Kernel Methods - Advanced Machine Learning

Beltrán Castro Gómez
beltran.castro.gomez@gmail.com

Abstract

This report aims to present a survey on kernel methods by showing the effect of kernels on different Machine Learning techniques: Principal Component Analysis, K-Means clustering, Maximum Mean Discrepancy measure and Metric Learning. For this purpose, already existing implementations of such topics have been compared with ad hoc implementations.

1. Kernel-PCA

Principant Component Analysis or PCA is a dimensionality reduction technique that projects data into a lower (or equal) dimensional space while trying to keep the most information possible. It is mainly used to visualize and interpret datasets with large numbers of features.

The classical PCA method computes what we call the *principal components* of the data, which actually correspond to eigenvectors of its covariance matrix. The components with the highest explained variance are kept and use for performing the transformation on the original data.

The kernelized version of PCA takes advantage of kernels to perform the decomposition in a Reproducing Kernel Hilbert Space (RKHS).

1.1. PCA Implementation

Scikit-Learn's (Pedregosa et al., 2011) PCA implementation includes three different Singular Value Decomposition (SVD) solvers and even an option to automatically detect the best of them based on the number of samples and features of the data passed. In my implementation, I have included both the three solvers as well, but without the automatic solver selection. The available SVD solvers are:

- *full*. Performs full SVD and selects the components afterwards.
- *arpack*. Performs SVD truncated to the selected numbers of components but imposes on them the restriction of being larger than the minimum dimension of the data.
- *randomized*. Performs randomized SVD (Halko et al., 2009).

This implementation includes, similarly to Scikit-Learn's, a sign flipping process. Due to the fact that singular value decomposition does not produce unique decompositions, sing flipping is performed to ensure uniqueness by imposing the rule that the element with the largest absolute value in the columns of U must have a positive sign.

1.2. Kernel-PCA Implementation

The implemented version of Kernel-PCA includes three posible kernels:

$$K(X,Y) = < X, Y > \tag{1}$$

the linear kernel (see Equation 1),

$$K(X,Y) = \gamma \cdot (< X, Y > + coef0)^{degree} \tag{2}$$

the polynomial kernel (see Equation 2),

$$K(X,Y) = \exp(-\gamma \cdot \|X - Y\|^2) \tag{3}$$

and the radial basis function kernel (see Equation 3).

In this case, *dense* and *arpack* eigensolvers are implemented, as well as the sign flip.

The steps for finding principal components with Kernel-PCA are the following:

1. Compute the gram matrix on the data with the given kernel function.
2. Center the gram matrix by the following equation:

   $$G_{centered} = G - 1_N \cdot G - G \cdot 1_N + 1_N \cdot G \cdot 1_N, \tag{4}$$

   being 1_N a nxn matrix filled with $1/n$ and n as the number of data points.
3. Find the eigenvectors and eigenvalues on the centered gram matrix.
4. Perform the sign flip if needed.
5. Normalize the eigenvectors by the square root of their respective eigenvalues and these would make the new components.

1.3. Method comparison

1.3.1. Linear separability

Doing an initial comparison of the PCA and Kernel-PCA methods by plotting their effect different datasets we can observe a couple of things.

First of all, in Figures 1 and 2, we can see that the outputs for both classical PCA and Kernel-PCA are exactly the same for my implementation and Scikit-Learn's one.

For a better appreciation of the differences between the PCA and Kernel-PCA approaches, I have decided to use 4 different datasets from Scikit-Learn: the moons dataset, the circles dataset, the blobs dataset and the gaussian quantiles dataset. For blobs, the number of features was set to two and the classes to three, and for the gaussian, the features to two and the classes to four. The size for all of them was of 1000 samples.

When training a simple Logistic Regression model on the whole datasets and then outputing its score to observe the degree of linear separability of the data, we obtain the results presented in table 1, with 2 components both for PCA and Kernel-PCA. After fine-tuning the kernel hyperparameters, all of the best scores were found with the use of the rbf kernel and with the gamma values of 17.5, 1.0, 0.007 and 4.22 for moons, circles, blobs and gaussian datasets, respectively.

What we can observe both in the figures and in the table, is that the PCA method does not improve the linear separability of the data at all, since it projects the original data in the same space. Kernel-PCA, however, allows the projection of the data in a different space thanks to the use of kernels, making it in the first two cases completely or almost completely linearly separable and making a big improvement in the case of the last dataset.

Furthermore, Kernel-PCA also enables the projection of the data in a higher dimensional space than the original one (not possible with classical PCA), making it almost always linearly separable for a number of dimensions higher than the number of data points. For the previous examples, we have that for the moons dataset, we can reach a score of 1.0 by just slightly increasing the number of dimensions, and for the gaussian case we can get to a score of 0.908 with just 10 components. With the blobs dataset, though, there is not much improvement regardless the dimensionality increase.

