On Kernel Principal Component Analysis

Cristina Aguilera and Jesús Antoñanzas
Data Science & Engineering, Technical University of Catalonia
(Dated: November 29, 2019)

Abstract

Principal component analysis (PCA) is a popular tool for linear dimensionality reduction and feature extraction. Kernel PCA is the non-linear counterpart of PCA, which can better exploit the complicated spatial structure of high-dimensional features. In this paper, we study how Kernel PCA is constructed, its properties, discuss its usefulness, complexity, compare its performance to the non-kernel version and discuss some variants.

I. INTRODUCTION

Extracting relevant information from data is key to a thorough understanding of the problem in question. The possibly high dimensionality of the data can be a real problem depending on the application, so it is often reduced. Although needed, reduction of dimensionality is of no practical use if the general structure and maybe important features of the data are lost. Principal Component Analysis, long for PCA, is a powerful statistical learning tool with which data is projected onto dimensions that explain most of its variance (*principal components*). It was developed by Pearson (1901) [1] and Hotelling (1933) [2], the most modern reference being Jolliffe (2002) [3]. The *principal components* are linear functions of the original variables which meet two requirements: they explain the most variability in decreasing order and are all pair-wise orthogonal. Once retrieved, the first few ones usually explain most of the structure of the data, allowing for information-preserving dimensionality reduction. In order to find the *principal components* of a centered dataset D, an eigenvalue problem on the covariance matrix C is solved [3]. The associated eigenvectors are, in decreasing order of eigenvalue the first *principal component*, the second *principal component*, ... , and so on.

And whereas PCA is very valid, many important real world data is not linearly explainable. Because of this, it is of great value to be able to find non-linear relationships, which can be done with *kernel methods*. *Kernel methods* are a family of statistical learning algorithms which work not in the input space I, but in a *feature space* F, the best known example being Support Vector Machines [4]. These methods are based on *kernel functions*, a measure of similarity between two mappings of a pair of points from I to F. The ability to work in F via the *kernel functions* allows to easily retrieve non-linear patterns of the data.

Consequently, instead of finding *principal components* in an input space I we want a way of finding them in the *feature space* F so that they are non-linearly correlated to the variables in I: Kernel PCA or kPCA for short. In this paper we are going to study kPCA: how it is constructed, its properties and utilities and how it can be implemented. Then, we are going to perform some brief experiments aimed at exploiting the potential of kPCA versus its counterpart linear PCA.

II. KERNELS

Many machine learning techniques are based on kernel functions, which provide great flexibility when dealing with different kinds of problems and allow for the study of the data in another, hopefully more useful, space.

Definition 1. Kernel Function Let $\mathcal{X} \neq 0$. A function $k : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ is a positive semi-definite kernel function $\Leftrightarrow$

1. **Symmetry**
$$\forall x, x` \in \mathcal{X}, k(x, x') = k(x', x) \tag{1}$$

2. **Positive semi-definiteness** $\forall n \in \mathbb{N}^+, \forall x_1, ..., x_n \in \mathcal{X}$,
$$\sum_{i=1}^{n} \sum_{j=1}^{n} c_i k(x_i, x_j) c_j \geq 0, \forall c \in \mathbb{R}^n \tag{2}$$

In the machine learning community, a similarity measure between a pair of points is a common interpretation for kernel functions. As the points can exist in a variety of spaces, we would like them to be in a space where their structure can be more easily explained. For that to happen, we introduce a mapping function ϕ, which can be of many kinds. ϕ then allows us to introduce prior knowledge of the problem being analysed into question.

Theorem 1. *[5] Let k be a kernel function in some space $\mathcal{X}$. Then, there exists a Hilbert space of functions $\mathcal{H}$ and a mapping $\phi : \mathcal{X} \rightarrow \mathcal{H}$ such that $k(x, x') = \langle \phi(x), \phi(x') \rangle_{\mathcal{H}}$.*

This Hilbert space $\mathcal{H}$ is what we call the *feature space*. Theorem 1 means that we can compute similarity of a pair of points in a feature space. But what really makes kernel functions powerful is the fact that we do not have to compute $\phi(x)$ nor know what ϕ is to compute $k(x, x')$. This useful implicit mapping contained within a kernel function k is called the *kernel trick*. Some examples of common kernel functions include:

- **Gaussian RBF kernel:**
$$k(x, x') = exp(-\gamma \|x - x'\|^2), \gamma > 0, \gamma \in \mathbb{R} \tag{3}$$

- **Polynomial kernel:**
$$k(x, x') = (x^T x' + c)^q, c \geq 0, q \in \mathbb{N}^+ \tag{4}$$

As it turns out, all kernel functions are inner products and vice-versa [6]. This means that, given a statistical learning algorithm, if we can express it in terms of inner products between

the data points, then it can be kernelized:
Given $\phi : \mathbb{R}^p \to \mathcal{H}$ and $x_i, x_j \in \mathbb{R}^p$, if we had $x_i^T x_j = \langle x_i, x_j \rangle_{\mathbb{R}^p}$, we could work instead with the data points in some *feature space* $\langle \phi(x_i), \phi(x_j) \rangle_{\mathcal{H}} = k(x_i, x_j)$ which can be expressed in terms of the kernel matrix K as the element K_{ij}. Going deeper into kernels is out of the scope of this paper.

III. PRINCIPAL COMPONENT ANALYSIS

The aim of PCA is to look for linear combinations of the original variables in a data set $\mathcal{D}$ (*principal components*) that preserve most of the information given by its variance.

