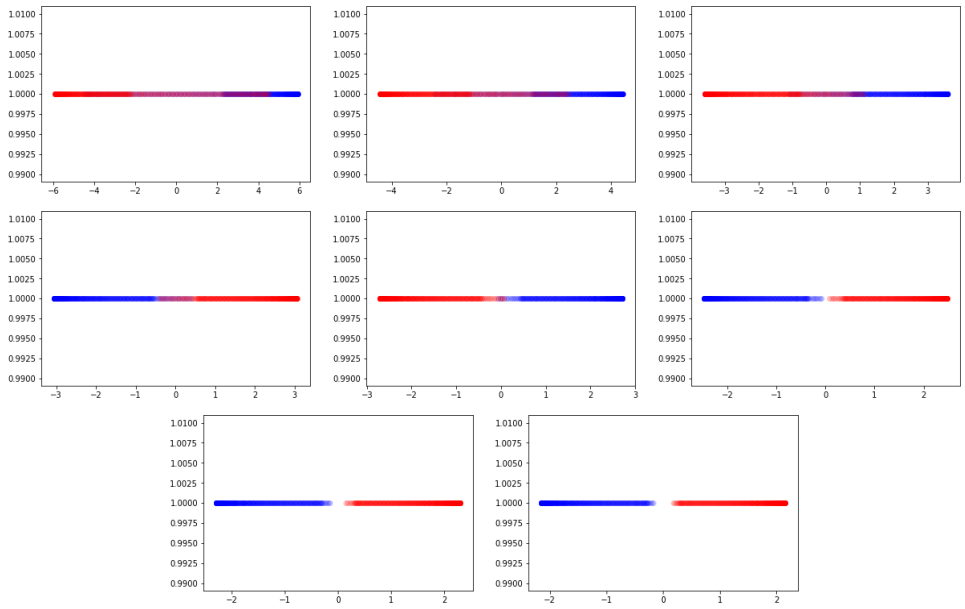

Figure 4: K-PCA results using gaussian kernel in one dimension
We start sigma at 1 and increase it by step of 1.

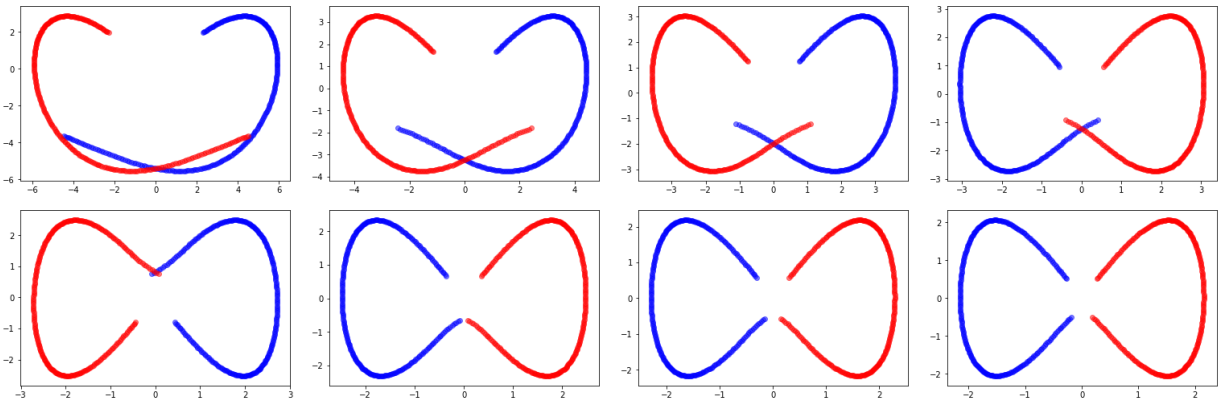

Figure 5: K-PCA results using gaussian kernel in two dimension
We start sigma at 1 and increase it by step of 1.

```
Processing time (moons gaussian 8) :  5.175716999999999 seconds
```

The gaussian work pretty well on the moons dataset. The processing time is 10 times the PCA ones.

1.2.5 K-PCA : Polynomial

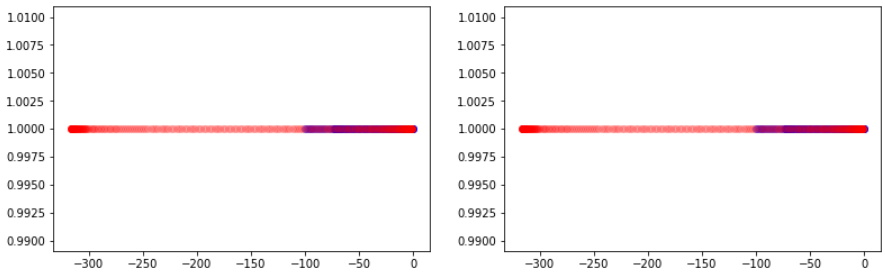

Figure 6: K-PCA results using polynomial kernel in one & two dimensions

```
Processing time (moons poly) :  2.4073649999999986 seconds
```

