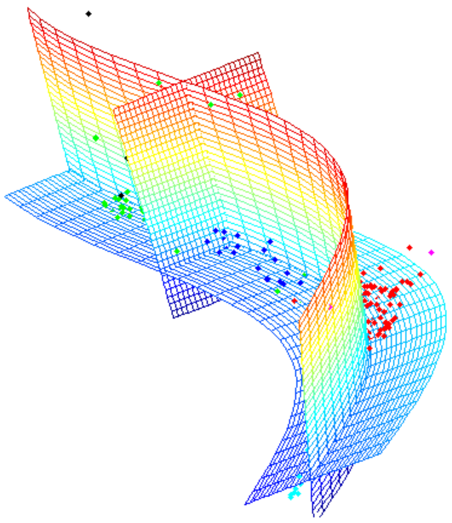
Kungliga Tekniska Högskolan
Valhallavägen 79
100 44 Stockholm

Machine Learning - Advanced Course
« Kernel Principal Component Analysis »

November 24th - December 17th



Authors
Thayabaran KATHIRESAN
Rémi DOMINGUES
Agnes Martine NIELSEN
Tobias WIENS

Teachers
Carl Henrik EK
Hedvig KJELLSTRÖM
Jens LAGERGREN

Scholar year 2014-2015

Contents

Introduction

Principal Component Analysis is a method which aims at reducing the dimensionality of a dataset into a linearly uncorrelated set of features, each maximizing the variance on the observations.

The method presented here gather PCA and kernel methods by describing an efficient way to compute principal components in a feature space of large dimensionality that is related to the input space by a non-linear mapping. An illustration of this can be seen in Figure 1. This is achieved by the use of kernel functions similar to the ones used in Support Vector Machines (SVM). This report details how to achieve this basis transformation and apply it through a feature extraction based on a digit recognition experiment.

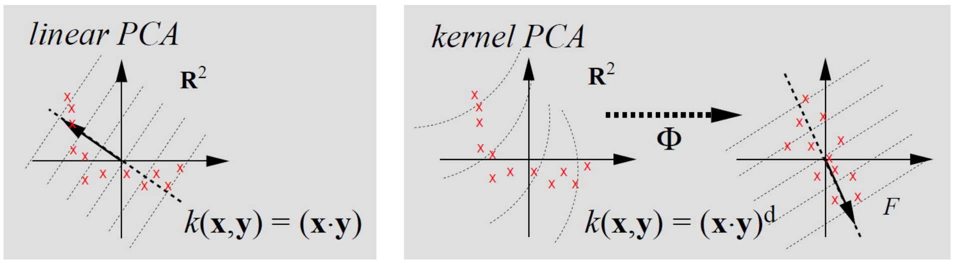


Figure 1: Comparison between linear PCA and kernel PCA

1 Theory

The following section describes the derivation of the method written in B. Schölkopf and A. J. Smola, *Kernel principal component analysis*[7].

The Kernel Principal Component Analysis allows us to find Principal Components which are nonlinearly related to the input space. Giving the possibility to extract and find relevant data having a nonlinear relation in the input space.

1.1 Concepts

1.1.1 Kernel

The kernels mentioned in this report refer to functions used to represent an inner product in a high-dimensional feature space which arise from a non-linear transformation from an input space. That means a kernel function k is,

$$k(\mathbf{x}, \mathbf{x'}) = (\Phi(\mathbf{x}) \cdot \Phi(\mathbf{x'})) \tag{1}$$

where $\Phi : \mathbb{R}^N \to F$ is a non-linear mapping from an input space $\mathbb{R}^N$ to a feature space F. $\mathbf{x}, \mathbf{x'} \in \mathbb{R}^N$, and $(\bullet \cdot \bullet)$ denotes the inner product in F.

The main interest of kernel functions resides in their computation cost, since they provide an inner product between each pair of points in the high dimension space for a lower complexity than the effective computation of the coordinates of the input points in the high dimension space followed by their pairwise inner product[1].