Given a data set $\mathcal{D} = \{x_1, x_2, ..., x_n\}$ of centered observation such that each $x_i \in \mathbb{R}^p$ and $\sum_{i=1}^n x_i = 0$, PCA diagonalizes the covariance matrix

$$C = \frac{1}{n} \sum_{i=1}^{n} x_i x_i^T \, . \tag{5}$$

To do this, an eigenvalue problem has to be solved derived from the following equation

$$Cv = \lambda v \, . \tag{6}$$

The eigenvectors v of C, sorted in decreasing order by the corresponding eigenvalue λ, correspond to the *principal components*. Up to p PCs can be found, but if many of the original variables have strong correlation between them, the most of the original variation will be accounted for by the first m PCs, where $m \ll p$. Finally, we need to compute projections $y(x)$ of new data points into the *principal components*. Let $x \in \mathbb{R}^p$ be a test point, then

$$y(x) = Vx \, , \tag{7}$$

V being the eigenvector matrix.

IV. PCA IN FEATURE SPACES

In order to develop the expression of PCA in a feature space, the first thing to do is substitute the expressions of the points $x_i \in \mathcal{D}, x_i \in \mathbb{R}^p$ for their projection into the feature space $\mathcal{H}$ via the mapping function $\phi : \mathbb{R}^p \to \mathcal{H}$. Then, we follow the procedure for the construction of PCA.

- **Centering the data points in $\mathcal{H}$.** First, the data points $\{\phi(x_1), \phi(x_2), ..., \phi(x_n)\}$ need to be centered in the feature space. That is, $\sum_{i=1}^n \phi(x_i) = 0$. $\phi^c(x_i)$ indicates the

centered mapping to the feature space of x_i. A centered point is:

$$\phi^c(x_i) := \phi(x_i) - \frac{1}{n}\sum_{j=1}^{n}\phi(x_j) \tag{8}$$

Because with kernelized algorithms one does not work explicitly with points in the feature space but with their similarity (kernel function), we develop the expression of the i,j-th element of the kernel matrix K^c obtained with the centered points:

$$\begin{aligned}
K^c_{ij} &= \langle \phi^c(x_i), \phi^c(x_j)\rangle_{\mathcal{H}} \\
&= \langle \phi(x_i) - \frac{1}{n}\sum_{l=1}^{n}\phi(x_l), \phi(x_j) - \frac{1}{n}\sum_{l=1}^{n}\phi(x_l)\rangle_{\mathcal{H}} && \text{substituting equation 8} \\
&= \langle \phi(x_i), \phi(x_j)\rangle_{\mathcal{H}} - \frac{1}{n}\sum_{l=1}^{n}\langle \phi(x_i), \phi(x_l)\rangle_{\mathcal{H}} \\
&\quad - \frac{1}{n}\sum_{l=1}^{n}\langle \phi(x_j), \phi(x_l)\rangle_{\mathcal{H}} + \frac{1}{n^2}\sum_{l=1}^{n}\sum_{l'=1}^{n}\langle \phi(x_l), \phi(x_{l'}\rangle && \text{using linearity of i.p.} \\
&= K_{ij} - \frac{1}{n}\sum_{l=1}^{n}K_{il} - \frac{1}{n}\sum_{l=1}^{n}K_{jl} + \frac{1}{n^2}\sum_{l=1}^{n}\sum_{l'=1}^{n}K_{ll'}
\end{aligned} \tag{9}$$

Given the expression for each element of K^c, we can aggregate to give the formula for the whole matrix:

$$K^c := K - \frac{1}{n}[JK + KJ] + \frac{1}{n^2}JKJ \tag{10}$$

Being J the $(n \times n)$ ones matrix. This result is, in fact, the expression for the centering of K, which can also be obtained by developing the expression M_cKM_c, where $M_c = (I_{n\times n} - \frac{1}{n}J)$ is the centering matrix. This fact means that in order to work with centered points in the *feature space* one just has to center the kernel matrix K. For notation simplicity, we assume for the rest of the section that $\phi(x_i)$ is already centered.

- **Solving an eigenvalue problem.** Given the covariance matrix C_ϕ of the points in the *feature space*

$$C_\phi = \frac{1}{n}\sum_{i=1}^{n}\phi(x_i)\phi(x_i)^T \tag{11}$$

we need to find its eigenvalues and associated eigenvectors such that, for $v \in \mathcal{H}$

$$\begin{aligned}
&\lambda v = C_\phi v && (12)\\
&\lambda v = \frac{1}{n}\sum_{i=1}^{n}\phi(x_i)\phi(x_i)^T v && \text{substituting expression 11}
\end{aligned}$$

from where v can be re-expressed as a linear combination of $\phi(x_i), i = 1, ..., n$:

$$\begin{aligned}
\lambda v &= \frac{1}{n}\sum_{i=1}^{n}\phi(x_i)\phi(x_i)^T v \\
v &= \sum_{i=1}^{n}\underbrace{\frac{1}{n\lambda}(\phi(x_i)^T v)}_{\alpha_i}\phi(x_i) \qquad\qquad \exists\alpha \in \mathbb{R}^p \\
v &= \sum_{i=1}^{n}\alpha_i\phi(x_i) \qquad\qquad (13)
\end{aligned}$$

which in other terms can be seen as v being the span of $\{\phi(x_1), \phi(x_2), ..., \phi(x_n)\}, \lambda > 0$. So, we can consider the following equations instead of (12):

$$\lambda v\,\phi(x_k) = C_\phi v\,\phi(x_k) \qquad \forall k = 1, ..., n. \qquad (14)$$