1.3 Circles

1.3.1 Sample

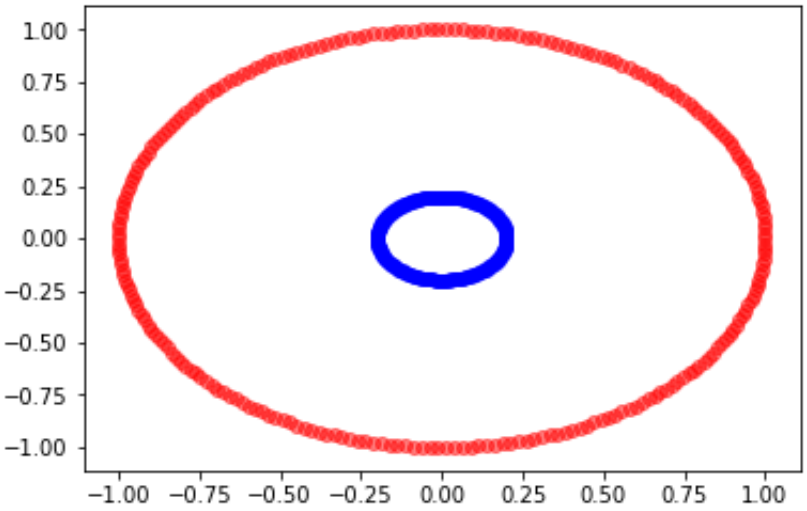

Figure 7: Circles sample

1.3.2 PCA

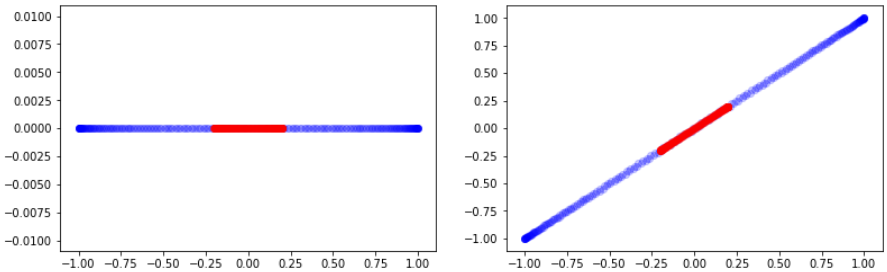

Figure 8: PCA results in one & two dimensions

```
Processing time for PCA : 0.3624499999999955 seconds
```

As expected, the classic PCA didn't work well on circles too.

1.3.3 K-PCA : Linear

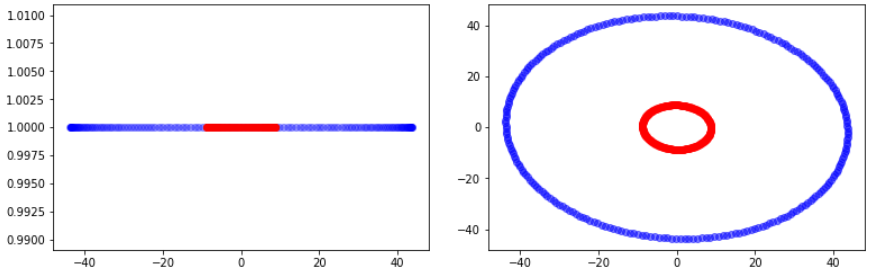

Figure 9: K-PCA results using linear kernel in one & two dimensions

```
Processing time (moons linear) :  1.9496979999999837 seconds
```

1.3.4 K-PCA : Gaussian

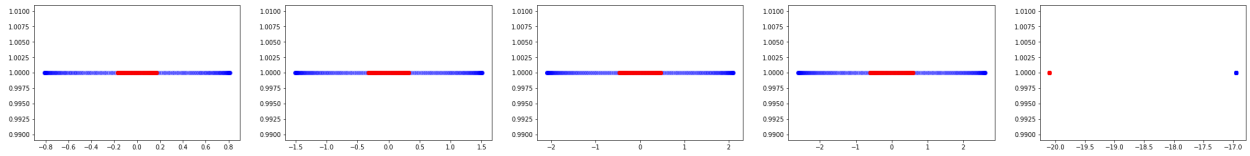

Figure 10: K-PCA results using gaussian kernel in one dimension
We start sigma at 0.1 and increase it by step of 0.01.

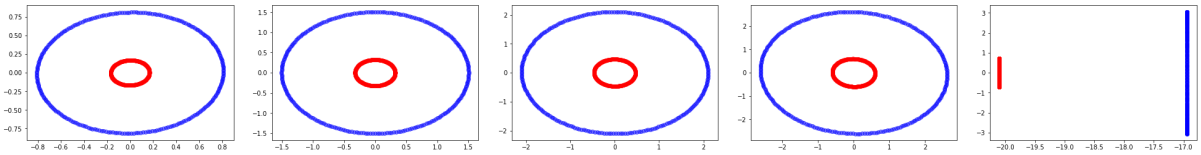

Figure 11: K-PCA results using gaussian kernel in two dimension
We start sigma at 0.1 and increase it by step of 0.01.