The efficiency of kernels when applied to Support Vector Machines has been demonstrated[6]. In the experiments, we will work with a polynomial kernel.

$$\text{Polynomial of degree } d\text{: } k(x,y) = (x \cdot y)^d \tag{2}$$

1.1.2 Principal Component Analysis (PCA)

The following section is based on [8].
PCA is a tool used to reduce the dimensionality of a dataset. When dealing with many input dimensions, the relevant data might be extracted using less dimensions, therefore we find a basis of orthogonal vectors in the input space, from which we know each vectors' contribution for the representing the data.
In order to find a lower dimensional representation of the data it requires to build the covariance matrix of the given dataset. Following, we extract the principal components by calculating the N eigenvectors corresponding the highest eigenvalues. The N is given by the user or deduced from Maximum Likelihood Estimation (MLE)[5]. In further detail, the eigenvectors with the highest eigenvalues are chosen, because those are the eigenvectors explain the highest amount of variance in the data.

The idea behind PCA is to "diagonalize the covariance matrix", in detail: we try to map our data into a new representation denoted by:

$$\mathbf{Y} = \mathbf{PX} \tag{3}$$

Where $\mathbf{X}$ is our data and $\mathbf{Y}$ is our new representation. The matrix $\mathbf{P}$ is the orthonormal projection matrix, transforming $\mathbf{X}$ into the representation $\mathbf{Y}$. The target of the PCA is to transform the data into a representation $\mathbf{Y}$ in which covariance matrix is a diagonal matrix. Thereby diagonalizing the covariance matrix. When we choose $\mathbf{P}$ to be a matrix where each row is an eigenvector of $\mathbf{X}$, the covariance matrix of $\mathbf{Y}$ becomes diagonal and the matrix $\mathbf{P}$ represents our principal components, for more details see [8].

1.2 Kernel Principal Component Analysis

1.2.1 Derivation

We define the high-dimension feature space F, containing the image of our data obtained after a non-linear mapping Φ from the input space

$$\Phi : \mathbb{R}^N \rightarrow F, \mathbf{x} \rightarrow \mathbf{X} \tag{4}$$

We first assume that our data in F is centered, $\sum\limits_{k=1}^{l} \Phi(\mathbf{x}_k) = 0$, i.e. the mean is 0.

As done in the standard PCA, the Kernel PCA algorithm constructs the covariance matrix of the data, but now we define it in the feature space F.

$$\bar{C} = \frac{1}{l}\sum_{j=1}^{l} \Phi(\mathbf{x}_j)\Phi(\mathbf{x}_j)^T \tag{5}$$

We now need to find the eigenvalues $\lambda \geq 0$ and eigenvectors $\mathbf{V} \in F\backslash\{0\}$ of the covariance matrix $\bar{C}$. That is, which satisfies $\lambda\mathbf{V} = \bar{C}\mathbf{V}$. The high dimensional eigenvectors cannot be computed since they might be infinite dimensional. But since all solutions $\mathbf{V}$ lie in span$\{\Phi(\mathbf{x}_1), \ldots, \Phi(\mathbf{x}_l)\}$ we can consider the equivalent system in (6), in which a data point $\Phi(\mathbf{x}_k)$ is projected onto the eigenvectors, and write the eigenvectors as in (7), where the data points are used as a basis.

$$\lambda(\Phi(\mathbf{x}_k) \cdot \mathbf{V}) = (\Phi(\mathbf{x}_k)) \cdot \bar{C}\mathbf{V}) \text{ for } k = 1, ..., l \tag{6}$$

$$\mathbf{V} = \sum_{i=1}^{l} \alpha_i \Phi(\mathbf{x}_i) \tag{7}$$

Let us now define a matrix $\mathbf{K} \in \mathbb{R}^{l \times l}$ that consists of inner products in the feature space F of the vectors $\Phi(\mathbf{x}_1), \ldots, \Phi(\mathbf{x}_l)$.

