CMSC878R PROJECT REPORT
Fast Kernel Principal Component Analysis for the Polynomial and Gaussian kernels

Vikas C. Raykar VIKAS@UMIACS.UMD.EDU

Ankur Ankur ANKUR@UMIACS.UMD.EDU

Perceptual Interfaces and Reality Labaratory
Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742

1. Summary

Title of the Project	Fast Kernel Principal Component Analysis for the Polynomial and Gaussian kernels
Student's Name	Vikas Chandrakant Raykar Ankur Ankur
Dimensionality of the Problem	Can be arbitrary..Implemented for 2 dimensions
Fast computational method used	Middleman Approach
Mother Function	$k(x, y) = (x.y + c)^d$ $k(x, y) = exp(-\frac{\|\|x-y\|\|^2}{2\sigma^2})$
Size of the problem	Arbitrary
Computational domain	Scaled to be a hypercube
Max allowed error	10^{-6}, absolute
Whether FMM is part of an iterative procedure	Yes for Eigen value Problem
What is the iterative method	Lanczos Iteration

2. Introduction

Kernel Principal Component Analysis(KPCA) is an attractive method for extracting nonlinear features from a given set of multi variate data[2][3]. While Principal Component Analysis(PCA) finds the best ellipsoidal fit for the data KPCA has the capability of extracting the nonlinear features which could be a more natural and compact representation of the data. KPCA finds applications in extracting features for a classification problem[2], image denoising[7], orthogonal series density estimation[5], clustering[4] etc. KPCA essentially performs a non linear mapping from the input space to some higher dimensional space. KPCA belongs to a class of more general methodology which use the *kernel trick*[2] for algorithms which can be written only in terms of dot products and not on the variable themselves. The *kernel trick* consists of replacing the dot product(which could be prohibitively expensive to compute) in some higher dimensional space by a kernel function which is more easy to compute. For N points in d dimensions whereas PCA can extract upto d principal components KPCA can give N components. One disadvantage of KPCA is that is is computationally very expensive. In KPCA we work with a $N \times N$ matrix called the Gram matrix. The two computationally intensive tasks are diagonalizing the Gram matrix and computation of the kernel principal components. Computing the eigen values and eigen vectors is of $O(N^3)$ complexity and computing the kernel principal components is of order $O(N^2)$ complexity. For most practical problems the data points generally represent training data and N can be very large. Different strategies have been reported for speeding up the KPCA. The different techniques involve sampling the Gram matrix[1], sparsifying the Gram matrix[1], using Kernel Principal Component Regression with the EM approach[9][8] etc. One thing to be noted is that in almost all cases the Gram matrix is essentially dense. In our approach we use the fact that in most cases all the eigen vectors are not needed. The number of eigen vectors needed is usually $<< N$. In such cases we can use iterative methods to compute the eigen vectors and the eigen values. Most commonly used methods include the Power iteration with Deflation, Orthogonal subspace iteration and the Krylov subspace methods[10]. Each method has its own advantages and disadvantages and is suited for different kinds of matrices. With iterative methods the computational complexity of finding the eigen vectors reduces to $O(N^2)$. The dominant factor is due to matrix vector multiplications. Also computing the kernel principal components is a matrix vector multiplication. It is this bottleneck of matrix vector multiplication which we solve in this paper. We reduce the complexity from $O(N^2)$ to $O(N)$ or $O(Nlog(N))$ depending on the kernel used. In this paper we discuss the Lanczos iteration[10] since the Gram matrix is symmetric. However it should be noted that the discussion applies equally well to other iterative methods which involve matrix vector multiplications. To this effect we use something called the *anti kernel trick* which is exactly opposite to what the kernel trick does. With the *kernel trick* we convert a possible infinite dot product into a kernel function. In our approach to reduce the complexity of matrix vector multiplication we expand the kernel function and by appropriate splitting of the indices we reduce the complexity. We propose fast algorithms for the most commonly used kernels the polynomial and the Gaussian kernel. For the polynomial kernel we give an exact method and for the Gaussian kernel we give a method with any desired accuracy and also establish tight error bounds. One more thing to be pointed out that the memory requirements for these methods are of the order N whereas for the normal method it is of the order of N^2 since in our method we do not explicitly form the matrix. These fast algorithms are in the spirit of the Fast Multipole Algorithms(FMM) reported in the literature.