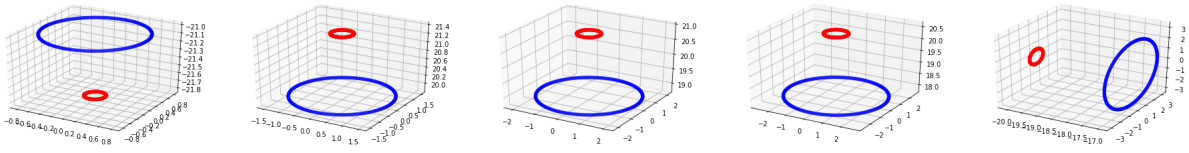

Figure 12: K-PCA results using gaussian kernel in three dimension
We start sigma at 0.1 and increase it by step of 0.01.

```
Processing time (moons gaussian 0.05 ) :  4.970557999999983 seconds
```

The gaussian work pretty well on the circles dataset.

1.3.5 K-PCA : Polynomial

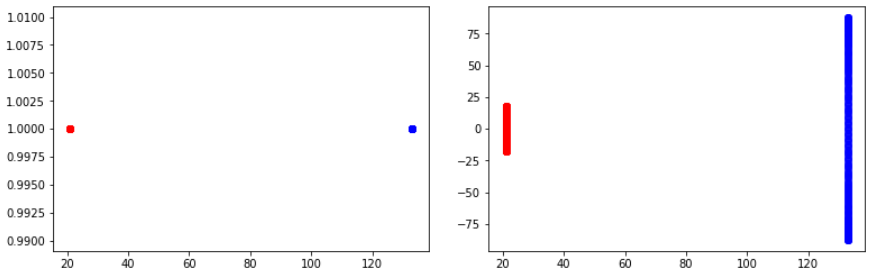

Figure 13: K-PCA results using polynomial kernel in one & two dimensions

```
Processing time (moons poly) :  2.4073649999999986 seconds
```

1.4 IRIS

1.4.1 Sample

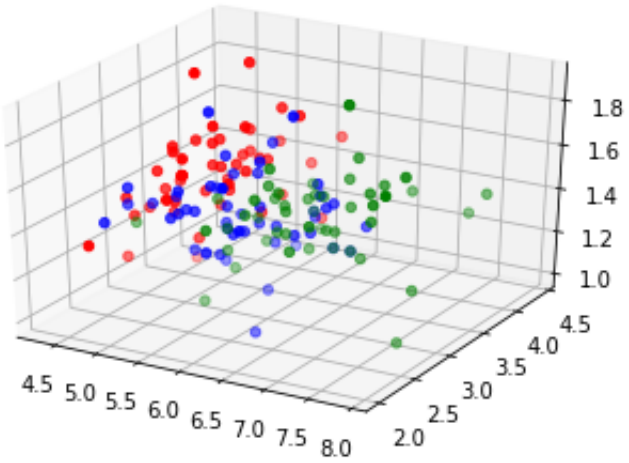

Figure 14: IRIS sample, plot with the 3 first components

1.4.2 PCA

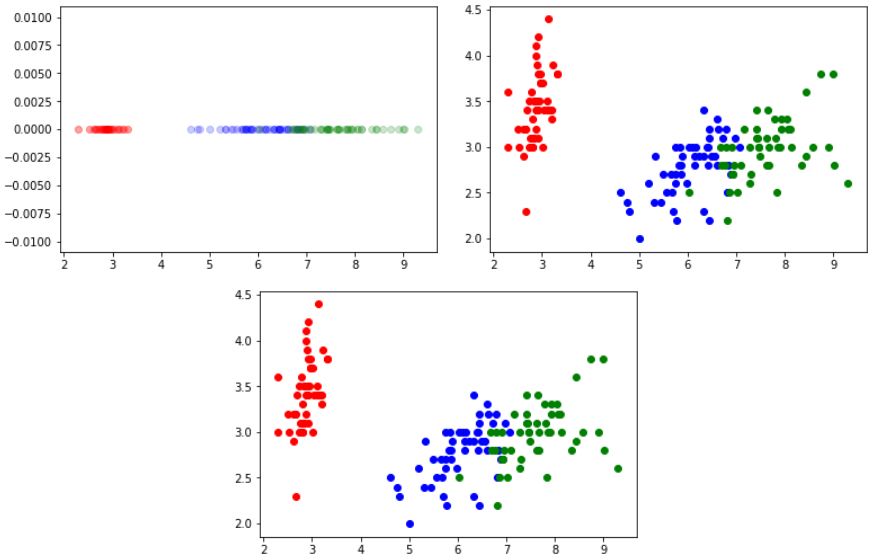

Figure 15: PCA results in one, two & three dimensions

```
Processing time for PCA : 0.5740129999999999 seconds
```