$$K_{ij} := (\Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)) \tag{8}$$

We now substitute (5) and (7) into (6),

$$\lambda(\Phi(\mathbf{x}_k) \cdot \sum_{i=1}^{l} \alpha_i \Phi(\mathbf{x}_i)) = (\Phi(\mathbf{x}_k) \cdot \frac{1}{l}\sum_{j=1}^{l} \Phi(\mathbf{x}_j)\Phi(\mathbf{x}_j)^T \sum_{i=1}^{l} \alpha_i \Phi(\mathbf{x}_i)) \text{ for } k = 1, ..., l$$

Rearranging this gives

$$l\lambda \sum_{i=1}^{l} \Phi(\mathbf{x}_k)^T \Phi(\mathbf{x}_i)\alpha_i = \sum_{j=1}^{l} \Phi(\mathbf{x}_k)^T \Phi(\mathbf{x}_j) \sum_{i=1}^{l} \Phi(\mathbf{x}_j)^T \Phi(\mathbf{x}_i)\alpha_i \text{ for } k = 1, ..., l$$

which can be written as follows with $\mathbf{K}$ defined above and $\boldsymbol{\alpha}$ is a column vector with entries $\alpha_1, \ldots, \alpha_l$,

$$l\lambda\mathbf{K}\boldsymbol{\alpha} = \mathbf{K}^2\boldsymbol{\alpha} \tag{9}$$

To find the solutions to (9) we solve the eigenvalue problem in (10) since all solutions to this problem are also solutions to (9). Because it can be shown that any additional solutions do not make a difference.

$$l\lambda\boldsymbol{\alpha} = \mathbf{K}\boldsymbol{\alpha} \tag{10}$$

The next step is to normalize the solutions $\boldsymbol{\alpha}^k$ corresponding to all eigenvalues, $\lambda > 0$. This is done by requiring that $(\mathbf{V}^k \cdot \mathbf{V}^k) = 1$, that is normalizing the eigenvectors in the feature space.

$$1 = (\mathbf{V}^k \cdot \mathbf{V}^k) = \sum_{i,j=1}^{l} \alpha_i^k \alpha_j^k (\Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)) = (\boldsymbol{\alpha}^k \cdot \mathbf{K}\boldsymbol{\alpha}^k) = \lambda_k (\boldsymbol{\alpha}^k \cdot \boldsymbol{\alpha}^k) \tag{11}$$

The principal components can be extracted by projecting a test data point $\Phi(\mathbf{x})$ onto the eigenvectors $\mathbf{V}^k$, according to equation (12).

$$(\mathbf{V}^k \cdot \Phi(\mathbf{x})) = \sum_{i=1}^{l} \alpha_i^k (\Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x})) \tag{12}$$

It can be observed from the previous derivation that the only operation applied to our data points in the feature space is a dot product in equations (8) and (12). This is precisely the low complexity operation provided by the kernel functions, which means that we do not need the mapping Φ explicitly. And in the above we can substitute all inner products in F with a kernel function. This will allow us to perform our calculations even if F is of infinite dimension.

1.2.2 Generalization

In order to simplify the derivation above, it has been previously assumed that the data has a mean of 0 in both input and feature space. However, we cannot generally center the data in the feature space, so the Kernel PCA algorithm should replace the use of the mapping Φ by

$$\tilde{\Phi}(\mathbf{x}_i) := \Phi(\mathbf{x}_i) - \frac{1}{l}\sum_{j=1}^{l} \Phi(\mathbf{x}_j) \tag{13}$$

Then we express the new matrix which we must diagonalize in terms of $\mathbf{K}$

$$\tilde{\mathbf{K}}_{ij} = \mathbf{K} - 1_l\mathbf{K} - \mathbf{K}1_l + 1_l\mathbf{K}1_l \text{ with } (1_l)_{ij} := \frac{1}{l} \tag{14}$$

2 Implementation

We have implemented the Kernel Principal Component Analysis method in Matlab. We have used the results from Kernel PCA in a classification problem using two different linear classifiers.