1.3.2. Quality of pre-processing for classification

For this experiment, my PCA and Kernel-PCA implementations were tested on Scikit-Learn's *make_classification* dataset, with 50 features and 10000 samples. A simple Logistic Regression model was trained on the 67% of the dataset and the accuracy was calculated on the other 33%. The baseline score for the classifier without PCA nor Kernel-PCA was of 0.8682.

The best results for classical and kernel PCA for 50 components can be observed in Figure 3, and the best scores for kernel PCA with more than 50 components can be observed in Figure 4.

1.3.3. Running time

For measuring the running time of my PCA and Kernel-PCA implementations and comparing them to Scikit-Learn running times, these were executed 100 runs each on the moons and circles datasets. The results can be seen in Table 2.

2. Kernel K-means

K-means is a clustering algorithm that groups data points into clusters with the closest mean (centroid). Similarly to Kernel-PCA, Kernel K-means makes use of kernels to apply the algorithm to non-linearly separable data.

2.1. K-Means implementation

K-means is an algorithm with a large training time, since it requires calculating every point's distance to every cluster centroid at each iteration, but with a low test time, since for a new point it only needs to calculate the distance of it to the cluster centroids.

In algorithm 1, we can see that the algorithm has two main phases: the assigment of the points' labels and the update of the cluster centroids. Nevertheless, there are another two key points taking into account by my implementation.

Firstly, the clusters initialization, that can be done in three different ways:

- by randomly chosing already existing points in the data
- by generating points from a uniform distribution taking the minimum and maximum values present in the data
- by passing the centroids as a parameter

Secondly, the policy that will be followed to deal with an empty cluster during the execution of the algorithm:

- Use the previous cluster mean
- Add one randomly selected point from another cluster
- Removig that cluster and changing k to k-1

Algorithm 1 K-Means

```
Input: data $X$, number of points $n$, number of clusters $k$
Initialize $oldLabels$
repeat
    for $i = 0$ to $n$ do
        Compute distances from $i$ to all clusters
        Assign $newLabel$ of i
    end for
    for $c = 0$ to $k$ do
        Update centroid
    end for
until $oldLabels$ is equal to $newLabels$
```

2.2. Kernel K-Means implementation

In the Kernel K-Means implementation, it is important to remark three aspects in which the algorithm differs with regards to classical K-Means:

1. For the cluster initialization, there is no need to calculate cluster centroids, but just to randomly assign each point to one of the labels.
2. Since the cluster centroids are not a mean of the member points as before but centers of mass, the assignment of a new label to a point will dynamically impact on the rest of the points' assignments.
3. The calculation of the distance from a point i to a cluster c is calculated by the formula in equation 5

$$d_{ic} = K(i,i) - \frac{2}{|c|}\sum_{j\in c} K(i,j) + \frac{1}{|c|^2}\sum_{j,l\in c} K(j,l) \tag{5}$$

2.3. K-Means comparison

Taking into account the implicit randomness of the K-Means algorithm due to its initialization, both implementations (mine and Scikit-Learn's) were each indepently run 10001 times on every dataset. Each point will be finally assigned the label which has previously been assigned the highest number of times during the complete experiment.

For my implementation, the clusters were initialized with already existing points and the empty clusters were treated with the strategy of adding a random point. For the Scikit-Learn case, the default *k-means++* initialization startegy was used, with *n_init* set to the default value of 10. For both cases, the maximum number of iterations was set to 1000.

For the sake of comparing the quality of the final clusters, the silhouette and homogeneity scores were computed.

The visualization of the resulting clustering can be observed in Figure 5 and the scores and average running time for both implementations in the different datasets are presented in Table 3.

2.4. Kernel K-Means comparison

For Kernel K-Means, considering the higher running times, the datasets' size was reduced from 1000 samples to 100 samples and the number of independent runs was set to 101.

My implementation was compared to Scikit-Learn's Spectral Clustering, which, even though they are not the same method, after enough runs, they should still converge similarly. In both cases, the rbf kernel was used. For Spectral Clustering, the *init* and *n_init* parameters were again set to default *k-means++* and 10 values.

Two different experiments were conducted since the results vary a lot due to the little number of runs: the the visualization of the resulting clusters for the first one can be observed in Figure 6 and the scores and average running time are presented in Table 4, and for the second experiment the resulting clusters can be observed in Figure 7, while the scores and running times are displayed in Table 5.