1.4.3 K-PCA : Linear

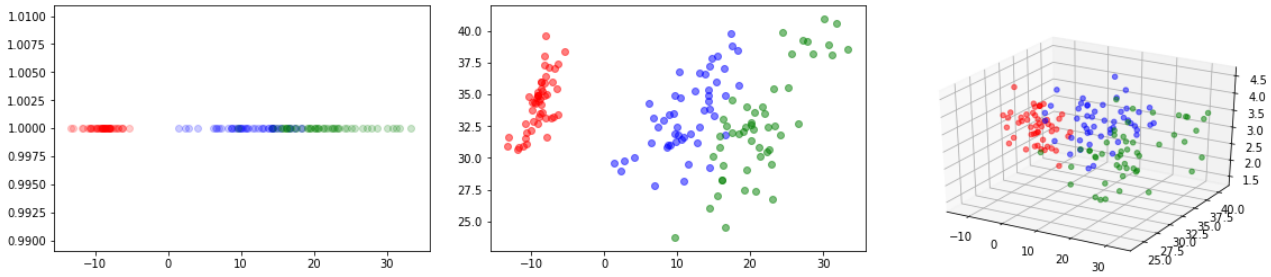

Figure 16: K-PCA results using linear kernel in one, two & three dimensions

```
Processing time (IRIS linear) :  0.6345670000000001 seconds
```

1.4.4 K-PCA : Gaussian

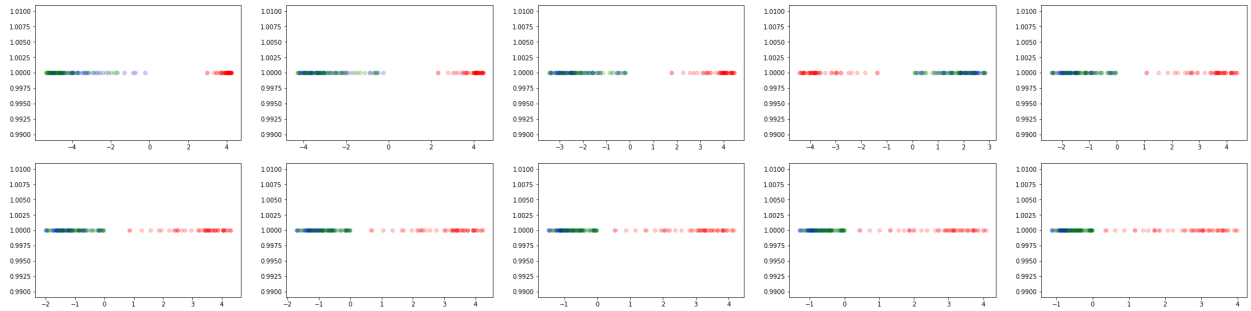

Figure 17: K-PCA results using gaussian kernel in one dimension
We start sigma at 1 and increase it by step of 1.

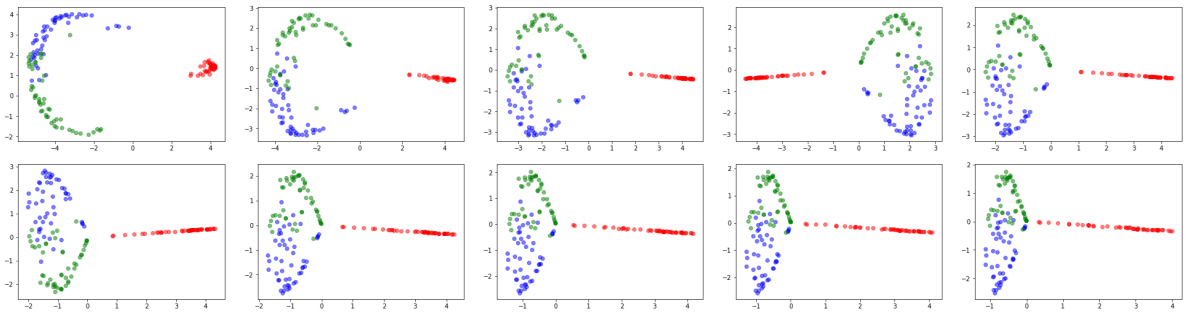

Figure 18: K-PCA results using gaussian kernel in two dimension
We start sigma at 1 and increase it by step of 1.

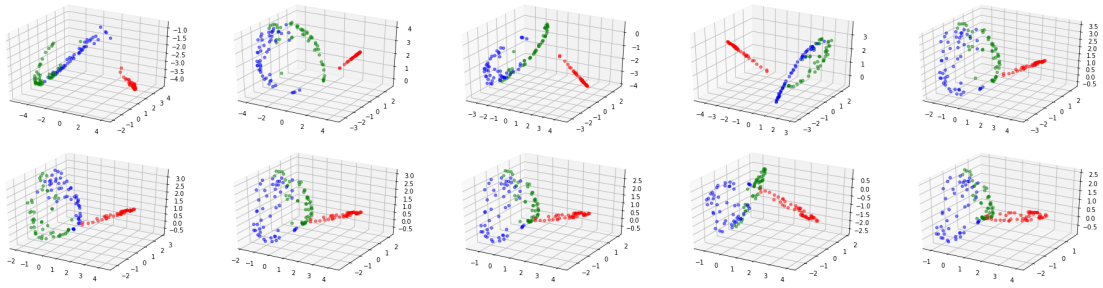

Figure 19: K-PCA results using gaussian kernel in three dimension
We start sigma at 1 and increase it by step of 1.

```
PProcessing time (IRIS gaussian 10 ) :  0.9170499999999997 seconds
```

The result are not so bad. The red class is well separated, the green and blue ones are quasi well separated.