The algorithm which we have implemented is,

1. Compute the matrix $\mathbf{K}$ (8) using a kernel function

2. Solve equation (10) by diagonalizing $\mathbf{K}$
3. Solve equation (11) to compute the $\boldsymbol{\alpha}^k$ coefficients normalizing the corresponding eigenvectors
4. Project a test point onto the principal components

We have implemented the algorithm in two functions. One function that takes in a training data set and computes the matrix $\mathbf{K}$ using a polynomial kernel, finds the expansion coefficients $\boldsymbol{\alpha}^k$ and projects the training data onto the eigenvectors. It then returns $\boldsymbol{\alpha}^k$ as well as the projected training data. The second function takes in a test data set and projects it onto the eigenvectors extracted from the training data .

We then perform classification using the two sets of projected data, training and test data. We train the two classifiers on the projected training data and then test it in on the projected testing data. The first classifier used are the built in Matlab function `classify` which fits a Multivariate Gaussian to each of the clusters and that way create a linear classifier. The other classifier that we have used is a Support Vector Machine classifier implemented in LIBSVM [2] where we have used the linear implementation.

2.1 Experiment

We have evaluated the performance of our implementation of the Kernel PCA with a classification problem. We are considering the USPS (US Postal Service) handwritten digit dataset. It consists of 9300 observations each of 256 dimensions where each dimension is a pixel in a gray scale image of a handwritten digit of 16×16 pixels. Each observation is associated with a label describing the handwritten digit. The test set is made up by 2000 observations and here we use 3000 observation for our training set.

We have then extracted the principal components from the training and test data sets as described above and trained a separating hyperplane classifier on this projection of the training data. After that we have tested the classifier using the projected testing data.

We have used a polynomial kernel of degree, $d = 1, \dots, 6$, and extracted the first 2^n, $n = 6, 7, \dots, 11$, principal components.

In these experiments we have used two different kinds of linear classifier. The first we have used is the Matlab function `classify`. It fits a Multivariate Gaussian to each of the 10 clusters and creates the classification boundaries. The results using this classifier are shown in Figure 2. Here, we can see that for polynomial degree $d = 1$ the best classification performance is for 128 principal components with an error of 8.2%. This corresponds well with the results obtained in [7] where they get 8.6% error for this combination. Further, we see that in all other cases of 128 non-linear components, that is $d = 2, \dots, 6$, give a better performance with an error around 5 to 6%. Again this corresponds well with the results in the original article. We also see that extracting more principal components leads to even better performances for $d > 2$ and with 2048 principal components we get an error of about 2.5%. So generally the pattern that our results show corresponds well with the results in the original article.