3. Maximum Mean Discrepancy

Maximum Mean Discrepancy or MMD (Tunali) computes the distance between two probabilistic distributions P and Q and is defined as:

$$MMD(P,Q) = ||\,\mathbb{E}_{x\sim P}[x] - \mathbb{E}_{y\sim Q}[y]\,|| \tag{6}$$

MMD can be extended to any mapping ϕ in the following way:

$$\begin{aligned} MMD(P,Q) &= ||\,\mathbb{E}_{x\sim P}[\phi(x)] - \mathbb{E}_{y\sim Q}[\phi(y)]\,|| \\ &= ||\mu_P - \mu_Q|| \end{aligned} \tag{7}$$

And by developing the square of the previous expression we finally obtain:

$$\begin{aligned} MMD^2(P,Q) &= <\mu_p,\mu_P> -2<\mu_P,\mu_Q> \\ &+ <\mu_Q,\mu_Q> \\ &= \mathbb{E}_{x\sim P}[K(x,x)] -2\cdot\mathbb{E}_{x\sim P,y\sim Q}[K(x,y)] \\ &+ \mathbb{E}_{y\sim Q}[K(y,y)] \end{aligned} \tag{8}$$

One experiment was conducted to compare the MMD distance between different domains of the *Office/Caltech* dataset in three different situations: in the original domain space, after performing classical PCA and after performing Kernel PCA. The results are presented in Table 6.

4. Metric Learning

With the objective of studying the effect of Kernels in Metric Learning, the Large margin nearest neighbor (LMNN) and Neighborhood Components Analysis (NCA) implementations from the *metric-learn* (de Vazelhes et al., 2020) library were used.

The algorithms were tested with the help of a 1-NN classifier on the *make_classification* dataset from Scikit-Learn. The generated data had a shape of (1000, 10) and it was divided into training and testing sets with a $60 : 40$ ratio.

The baseline score for the classifier was of 0.87 and the MMD between positive and negative instances of 0.0091.

After applying the LMNN transformation with $k = 10$ on the original data, the MMD grew up to 0.0281. However, when applying the NCA transformation, it decreased to 0.0067. The 1-NN classifier achieved an score of 0.8825 with the help of the LMNN metric and of 0.87 with the NCA metric.

As a result of applying Kernel-PCA with an rbf kernel and 15 components on the original data, the new base score changed to 0.8775 and the MMD to 0.1346. When transforming the projected data with the Metric Learning algorithms, the MMD for the LMNN case (with $k = 10$ again) decreased to 0.0308 and to 0.0067 for NCA. In terms of accuracy, the 1-NN classifier with the LMNN metric improved the score to 0.8925 and to 0.88 with the NCA one.

References

de Vazelhes, W., Carey, C., Tang, Y., Vauquier, N., and Bellet, A. metric-learn: Metric Learning Algorithms in Python. *Journal of Machine Learning Research*, 21(138): 1–6, 2020.

Halko, N., Martinsson, P.-G., and Tropp, J. A. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. 2009. doi: 10.48550/ARXIV.0909.4061. URL https://arxiv.org/abs/0909.4061.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Tunali, O. Maximum mean discrepancy (mmd) in machine learning. https://www.onurtunali.com/ml/2019/03/08/maximum-mean-discrepancy-in-machine-learning.html. Accessed: 2022-12-04.

A. Figures

The following section contains all of the figures referred to in this document. They have been displayed here for a better visualization taking advantage of the one-column format.