Using (13) and (11) we may develop (14) for all $k = 1, ..., n$:

$$\begin{aligned}
\langle\,\lambda\sum_{i=1}^{n}\alpha_i\phi(x_i), \phi(x_k)\,\rangle_{\mathcal{H}} &= \langle\,\frac{1}{n}\sum_{i=1}^{n}\phi(x_i)\langle\,\phi(x_i), \sum_{j=1}^{n}\alpha_j\phi(x_j)\,\rangle_{\mathcal{H}}, \phi(x_k)\,\rangle_{\mathcal{H}} \\
\lambda\sum_{i=1}^{n}\alpha_i\langle\,\phi(x_i), \phi(x_k) &= \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_j\langle\,\phi(x_i), \phi(x_j)\,\rangle_{\mathcal{H}} \cdot \langle\,\phi(x_i), \phi(x_k)\,\rangle_{\mathcal{H}} \\
\lambda\sum_{i=1}^{n}\alpha_i K_{ik} &= \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_j K_{ij}K_{ik} \qquad (15)
\end{aligned}$$

where $K_ij = \phi(x_i)\phi(x_j)$, giving us to

$$\lambda K\alpha = \frac{1}{n}K^2\alpha \;\rightarrow\; \lambda\alpha = \frac{1}{n}K\alpha. \qquad (16)$$

From the result on (16), it can be deduced that the centered sample covariance matrix of the mapped data is K. Solving it, $\alpha^{(1)}, \alpha^{(2)}, ..., \alpha^{(n)}$ eigenvectors of $\frac{1}{n}K$ are obtained.

Performing the normalization condition of α eigenvectors, that is, forcing the eigenvectors $v^{(1)}, v^{(2)}, ..., v^{(n)}$ to be normalized in the feature space

$$1 = \langle v^{(l)}, v^{(l)}\rangle = \langle\sum_{i=1}^{n}\alpha_i^{(l)}\phi(x_i), \sum_{j=1}^{n}\alpha_j^{(l)}\phi(x_j)\rangle = \alpha^{(l)^T}K\alpha^{(l)}. \qquad (17)$$

Substituting K by the expression obtained in (16), it is obtained that $K = n\lambda$. As α eigenvectors have been computed from the matrix $\frac{1}{n}K$, their eigenvalues must be

expressed according to the fact that λ of K and $\frac{1}{n}K$ differs by a factor of $\frac{1}{n}$.

$$1 = \alpha^{(l)^T} \lambda \alpha^{(l)} = \lambda \|\alpha^{(l)}\|^2 \qquad \forall l = 1, ..., n. \tag{18}$$

From which one obtains,

$$\|\alpha^{(l)}\| = \frac{1}{\sqrt{\lambda}} \tag{19}$$

that in order to get the principal components in the feature space (kernel/feature principal components), eigenvectors α must be normalized by their corresponding eigenvalue λ.

- **Computing projections.** Supposing that $w_i \in \mathbb{R}^p$ for $i = 1, ..., m$ is a set of test points and their centered image in the feature space $\mathcal{H}$ is given by

$$K_{ij}^{c(test)} = K_{ij}^{(test)} - \frac{1}{n}\sum_{l=1}^{n} K_{il} - \frac{1}{n}\sum_{l=1}^{n} K_{jl}^{(test)} + \frac{1}{n^2}\sum_{l=1}^{n}\sum_{l'=1}^{n} K_{ll'} \tag{20}$$

whose general formulation,

$$K^{c(test)} = K^{(test)} - \frac{1}{n} J_{m \times n} K - \frac{1}{n} K^{(test)} J_{n \times n} + \frac{1}{n^2} J_{m \times n} K J_{n \times n}\,, \tag{21}$$

is obtained through an analogous evaluation like the one done in (9) . The projection of a point w_i into the l-th kernel principal component is:

$$f_l(w_i) = \langle v^{(l)}, \phi(w_i) \rangle = \langle \sum_{j=1}^{n} \alpha_j^{(l)} \phi(x_j), \phi(w_i) \rangle = \sum_{j=1}^{n} \alpha_j^{(l)} k(x_j, w_i) \tag{22}$$

Doing the same operation for each of the eigenvectors obtained before, the kPCA representation $f(w_i)$ is obtained.

Finally, the properties of kernel PCA are the following ones (assuming that the eigenvectors are sorted in ascending order of eigenvalue):

- All kernel principal components are pair-wise orthogonal, so uncorrelated.
- The first kernel principal components are the ones that explain the most part of the variance in the data.
- The mean-squared error in representing the observations with the first kernel principal components is minimal.
- The first kernel principal components have maximal mutual information with respect to the inputs.