1.4.5 K-PCA : Polynomial

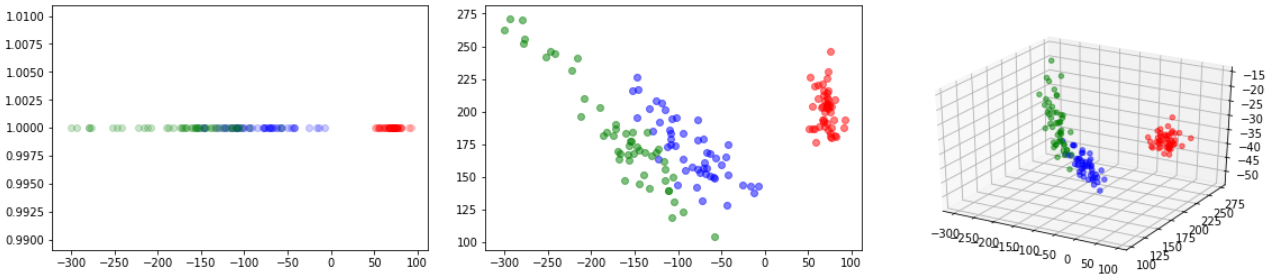

Figure 20: K-PCA results using polynomial kernel in one, two & three dimensions

```
Processing time (IRIS poly) :  0.7323779999999971 seconds
```

As the gaussian, the polynomial work pretty well.

2 Kernel-KMeans

2.1 Regular KMeans implementation

Here is our implementation of KMeans in python :

```
def kmeans_centroids(X, n, eps):
    # n centroids initialization (just once)
    np.random.seed(0)
    r = [np.random.randint(0,len(X)) for i in range(n)]

    #todo check r for doubles
    centroids_before = X[r]

    while True:
        #
        y_c = kmeans_fit(X,centroids_before)

        centroids_after = np.zeros((n,len(X[0])))
        sum_moove = 0
        for c in range(n):
            for d in range(np.size(X,axis=1)):
                centroids_after[c,d] = np.mean(X[y_c==c,d])
            # adding the distance from the old centroid to the new one
            sum_moove += np.linalg.norm(centroids_before[c] - centroids_after[c])

        if (sum_moove <= eps): break

        centroids_before = centroids_after

    return centroids_after

def kmeans_fit(X,centroids):
    y = np.zeros(len(X))
    dist = np.zeros(len(centroids))
    for i in range(len(X)):
        # computing the argmin distance to each centroid
        for c in range(len(centroids)):
            dist[c] = np.linalg.norm(X[i] - centroids[c])
        y[i] = np.argmin(dist)

    return y
```

2.2 Kernel KMeans

We just kernelize our data before doing the KMeans :

```
X_centered = X - np.mean(X)
X_normalized = X_centered / np.std(X)
X_normalized = kernel(X_normalized,gaussian,sigma=1)

centers = kmeans_centroids(X_normalized,n=2, eps=0.1)
y_c = kmeans_fit(X_normalized,centers)
```

2.3 Moons

2.3.1 Sample

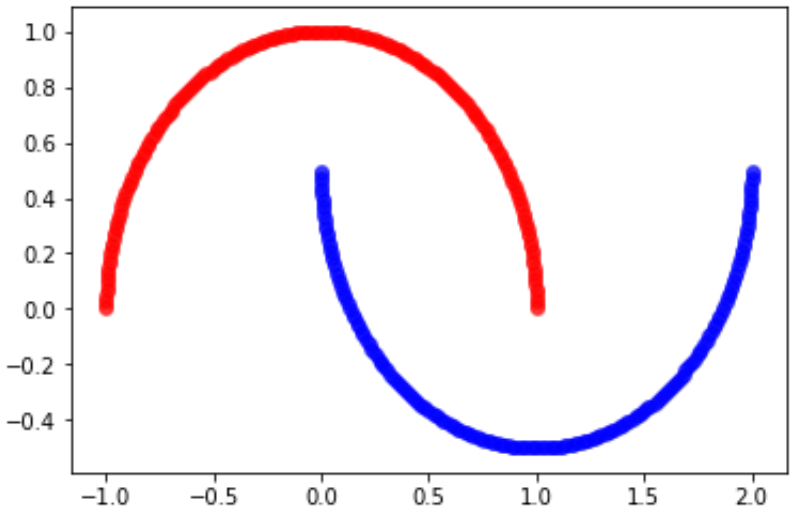

Figure 21: Moons sample

2.3.2 KMeans

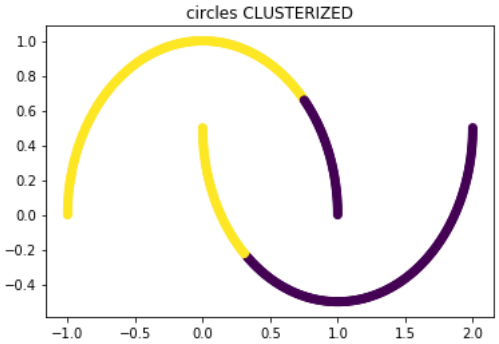

Figure 22: KMeans results

```
Processing time (moons kmeans) :  0.30734400000000006 seconds
```

Kmeans didn't work on Moons sample.