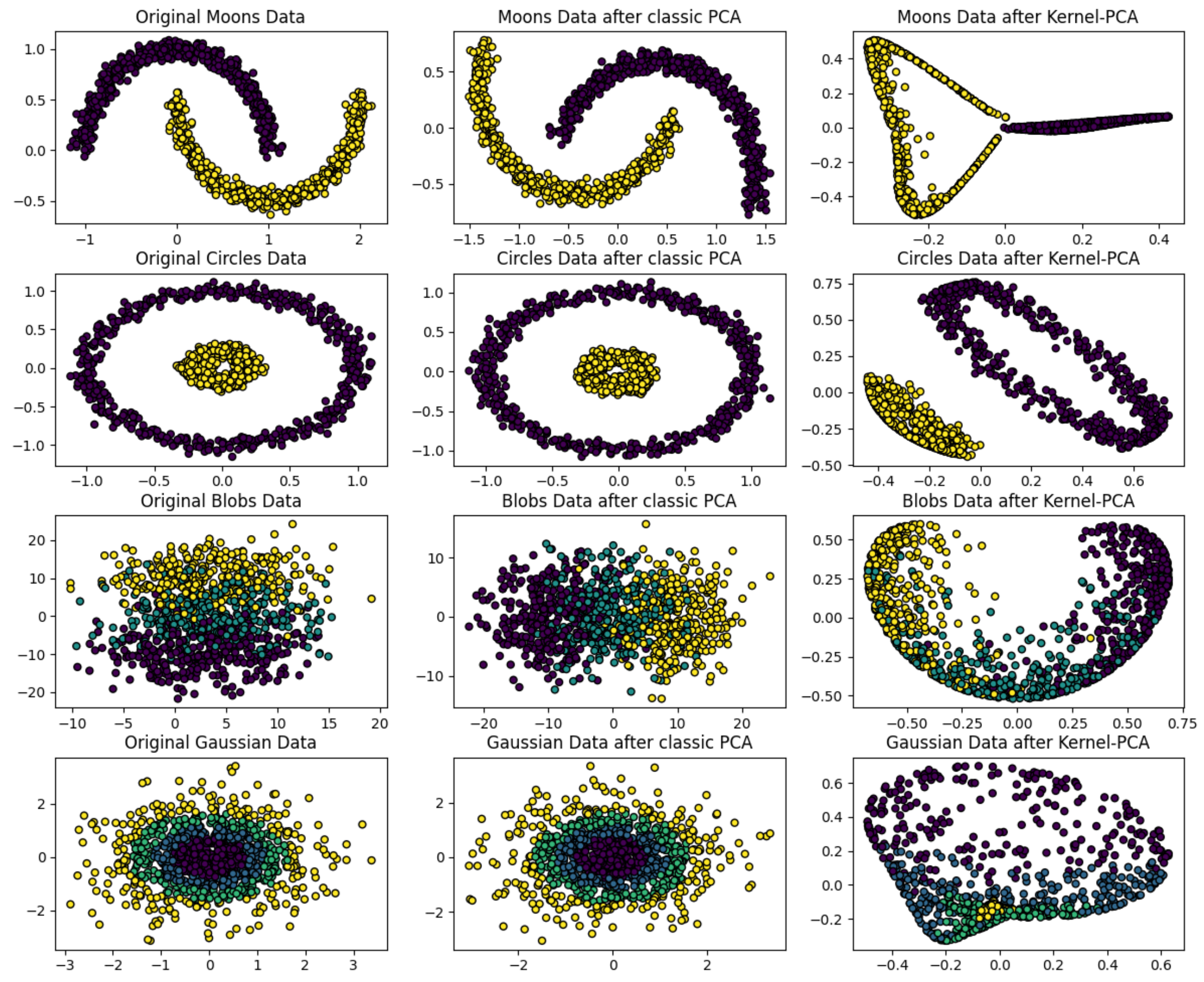

Figure 1. Comparison of different datasets with and without PCA approches using Scikit-Learn's implementations.