V. RECONSTRUCTION FROM NON-LINEAR PRINCIPAL COMPONENTS [7]

Performing linear PCA one can completely reconstruct the data which it was originally used on with the complete set of principal components [8]. Using kPCA, one can also find the reconstruction of the points in the feature space $\mathcal{H}$. Indeed, we would need to reconstruct the image $\phi(x_i)$ in $\mathcal{H}$ of a point x_i from its projections β_k onto l non-linear components. So, we define a projection operator P_l such that

$$P_l\phi(x_i) = \sum_{k=1}^{l} \beta_k v^{(k)}. \tag{23}$$

It will happen that $P_l\phi(x_i) = \phi(x_i)$ if one uses all eigenvectors with non-zero associated eigenvalue. If not, $P_l\phi(x_i)$ will be an approximation that minimizes the squared reconstruction error. This approximation is a stepping-stone: it can be used to give a reconstruction of the pre-image in the input space of x_i. That is, a point r in the input space such that $\phi(r) = P_l\phi(x_i)$ which hopefully will be a good approximation of x_i. As this point r may not always exist, we approximate it by means of minimization of the squared reconstruction error

$$\begin{aligned} \rho(r) &= \|\phi(r) - P_l\phi(x_i)\|^2 \\ &= \|\phi(r)\|^2 - 2\,\phi(r)P_l\phi(x_i) + \gamma\,, \end{aligned} \tag{24}$$

where $\gamma = \|P_l\phi(x_i)\|^2$. Substituting (13) and (23) into (24) we arrive at an expression which does not depend on $\phi(\cdot)$:

$$\begin{aligned} \rho(r) &= \langle\, \phi(r), \phi(r)\,\rangle_{\mathcal{H}} - 2\sum_{k=1}^{l}\beta_k \sum_{i=1}^{n}\alpha_i \langle\, \phi(r), \phi(x_i)\,\rangle_{\mathcal{H}} + \gamma \\ &= k(r,r) - 2\sum_{k=1}^{l}\beta_k \sum_{i=1}^{n}\alpha_i k(r, x_i) + \gamma\,, \end{aligned} \tag{25}$$

which can be optimized with respect to r using methods such as standard gradient descent for differentiable kernel functions. For example, a straight forward use case of this technique can be de-noising: given a projection of a point (say, an image in $\mathbb{R}^p$) onto some principal components in the feature space, if we compute the approximated pre-image discarding principal components that account to noise, the result will be a de-noised version of the original point in the input space. This will be further explained in a following section.

VI. AN ALGORITHM FOR KERNEL PCA

Having explained all the necessary steps, we combine them in the following pseudo-code. First, how to compute principal components in a feature space:

```
function compute_principal_components_fs(data, kernel)
    n = num_of_observations(data)
    K = kernel_matrix(data)
    eigenvects = compute_eigenvects(K/n)
    eigenvals = compute_eigenvals(K/n)
    for each eigenvect i with eigenval > 0:
        pcs[i] = eigenvects[i]/sqrt(eigenvals[i])
    return pcs as principal_components
```

And how to project points, both from the training set and new observations, into the feature space principal components:

```
function project_points(data_to_project, principal_components, kernel,
                        training_data, kernel_matrix_tr_data)
    n = num_of_observations(data_to_project)
    m = num_of_observations(training_data)
    if data_to_project is training_data:
        gx = center(kernel_matrix_tr_data)
    else:
        gx = matrix(nrow = m, ncol = n)
        for each point i in data_to_project:
            for each point j in training_data:
                gx[i,j] = kernel(data_to_project[i], training_data[j])
        gx = center(gx, kernel_matrix_tr_data) # according to eq.(24)
    return matrix_multiplication(gx, principal_components)
```

We have coded an implementation of kPCA in the R language, which is attached to this document.

VII. COMPLEXITY ANALYSIS

For the purpose of analysing the complexity of kPCA we need to keep in mind some considerations about the algorithm described above. First, i) that we do not look for eigenvectors in the full space of $\mathcal{H}$, but just in the subspace spanned by the images of our samples in the feature space. This can be costly if there are many training samples, but not prohibitively expensive if we were to do it for i). Second, ii) we do not need to compute dot products explicitly between points in $\mathcal{H}$ as we are using kernel functions with the *kernel trick*.
We now analyse the **time complexity** in order of execution of the algorithm described in the previous section.

1. **"compute_principal_components_fs()"** First, we assume that evaluating the kernel function k for each training point $x_i, i = 1, ..., n$ in $\mathcal{R}^p$ is $O(p)$, as this is true for commonly used kernels (such as Gaussian RBF or polynomials). This, though, may be far from what other kernels cost. Computing the symmetric matrix $K_{n\,x\,n}$ (line 3) from the training data is then $O(p \cdot \frac{n(n+1)}{2}) \equiv O(p \cdot n^2)$. Then, we perform eigendecomposition of K (lines 4 and 5), which in practice approximately costs $O(n^3)$. Finally, we normalize all n (in the worst case) eigenvectors (line 6), which is $O(1)$ for each one, so $O(n)$. In total, this part is $O(p \cdot n^2) + O(n^3) + O(n) \equiv O(n^3)$ if $p < n$.

2. **"project_points()"** For the projection, we have to consider that if projecting the training points (line 4) we will have already computed the kernel matrix in the previous step. Also, centering the kernel matrix $K_{n\,x\,n}$ (line 5) can take (see implementation in R) $O(n^2)$ operations. If the m points in $\mathbb{R}^p$ to project are new (line 8), computing g_x takes $O(p \cdot m \cdot n)$ operations, and centering $O(m \cdot n) + O(n^2) \equiv O(max(m \cdot n, n^2))$. Finally, computing the projections (line 12) using a naive approach is $O(n^2 \cdot m)$ or $O(n^3)$ if projecting training points. So, the projection stage into all principal components in $\mathcal{H}$ is $O(n^2) + O(n^3) \equiv O(n^3)$ if we project training points or $O(p \cdot m \cdot n) + O(max(m \cdot n, n^2)) + O(n^3)$ if the projected points are new. In worst case and because usually $n > p$ and $n > m$, projection is $O(n^3)$, which can be reduced projecting into $l < n$ components.