2.3.3 Kernel-KMeans : Linear

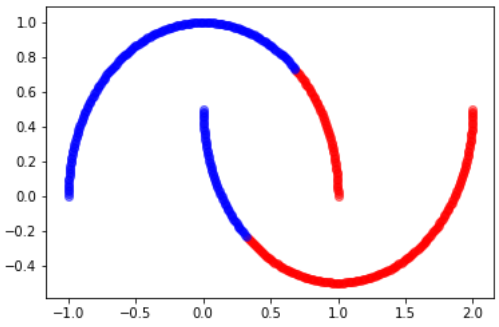

Figure 23: Kernel-KMeans results using Linear kernel

```
Processing time (moons linear) :  1.0717149999999998 seconds
```

Linear didn't work too and it's 3 times slower than kmeans

2.3.4 Kernel-KMeans : Gaussian

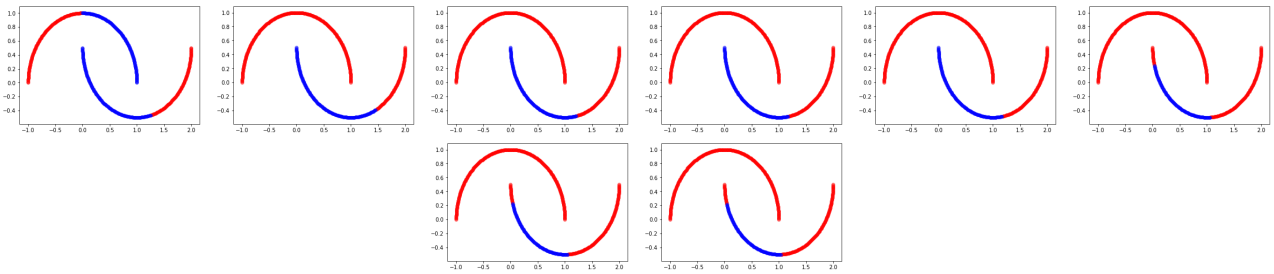

Figure 24: Kernel-KMeans results using Gaussian kernel
We start with a sigma = 1 and increase it by step of 4

```
Processing time (moons gaussian 19 ) :  3.6798220000000015 seconds
```

The gaussian kernel didn't work well, and that is strange because it work well in K-PCA.

2.3.5 Kernel-KMeans : Polynomial

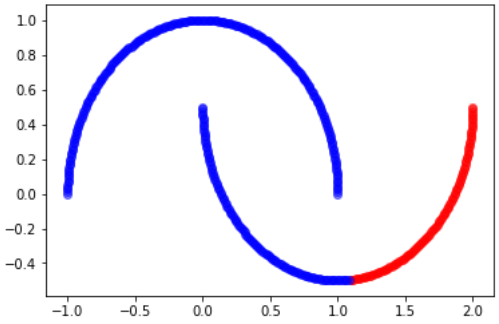

Figure 25: Kernel-KMeans results using Polynomial kernel

```
Processing time (moons poly) :  1.1784779999999984 seconds
```

Polynomial didn't work too.

2.4 Circles

2.4.1 Sample

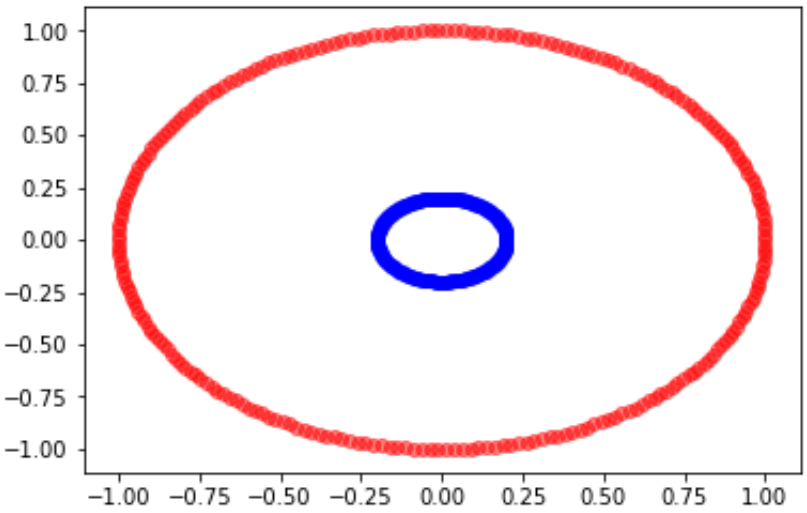

Figure 26: Circles sample

2.4.2 KMeans

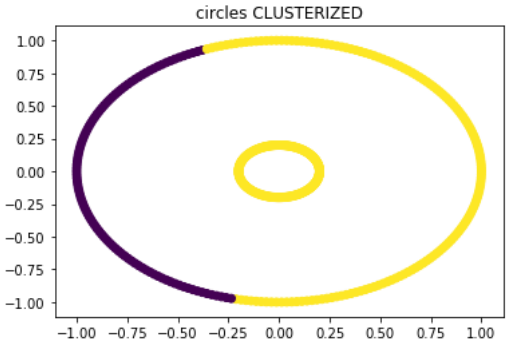

Figure 27: KMeans results

```
Processing time (circles kmeans) :  0.3013130000000075 seconds
```

Kmeans didn't work on Circles sample.