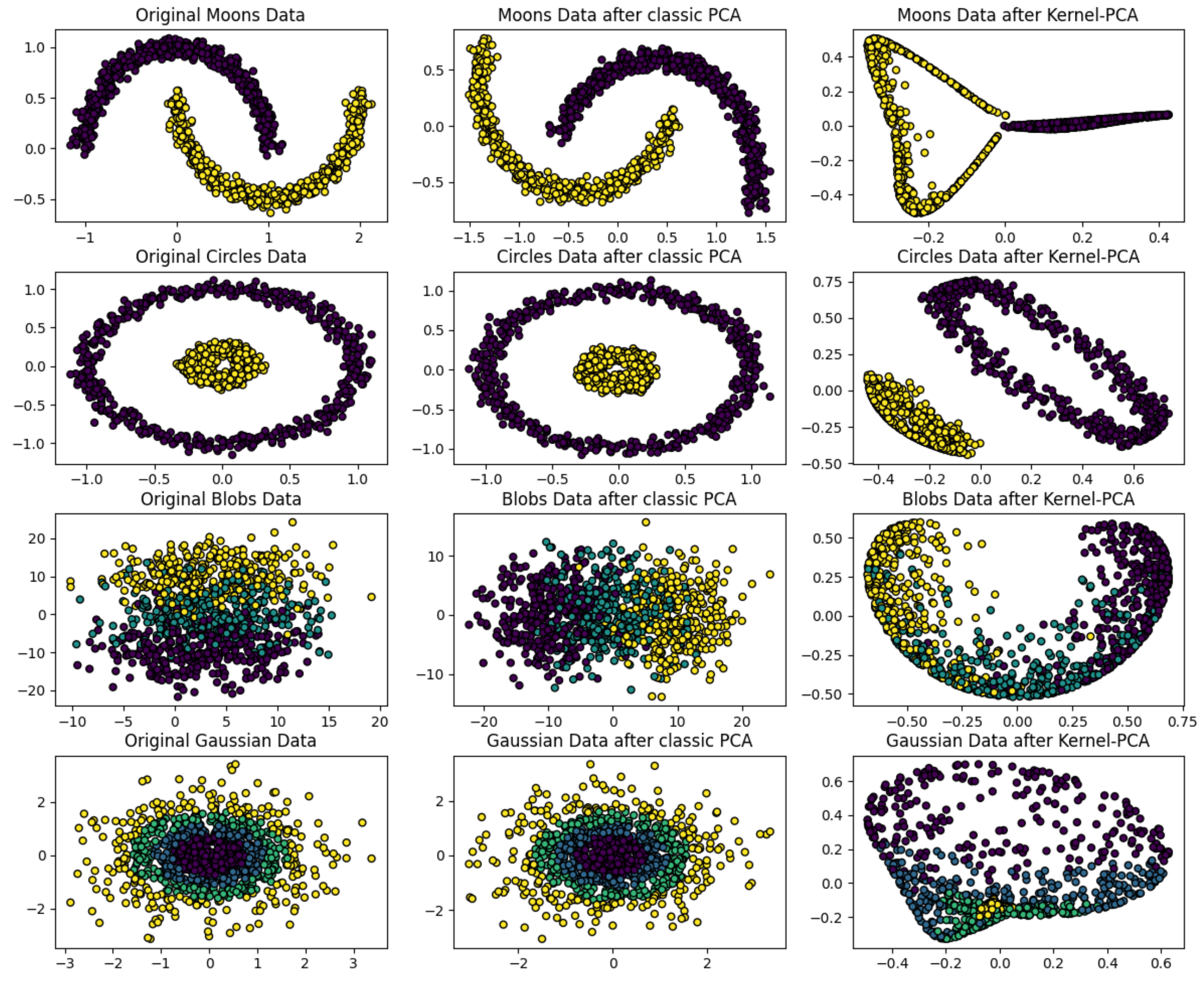

Figure 2. Comparison of different datasets with and without PCA approches using my own implementations.

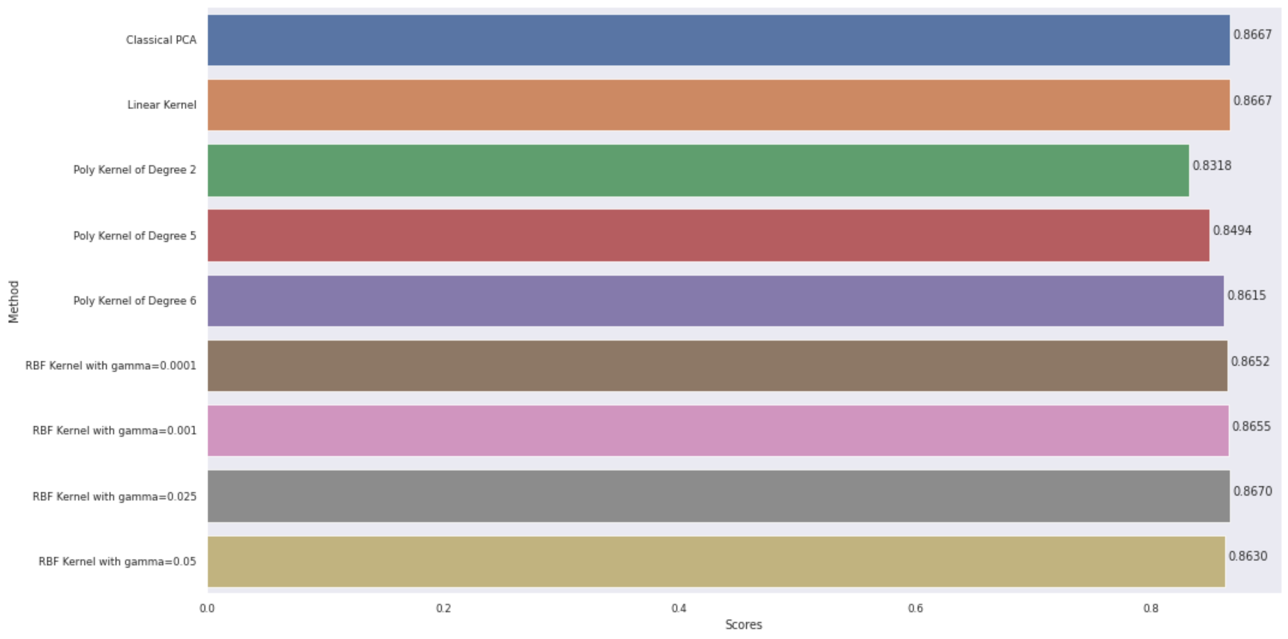

Figure 3. Best scores for logistic regressor after PCA and Kernel PCA with 50 components on a classification task.