In terms of **memory complexity**, the memory heavy hitters are: the kernel matrix K which is $O(n^2)$, the matrix g_x that takes $O(m \cdot n)$ space and the kernel principal component matrix, which is $O(n^2)$. Therefore, memory complexity is $O(max(n^2, n \cdot m))$ which is usually equivalent to $O(n^2)$.

We can then say that our implementation of kPCA is approximately $O(n^3)$ in time and $O(n^2)$ in memory. Due to its cost, both computationally and in terms of memory (if n is high), there have been studies that focus on reducing the cost, especially computational. They can be classified into two categories: approximative and iterative.

Firstly, **approximate** approaches [9] construct a low-rank approximation of the kernel matrix, computed from a subset of $m < n$ samples. In this way, the kernel matrix can be easily stored and manipulated. Their main limitation is that there's always a difference in their solution compared to the one obtained from the eigendecomposition of k directly. In particular, *Greedy kPCA* [10] creates an approximation of the training set with a simple algorithm from which features are extracted in the feature space, saving both memory and time.

Secondly, **iterative** approaches [11] use partial information of K in each round to estimate the top eigenvectors, so they do not keep the entire matrix in memory. Their main limitation is the lack of global convergence assurance.

VIII. PREVIOUS WORKS

Kernel PCA has been widely used since it was first introduced as a method for feature extraction and dimensionality reduction as a generalization of linear PCA. Some researchers have proved its usefulness for two different applications: Heiko Hoffmann for novelty detection [12] and Sebastian Mika, Bernhard Schölkopf, Alex J. Smola and Klaus-Robert Müller for image de-noising [7].

On the one hand, the first paper studies how to detect which samples differ from "ordinary" or "normal" data. Hoffmann used kernel PCA to model the distribution of training data and then compute the reconstruction error in the feature space (squared distance to the corresponding principal subspace) as a novelty measure. In addition, Hoffmann qualitatively compared kernel PCA with other novelty detection methods, such as one-class SVM, standard PCA or the Parzen [13] window density estimator. The new method he proposed demonstrated a competitive performance on different artificial and real data sets.

On the other hand, in the second paper, researchers found that kernel PCA achieves significantly better results than linear PCA in the field of image de-noising. Linear PCA extracted at most p components, being p the dimensionality of the training data, so all p components together fully describe the original data. Thus, if the data are noisy, the last components, those with smaller eigenvalue, would capture the structure of noise. In contrast, kernel PCA can extract up to n components, being n the number of training examples. In this way, kernel PCA provides more features that explain the structure in the data (as usually $n > p$) and takes more principal components to reconstruct noise.

Finally, we want to remark that there exists other generalizations of the linear PCA method to the nonlinear case. We are only going to mention them, as it is out of the scope of this paper. Most of them are explained in Diamantaras & Kung (1996) [14].

- Hebbian Networks: it consists on a number of unsupervised neural-network algorithms that compute principal components. Different variants of this algorithm can be obtained by adding nonlinear activation functions. However, this method do not have the geometrical interpretation that kernel PCA does, so it is more difficult to understand what exactly is doing.

- Autoassociative Multi-Layer Perceptron: the idea is to train a perceptron to reproduce the input values as output, having hidden layer with fewer neurons than the input. In this way, a lower-dimensional representation of the data is found. In addition, one can use nonlinear activation functions and additional layers and get a generalization of the problem.

- Principal Curves: this algorithm iteratively estimates some curves capturing the structure of the data. Then the data is mapped to the closest point on those curves, which have the property that each point on them is the average of all data points projected

onto each of them. As it happens, the only lines satisfying these last properties are principal components. To find the solution, a nonlinear optimization problem has to be solved.

IX. EXPERIMENTING

In this section, we present a set of experiments where we used kernel PCA to illustrate its performance compared to the linear version of PCA. We ran these experiments on synthetic and real-world data sets.

Artificial Data: we constructed three artificial data sets with two dimensions each. In order to graphically observe what, intuitively, kernel PCA is doing in the input space, we computed and represented the isolines of the first l feature principal components. These are contours representing the area where points in the input space have the same (approximately) value when projected onto whichever principal component in the feature space. This way one can see how well kernel PCA is interpreting the structure of our data. It can be seen that kPCA represents this structure way better than linear PCA, both because of its non-linearity and because the bigger quantity of principal components extracted. We also represented these isolines for the feature principal components in the case of a linear kernel to show that kPCA is a generalization of linear PCA.

Image segmentation: this dataset was created by the Vision Group in the University of Massachusetts and has been retrieved from UCI Machine Learning Repository. The instances were drawn randomly from a database of seven outdoor images and hand-segmented to create a classification for every 3x3 pixel region. It contains 19 continuous attributes and it is divided into 2100 samples for test data and 210 for training. Nonetheless, we would like to illustrate how with more principal components, kPCA can achieve a better rate than its linear counterpart, so the test set was used as training and vice-versa. The different categories into which each region falls are: brick-face, sky, foliage, cement, window, path, grass and are distributed as 300 instances per class for our training and 30 instances per class for our test.