2.4.3 Kernel-KMeans : Linear

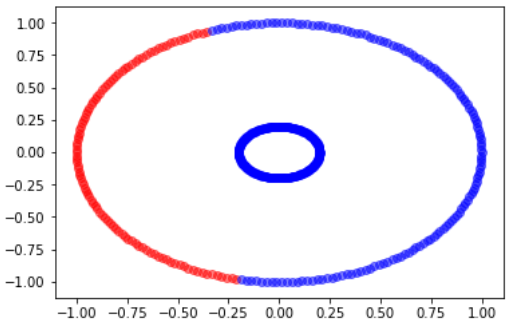

Figure 28: Kernel-KMeans results using Linear kernel

```
Processing time (circles linear) :  1.0568740000000076 seconds
```

Linear didn't work too and it's 3 times slower than kmeans

2.4.4 Kernel-KMeans : Gaussian

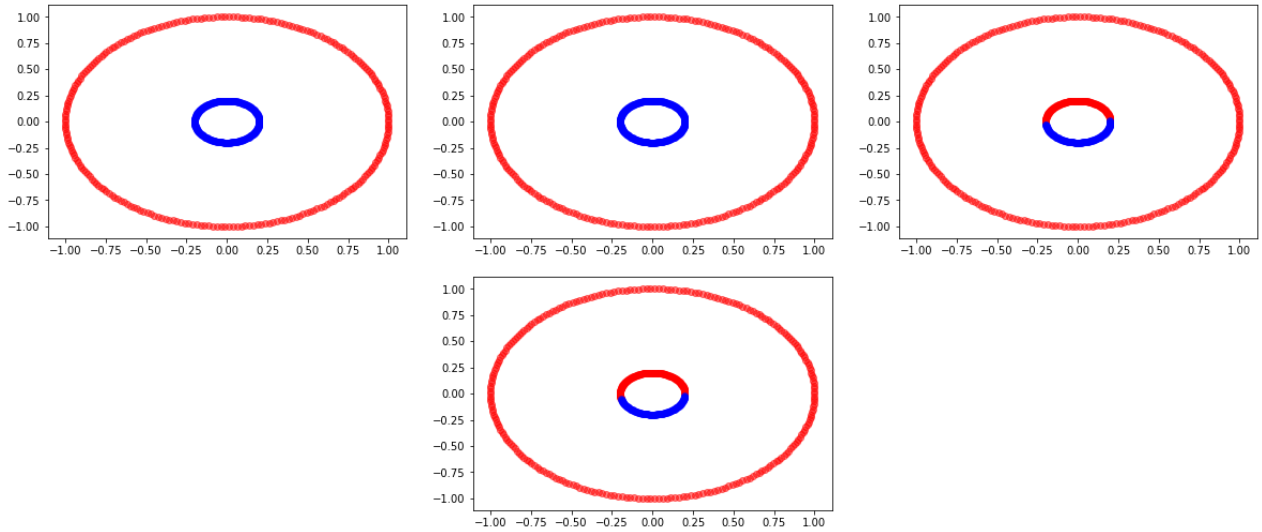

Figure 29: Kernel-KMeans results using Gaussian kernel
We start with a sigma = 1 and increase it by step of 4

```
Processing time (moons gaussian 13 ) :  3.9120200000000125 seconds
```

The gaussian kernel work perfectly, but the hyper-parameter sigma has not the same value than in K-PCA.

2.4.5 Kernel-KMeans : Polynomial

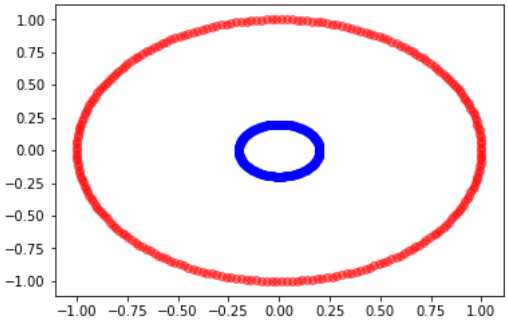

Figure 30: Kernel-KMeans results using Polynomial kernel

```
Processing time (moons poly) :  1.1784779999999984 seconds
```

Polynomial work too.