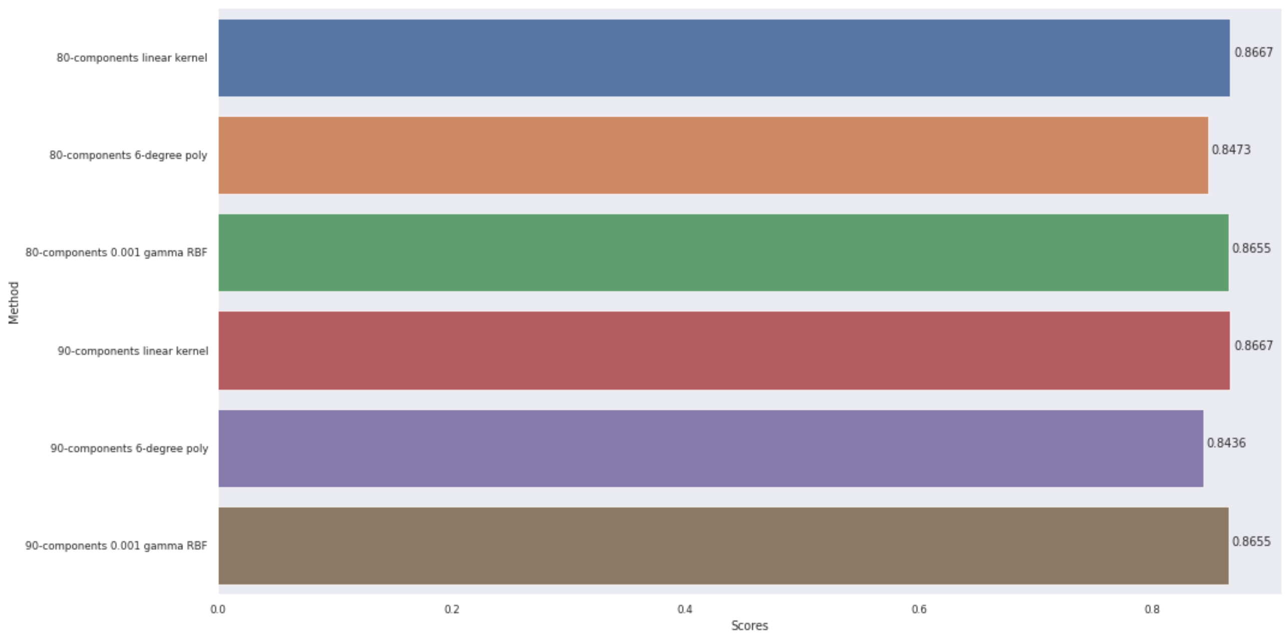

Figure 4. Best scores for logistic regressor after Kernel PCA with 80 and 90 components on a classification task.