FIG. 1: In this first experiment, we have created an artificial dataset consisting on three cluster. Each clusters was created from a 2 dimensional Gaussian with standard deviation 0.8, whose centers are equally spaced on a circle around the origin of radius 4. The first 8 kernel principal component, obtained from a RBF kernel (3) with $\sigma = 0.3$, are shown in order of decreasing eigenvalue size (left to right). One can realise that the first and second kernel principal components nicely separate the three clusters. Then, from the third to the fifth kernel PC divide the different clusters into halves. Similarly, the last components split them again orthogonal to the above splits.

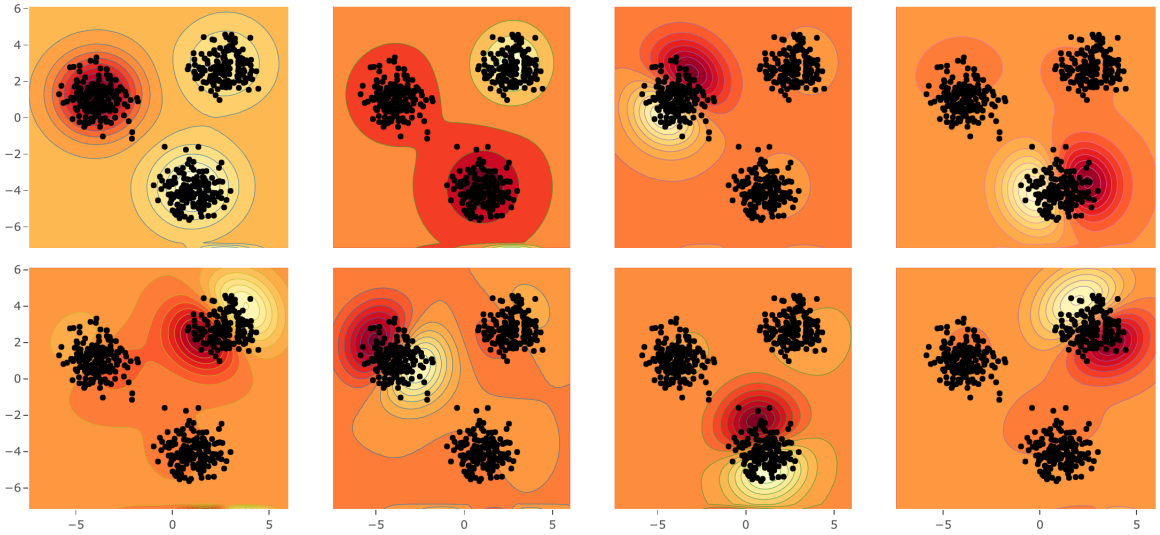


FIG. 2: In this second experiment, we created artificial data based on the inputs of the *cassini* problem, that is, data points uniformly distributed within 3 structures: two external with bean shape and in between them one with circular shape. With this dataset two different analysis have been performed: classic lineal PCA and using our implementation of kernel PCA with the vanilla kernel and the same kernel used in figure 1 with $\sigma = 0.2$. On the left, first and second principal components corresponding to the linear PCA are drawn overlayed on input data. At its right, the isolines of kernel PCA using vanilla kernel or linear kernel. In this case, the contour lines are orthogonal to the eigenvectors. Observe that both options are equivalent.

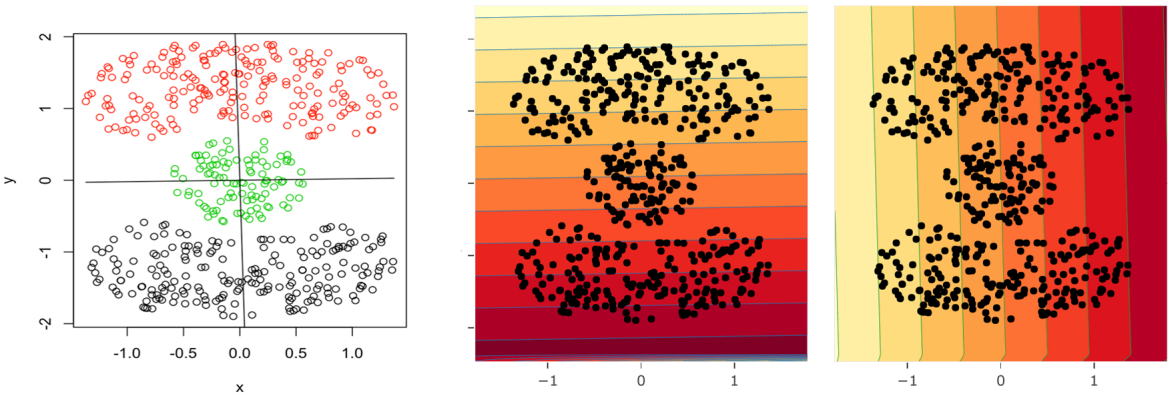


FIG. 3: These are the isolines for the first 8 principal components in the feature space of the "cassini problem", described in figure 2. We used a Gaussian RBF kernel (3) with an inverse kernel width of $\sigma = 0.2$. Notice how all components fragment the data symmetrically. The first one separates the 'beans' from the circle, the second divides the clusters vertically, the third one identifies the middle circle and the fourth splits the data in four. The rest of the components account for smaller and noisier structures.