2.5 Iris

2.5.1 Sample

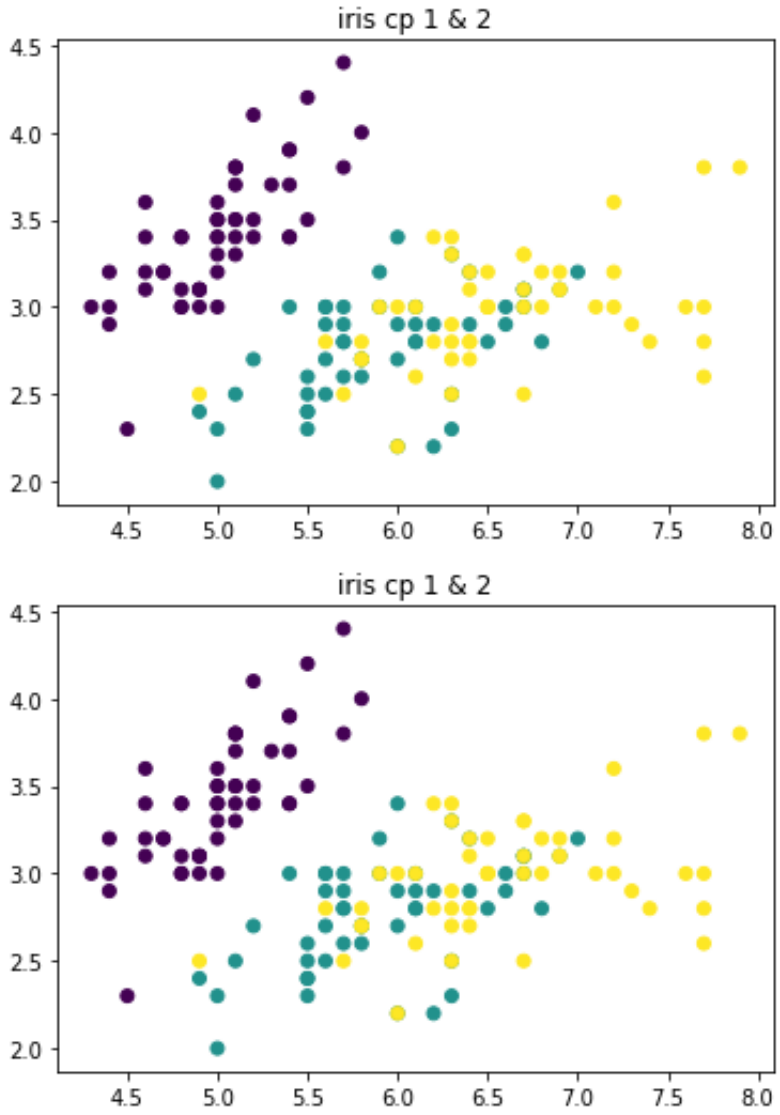

Figure 31: Iris sample

2.5.2 KMeans

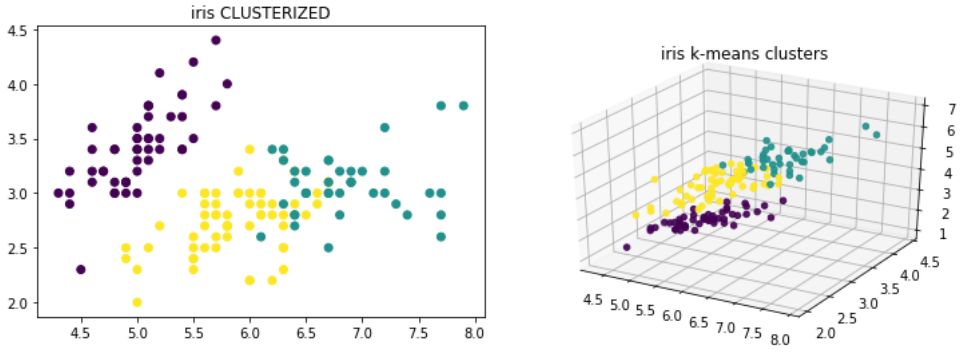

Figure 32: KMeans results

Kmeans work pretty well on Iris sample.

3 Logistic Regression

Unfortunately, we haven't done anything for this part, we prefered to do the next one before by lack of time.

4 One class SVM and Maximum Enclosing Ball

4.1 A first implementation

The first step is to write a simple 2D problem with a few data: a set X of points in D dimensions. We want to minimize the radius r of a ball $D = \{r, c\}$ with a center c such that :

$$\forall x \in X, \sum_{d=1}^{D} (X[i,d] - c[d])^2 \leq r^2$$

The first result that we get with $X = \{(0,0),(2,0),(0,2),(1,1)\}$ is $C = [1,1]$ and $r = 1.41 = \sqrt{2}$. Which is correct.
Here is how we implemented this problem:

```
reset;
option solver loqo;


# data
param n_feat;
param dim;
param X {1..n_feat, 1..dim};

# variables
var r >= 0;
var C {1..dim};

minimize ball: r;
subject to c1 {i in 1..n_feat}: (sum{d in 1..dim} ((X[i,d] - C[d]) ^ 2)) <= (r * r);


data;
param n_feat = 4;
param dim = 2;
param X: 1 2 :=
1 0 0
2 2 0
3 0 2
4 1 1 ;

solve;
display C;
display r;
```

4.2 Running on IRIS dataset

Then, we solve the MEB problem with single classes of the IRIS dataset using a program to convert the data that we get in python into a module for AMPL.
Here is the obtained maximum enclosing balls :

```
# CLASS 0 :
C [*] :=
1  5.1
2  3.35
3  1.4
4  0.35
;
```

```
r = 1.2145

# CLASS 1 :
C [*] :=
1  5.98135
2  2.75136
3  3.98539
4  1.28878
;
r = 1.35889

# CLASS 2 :
C [*] :=
1  6.33049
2  2.98564
3  5.65369
4  1.97031
;
r = 1.91996
```

Thus, we can project kernelize the data before solving the MEB problem as we have done it on KMeans.

List of Figures