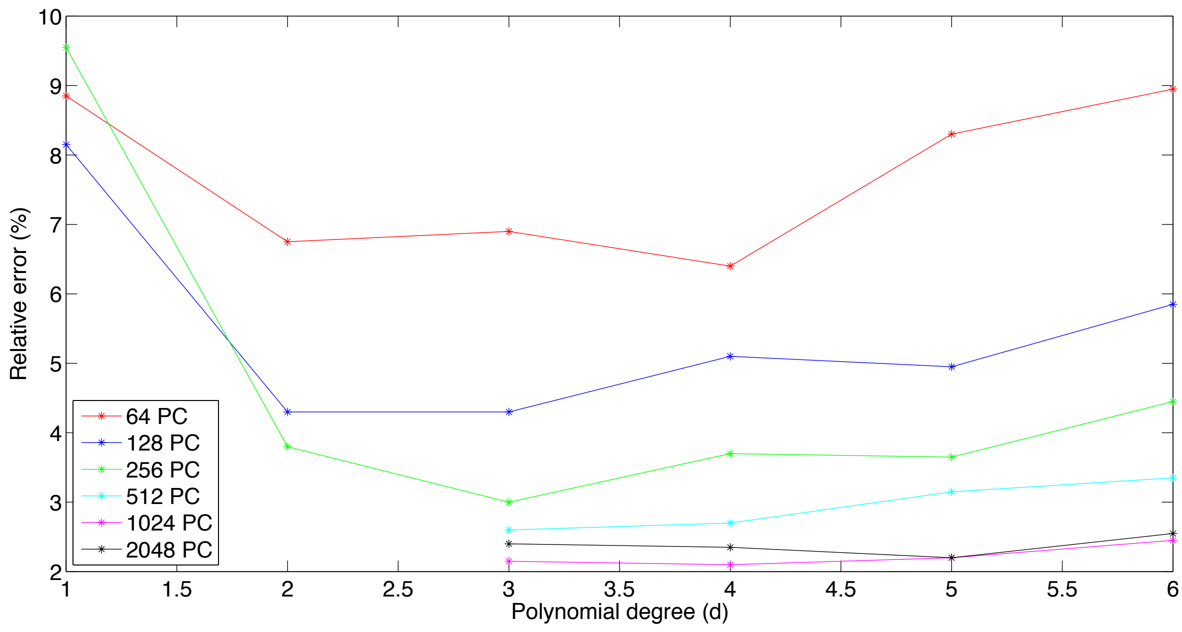


Figure 2: Classification errors using `classify` for different degrees of polynomial kernel and different numbers of principal components. The classification for a number of principal components higher than the dimension of the observations was only possible for polynomial degree $d > 2$.

However, in the article [7] they used a linear Support Vector Machine (SVM) classifier and not just any linear classifier. The results using linear SVM can be seen in Figure 3. Here we see that the best classification performance for linear PCA, $d = 1$, was reached for 512 principal components and that was 6.6%. The worst performance for $d = 1$ was with 64 principal components with an error of 7.3%. We notice that even the worst performance is better than the best performance we found for the previous classifier. If we then consider higher polynomial degrees $d > 1$, we notice that the best performance for all numbers of principal components is achieved at polynomial degree 3 or 4 depending on the number of principal components extracted and for degrees higher than that the error increases again. Overall the level of error corresponds well with the original article's results but we do not get the same pattern in the results as we got for the previous classifier. Also, since we do not know which type of multiclass SVM was used, one-vs-all or one-vs-one, whether a slack was used or not and how large and how compliant was the slack, we could not reproduce the exact results described in the paper.

Generally, we were able to recreate their experiment even though all our error rates do not correspond exactly to the ones obtained in the original article. This can be due to differences in the linear classifiers because of different parameters or simply improvements in the implementations of these classifiers since the article was published in 1997.