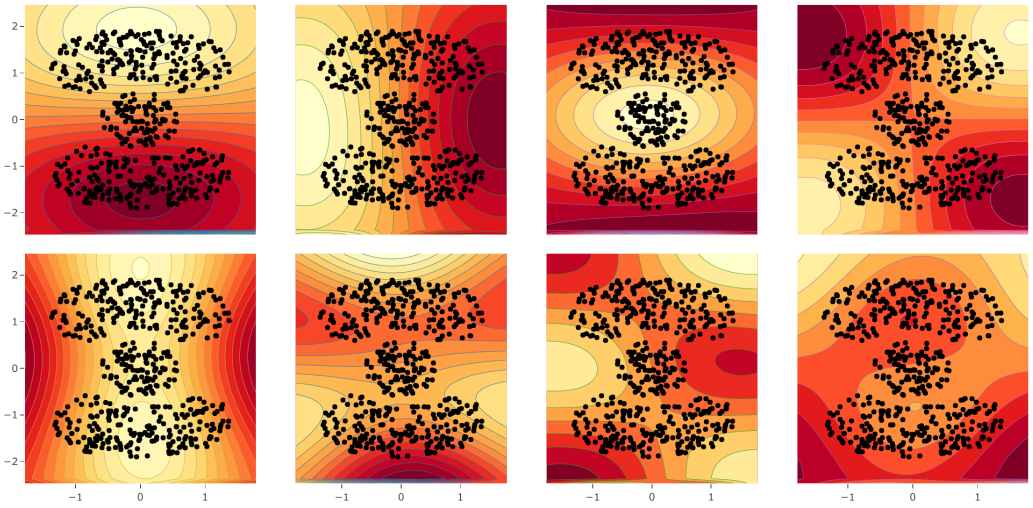


FIG. 4: In this third experiment, we have created artificial data consisting on two half moons: one sine wave and one cosine wave, both with additive Gaussian noise. Form left to right, we have represented: i) first and second linear principal components overlayed with input data, ii) data projected into the first and second kernel principal components (RBF kernel with $\sigma = 0.01$), iii) scree plot (value of eigenvalues) of the previous representation and iv) data projected into the first and second kernel principal components (RBF kernel with $\sigma = 2$). From these plots, one can deduce that although kernel PCA can explain almost perfectly the structure of the data in the feature space with few principal components (the "elbow" of the scree plot where eigenvalues seem to level off is at the second kernel PC), for some purposes, a different kernel configuration can separate better the data in the feature space. Also it's clear how kernel PCA achieves the sinusoidal shape of data while linear PCA sticks to its linear form.

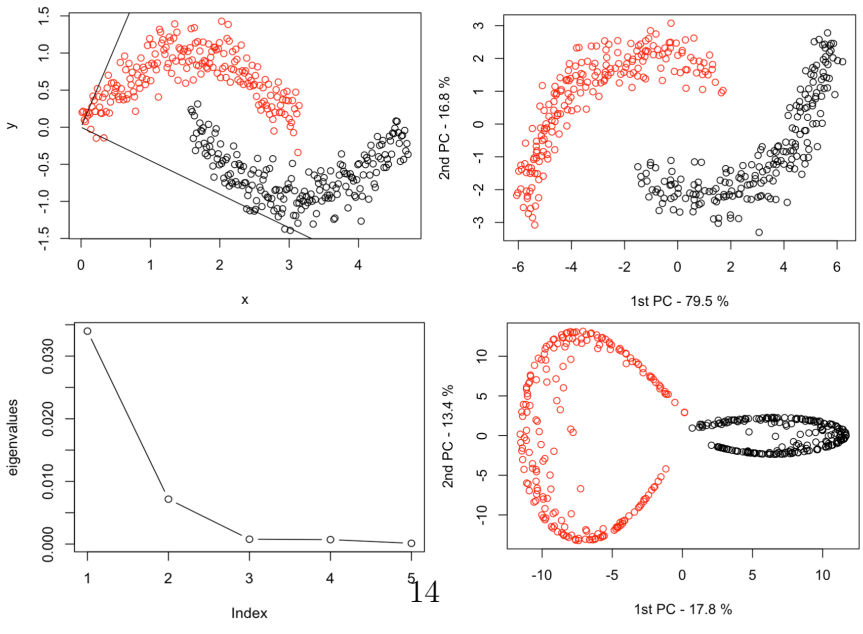
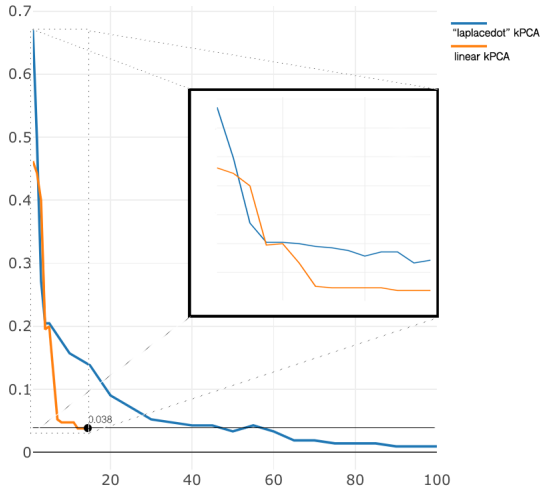


TABLE I: Results of experiments on pixel segmentation data. We tested doing feature extraction via kPCA first and then training a linear predictor (C-SVM, C = 1) on the feature projections (first four entries). For comparison, we also trained a non-linear C-SVM (C = 1, last entry) directly on the raw data, which does not perform as well as the other methods. We used several kernels for the first step: polynomial (degree 2, scale 1, offset 1), Gaussian RBF (σ = 0.01) and Laplace (σ = 0.01). See that both the first and last methods used can only use as many features as available in the raw data, while the others have more than 100. However, 50 are enough for the feature extractions with Laplace and polynomial kernels to beat PCA and kSVM in test error.