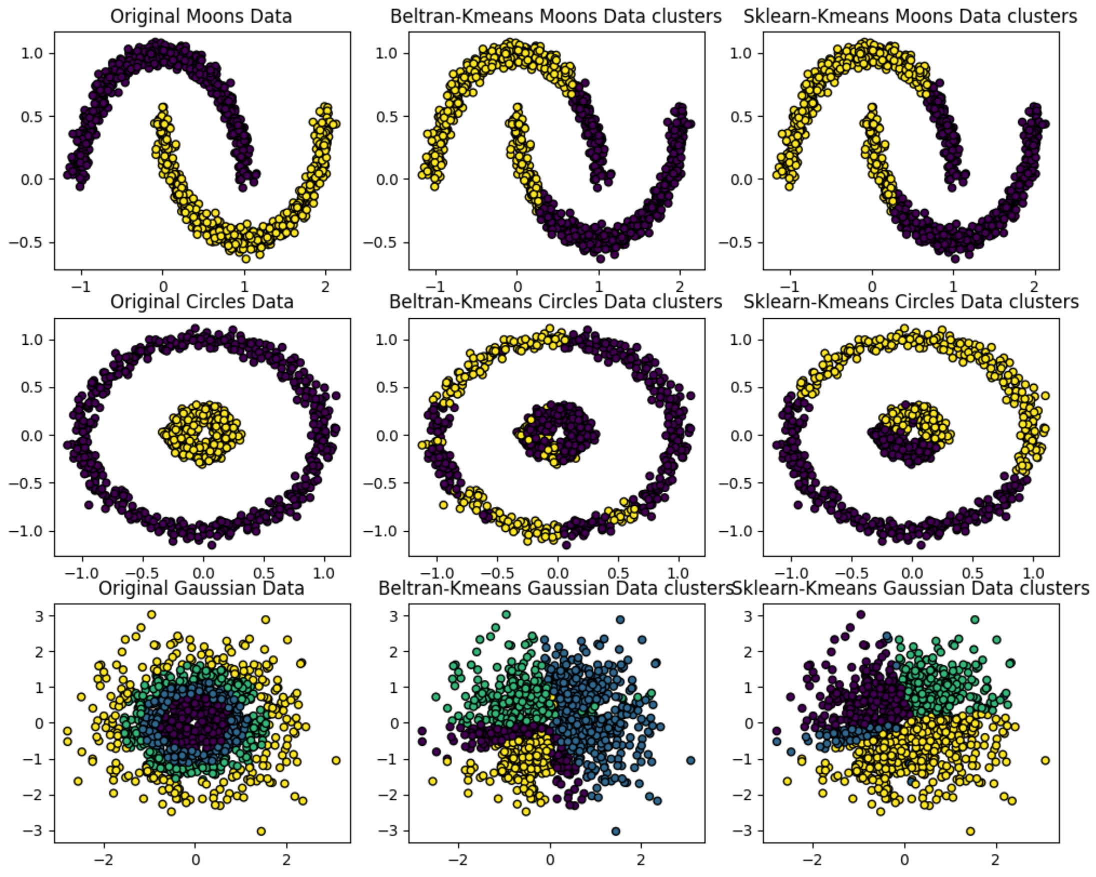

Figure 5. Comparison of KMeans clustering on different datasets.

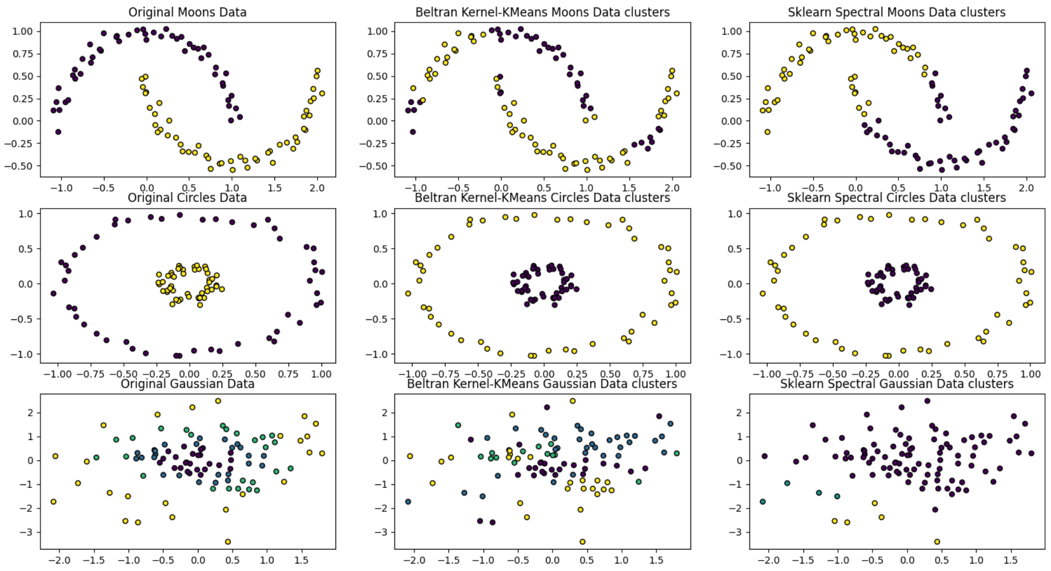

Figure 6. First comparison of Kernel-KMeans clustering on different datasets.

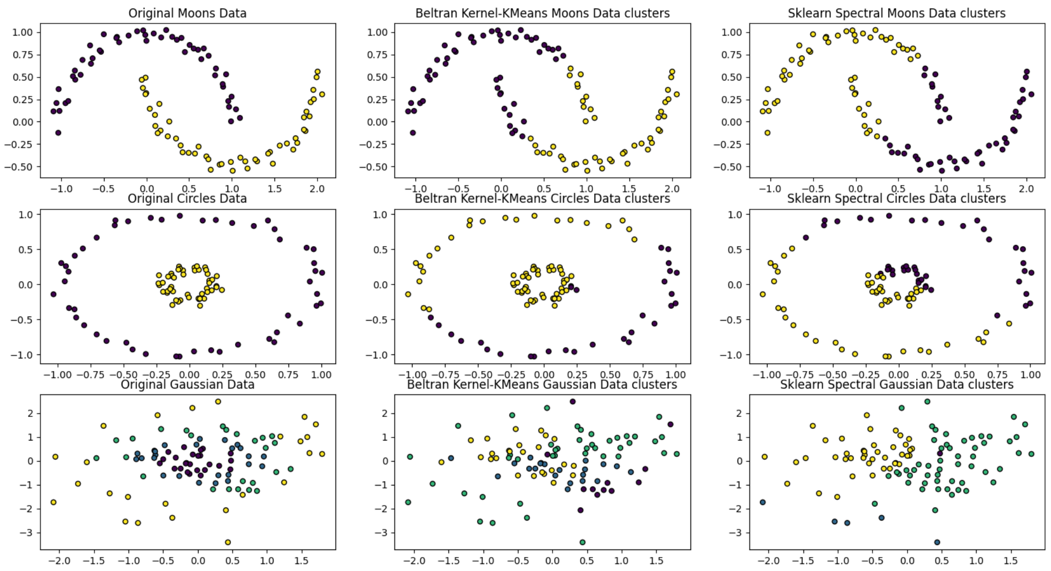

Figure 7. Second comparison of Kernel-KMeans clustering on different datasets.