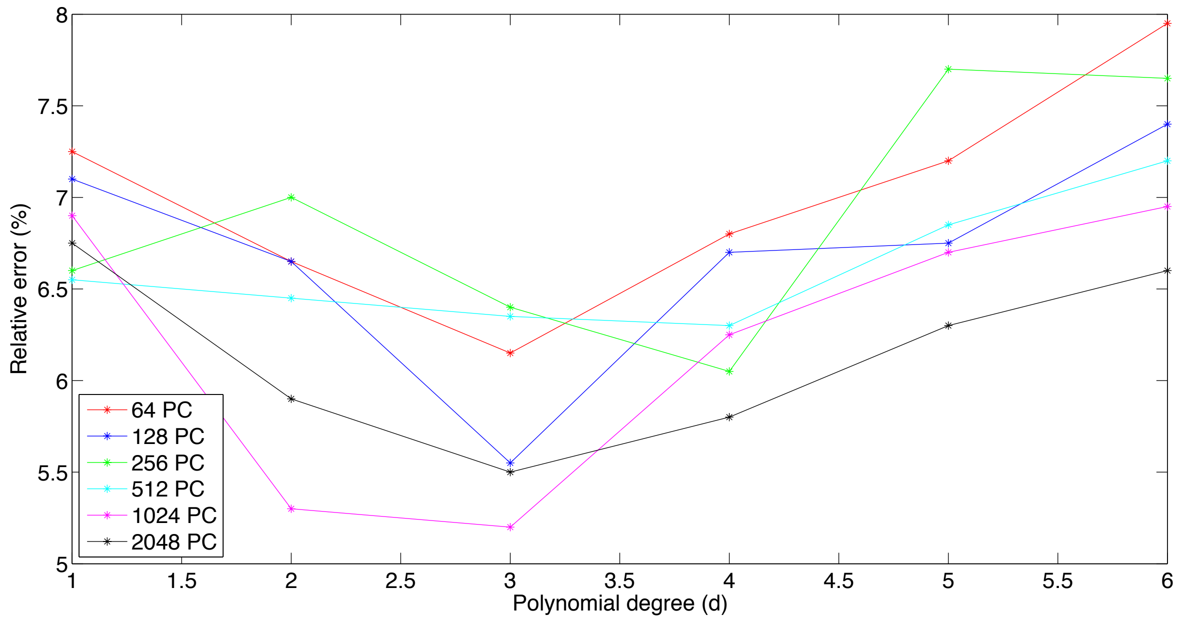


Figure 3: Classification errors using linear SVM classifier for different degrees of polynomial kernel and different numbers of principal components.

3 Discussion

Kernel PCA proposes an efficient way to compute principal components on a high-dimensional feature space that is related to the input space in a non-linear way. It does that by using a kernel to represent the dot product between two vectors in the feature space. This means that it can better describe non-linearities in the data. In our experiments non-linear Kernel PCA gave better results than standard linear PCA when extracting features and classifying using a linear classifier. This is advantage that could replicate from the original article. Another advantage is that we only need to solve an eigenvalue problem, as we do in standard PCA, and not use any non-linear optimization method. This is an advantage over other forms of non-linear PCA [7].

The paper introduces several different kernels that can be used in the method and have previously been successfully used in Support Vector Machines. This knowledge can be used in kernel PCA but also in any other algorithm that only requires the inner product between vectors. However, the article only describes results from experiments for a polynomial kernel function. It would be interesting to reproduce the experiment using a different kernel function, and see how well this algorithm performs. In general use a kernel that is specific to the problem in order obtain a better performance.

The kernel PCA is not suitable for large datasets as PCA is not, since the method is computationally expensive and we cannot fit our data using conventional data structures in RAM and an out-of-core computation is required [4]. It would be interesting to adapt current PCA algorithms that attempt to take care of these computational problem to a kernel PCA method.

Eventually, we have noticed that when we computed the covariance matrix, $\bar{C}$, in derivation of the method, then the scaling factor applied was l. Since the use of this factor would induce a bias [3], we propose to improve the Kernel PCA method by using the Bessel's correction and therefore using $l - 1$ as a scaling factor to remove the bias. This is relevant because the method exclusively rely on the variance. This would change the eigenvalue problem slightly and thereby the normalization of the eigenvectors.

References

[1] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152. ACM, 1992.

[2] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[3] Richard William Farebrother. *Fitting Linear Relationships: A History of the Calculus of Observations 1750-1900.* Springer, 1999.

[4] Nathan Halko, Per-Gunnar Martinsson, Yoel Shkolnisky, and Mark Tygert. An algorithm for the principal component analysis of large data sets. *SIAM Journal on Scientific computing*, 33(5):2580–2594, 2011.

[5] Thomas P Minka. Automatic choice of dimensionality for pca. In *NIPS*, volume 13, pages 598–604, 2000.

[6] Bernhard Schölkopf, Chris Burges, and Vladimir Vapnik. Incorporating invariances in support vector learning machines. In *Artificial Neural Networks—ICANN 96*, pages 47–52. Springer, 1996.

[7] Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. Kernel principal component analysis. In *Artificial Neural Networks—ICANN'97*, pages 583–588. Springer, 1997.

[8] Jonathon Shlens. A tutorial on principal component analysis. *arXiv preprint arXiv:1404.1100*, 2014.