Method	Test error (all features)	Test error (50 features)
kPCA w/ lin. krnl	0.0380	NA
kPCA w/ poly. krnl	0.0095	0.0238
kPCA w/ RBF Gauss. krnl	0	0.4333
kPCA w/ Laplace krnl	0	0.0333
kSVM w/ Laplace krnl	0.0381	NA

FIG. 5: Test error of the method with kPCA (Laplace kernel, σ = 0.01) feature extraction plus linear C-SVM (C = 1) vs linear PCA plus linear C-SVM (C = 1). On the x axis are the number of principal components used. Notice how even though standard PCA beats the non-linear version for the first 14 dimensions, performance of the other method increases when more components are used: these components are capable of explaining nuances of the data, achieving very low error.



X. CONCLUSION

While nowadays there are many very effective machine learning techniques, it can be possible to gain some performance with a previous feature extraction. Kernel Principal Component analysis is a powerful non-liner feature extraction technique. With a previous knowledge of the problem in question, one can choose an appropriate kernel function which can provide very good interpretation of the data structure. kPCA can be applied to all areas where

linear PCA is used (and where non-linear feature extraction could make sense) with some advantages. First, it can identify non-linear patterns. It can also work with very high-dimensional data, as its cost only depends on n (the number of points) while the cost of linear PCA is related to p (the number of variables in the data). Moreover, it is capable of dealing with data not in $\mathbb{R}^p$ thanks to the flexibility of kernel functions and its performance can be further improved by using more components than possible in the linear case. Likewise, it is of interest that kPCA, compared to other nonlinear feature extraction methods, does not require (nonlinear) function optimization, just computing eigenvalues of an $n \times n$ matrix. With the advent of big data, however, massive amounts of information are being retrieved and analysed every day. This has created the necessity of modified versions of classical algorithms. In the case of kPCA, for example, we discussed several approximation and iterative methods, which allow for a lesser execution cost.
Overall, although kPCA was presented more than 20 years ago, a simple web search shows how it is still very much used in areas such as 3D face recognition [15] or road traffic analysis [16] and that it is a big part of active feature extraction research [17], [18].

---

[1] K. Pearson, *On lines and planes of closest fit to systems of points in space* (1901), vol. 2 of *6*, pp. 559–572.

[2] H. Hotelling, *Analysis of a Complex of Statistical Variables Into Principal Components* (1933), vol. 24, pp. 417–441.

[3] I. Jolliffe, *Principal Component Analysis* (Springer Berlin Heidelberg, Berlin, Heidelberg, 2011), pp. 1094–1096, ISBN 978-3-642-04898-2, URL `https://doi.org/10.1007/978-3-642-04898-2_455`.

[4] C. Cortes and V. Vapnik, Machine Learning **20**, 273 (1995), ISSN 1573-0565, URL `https://doi.org/10.1023/A:1022627411411`.

[5] N. Aronszajn, Transactions of the American Mathematical Society **68**, 337 (1950), ISSN 00029947, URL `http://www.jstor.org/stable/1990404`.

[6] L. Belanche, *Notes on AA2* (2019).

[7] S. Mika, B. Schölkopf, A. Smola, K.-R. Müller, M. Scholz, and G. Rätsch, pp. 536–542 (1998).

[8] B. Schölkopf, A. Smola, E. Smola, and K.-R. Müller, *Nonlinear Component Analysis as a Kernel Eigenvalue Problem* (1998), vol. 10, pp. 1299–1319.

[9] D. Lopez-Paz, S. Sra, A. Smola, Z. Ghahramani, and B. Schoelkopf, in *Proceedings of the 31st International Conference on Machine Learning*, edited by E. P. Xing and T. Jebara (PMLR, Bejing, China, 2014), vol. 32 of *Proceedings of Machine Learning Research*, pp. 1359–1367, URL `http://proceedings.mlr.press/v32/lopez-paz14.html`.

[10] V. Franc and V. Hlavac (2006), pp. 87–105.

[11] K. Kim, M. Franz, and B. Schölkopf, IEEE Transactions on Pattern Analysis and Machine

Intelligence **27**, 1351 (2005), ISSN 0162-8828.

[12] H. Hoffmann, Pattern Recognition **40**, 863 (2007), ISSN 0031-3203, URL http://www.sciencedirect.com/science/article/pii/S0031320306003414.

[13] E. Parzen, Ann. Math. Statist. **33**, 1065 (1962), URL https://doi.org/10.1214/aoms/1177704472.

[14] K. I. Diamantaras and S. Y. Kung, *Principal Component Neural Networks: Theory and Applications* (John Wiley & Sons, Inc., New York, NY, USA, 1996), ISBN 0-471-05436-4.

[15] M. Peter, J.-L. Minoi, and I. H. M. Hipiny, *3D Face Recognition using Kernel-based PCA Approach* (Springer, 2019), pp. 77–86.

[16] W. Yong-dong, X. Dong-wei, P. Peng, L. Yi, Z. Gui-jun, and X. Xiao, IET Intelligent Transport Systems (2019).

[17] C. Kim and D. Klabjan, IEEE transactions on pattern analysis and machine intelligence (2019).

[18] J. Chen and X. Li, Journal of Machine Learning Research **20**, 1 (2019).