B. Tables

The following section contains all of the tables referred to in this document. They have been displayed here for a better visualization taking advantage of the one-column format.

Dataset	Original	PCA	K-PCA
Moons	0.891	0.891	0.998
Circles	0.499	0.499	1.0
Blobs	0.781	0.781	0.784
Gaussian	0.284	0.284	0.739

Table 1. Classification accuracies for logistic regression on the original datasets, and after performing PCA and Kernel-PCA on them.

Dataset	PCA	Implementation	Avg. Time [s]
Moons	Classical	Beltran	0.0088
Moons	Classical	Scikit-learn	0.0004
Moons	Kernelized	Beltran	4.6023
Moons	Kernelized	Scikit-learn	0.0333
Circles	Classical	Beltran	0.0079
Circles	Classical	Scikit-learn	0.0004
Circles	Kernelized	Beltran	4.6003
Circles	Kernelized	Scikit-learn	0.0289

Table 2. Average running time comparison of PCA and Kernel-PCA implementations.

Dataset	Implementation	Silhouette Score	Homogeneity Score	Avg. Time [s]
Moons	Beltran	0.4904	0.1935	0.0908
Moons	Scikit-Learn	0.4904	0.1935	0.0136
Circles	Beltran	0.1779	0.0699	0.1119
Circles	Scikit-Learn	0.2532	0.0005	0.0186
Gaussian	Beltran	0.1981	0.0059	0.3094
Gaussian	Scikit-Learn	0.2018	0.0130	0.0195

Table 3. Scores and running times for the K-Means comparison.

Dataset	Implementation	Silhouette Score	Homogeneity Score	Avg. Time [s]
Moons	Beltran	0.0796	0.1365	7.5415
Moons	Scikit-Learn	0.4717	0.3425	0.1421
Circles	Beltran	0.2498	1.0	6.5498
Circles	Scikit-Learn	0.2498	1.0	0.1413
Gaussian	Beltran	-0.0066	0.0979	6.2510
Gaussian	Scikit-Learn	0.3666	0.1004	0.1891

Table 4. Scores and running times for the first Kernel K-Means experiment.

Dataset	Implementation	Silhouette Score	Homogeneity Score	Avg. Time [s]
Moons	Beltran	0.4913	0.2054	8.3377
Moons	Scikit-Learn	0.4913	0.2054	0.1422
Circles	Beltran	0.3320	0.2199	7.7568
Circles	Scikit-Learn	0.2529	0.0026	0.1276
Gaussian	Beltran	-0.0423	0.1626	5.5696
Gaussian	Scikit-Learn	-0.1034	0.0825	0.1614

Table 5. Scores and running times for the second Kernel K-Means experiment.

Domain 1	Domain 2	Space	MMD
dslr	webcam	Original Space	0.0098
dslr	webcam	After PCA with 10 components	0.0559
dslr	webcam	After PCA with 50 components	0.0327
dslr	webcam	After PCA with 100 components	0.0241
dslr	webcam	After PCA with 150 components	0.0199
dslr	webcam	After Kernel-PCA with 10 components	1.7520e-06
dslr	webcam	After Kernel-PCA with 50 components	1.4411e-06
dslr	webcam	After Kernel-PCA with 100 components	1.0689e-06
dslr	webcam	After Kernel-PCA with 150 components	5.1987e-07
amazon	caltech10	Original Space	0.0020
amazon	caltech10	After PCA with 10 components	0.0733
amazon	caltech10	After PCA with 50 components	0.0371
amazon	caltech10	After PCA with 100 components	0.0296
amazon	caltech10	After PCA with 150 components	0.0254
amazon	caltech10	After PCA with 500 components	0.0164
amazon	caltech10	After PCA with 800 components	0.0122
amazon	caltech10	After Kernel-PCA with 10 components	3.3535e-06
amazon	caltech10	After Kernel-PCA with 50 components	2.3983e-07
amazon	caltech10	After Kernel-PCA with 100 components	6.9436e-08
amazon	caltech10	After Kernel-PCA with 150 components	3.0891e-08
amazon	caltech10	After Kernel-PCA with 500 components	1.3941e-09
amazon	caltech10	After Kernel-PCA with 800 components	5.9521e-09
amazon	caltech10	After Kernel-PCA with 900 components	7.2427e-09

Table 6. Comparison of MMD distances on the *Office/Caltech* dataset in different contexts.