The rest of the paper is organized as follows. Section 3 and 4 discuss the basic concepts of PCA and KPCA and Section 5 demonstrates the superiority of KPCA with a toy example. Section 6 discusses the computational complexity of KPCA. Section 7 reviews the iterative methods for solving dense eigen value problems. Section 8 introduces the most commonly used kernels and their properties. Section 9 introduces the notion of compress operator which will be used in subsequent sections. Section 10 and Section 11 develop fast algorithms for the polynomial and the Gaussian kernel respectively. Section 12 discusses the above sections in the light of a more practical example like face detection. Section 13 discusses further areas where such methods could be applied to reduce the computational complexity. Section 14 finally concludes.

3. Principal Component Analysis(PCA)

PCA is a statistical dimensionality reduction technique. Given N points in d dimensions PCA essentially projects the data points onto p, directions($p < d$)which capture the maximum variance of the data. These directions correspond to the eigen vectors of the covariance matrix of the training data points. Intuitively PCA fits an ellipsoid in d dimensions and uses the projections of the data points on the first p major axes of the ellipsoid.

PCA was first introduced by Harold Hotelling in the 1930's [6] though it was known much earlier in the field of psychology to Pearson[**]. Following is a brief derivation of PCA:

Let us say we have N data points in d dimensions with the mean subtracted. We would like to find c directions which capture the maximum variance of the data points. Then we can use the projection of the points on these c directions also known as the basis. The projections are called as principal components. Let α_{nm} be the projection of the nth data point on the mth basis e_m. Using these projections and the basis functions we can reconstruct a cth order approximation of x_n as

$$\widehat{x}_n = \sum_{m=1}^{c} \alpha_{nm} e_m \tag{1}$$

Define a new $d \times c$ matrix E where each column is a basis function and a vector α_m where the ith element is the projection on the ith basis.

$$E = [e_1|e_2|......e_c] \tag{2}$$

$$\alpha_n^T = [e_{n1}e_{n2}......e_{nc}] \tag{3}$$

Then $\widehat{x}_n$ can be written as

$$\widehat{x}_n = E\alpha_n \tag{4}$$

So now the problem can be stated as in terms of finding E and α_n which minimize the reconstruction error over all the data points subject to the constraint that $E^T E = I$ i.e. essentially we are looking at orthonormal directions.

$$\min_{E,\alpha_n} \frac{1}{2} \sum_{n=1}^{N} |x_n - E\alpha_n|^2 \tag{5}$$

subject to the constraint that $E^T E = I$. This function can be minimized by forming using the method of Lagrange multipliers.

$$J_L = \sum_n \frac{|x_n|^2}{2} + \frac{|\alpha_n|^2}{2} - x_n^T E\alpha_n + \frac{1}{2} Tr(E^T E - I)\Lambda \tag{6}$$

where Λ consists of the Lagrange multipliers

$$\Lambda = \begin{pmatrix} \lambda_1 & 0 & . & . & 0 \\ 0 & \lambda_2 & 0 & . & 0 \\ . & . & . & . & . \\ . & . & . & . & . \\ 0 & 0 & . & . & \lambda_c \end{pmatrix} \tag{7}$$

Differenciating with respect to α_k and E it can be shown that

$$\alpha_k = E^T x_k \tag{8}$$

and E is got by solving the eigen value problem

$$SE = E\Lambda \tag{9}$$

where S is the covariance matrix

$$S = \sum_{n=1}^{N} x_n x_n^T \tag{10}$$

The PCA algorithm can be summarized as below:

1. *Subtract the mean from all the data points* $x_n \leftarrow x_n - \frac{1}{N}\sum_{i=1}^{N} x_i$
2. *Compute the covariance matrix* $S = \sum_{n=1}^{N} x_n x_n^T$
3. *Diagonalize* S *to get its eigen values and eigen vectors i.e get* E *and* Λ.

4. *Retain c eigen vectors corresponding to c largest eigen values such that $\frac{\sum_{j=1}^{c}\lambda_j}{\sum_{j=1}^{N}\lambda_j}$ equals the desired variance to be captured.*
5. *Project the data points on the eigen vectors $\alpha_n = E^T(x_k - \frac{1}{N}\sum_{i=1}^{N} x_i)$ and use the projections instead of the data points.*

Solving the eigen value problem $SE = E\Lambda$ is of $O(d^3)$ complexity. However there are some applications where we have a few samples of very high dimensions. In such cases where $N < d$ we can formulate the problem in terms of an $N \times N$ matrix called as the dot matrix. In fact this is the formulation we will be using to develop KPCA in the next section. This method was first introduced by Kirby and Sirovitch in 1987[11] and also used by Turk and Pentland in 1991[12] for face recognition. The problem can be formulated as follows. Just for notational simplicity define a new $d \times N$ matrix X where each column correspond to the d dimensional data point i.e.

$$X = [x_1|x_2|......x_N] \tag{11}$$

According to our previous formulation for one eigen value we have to solve

$$\begin{aligned} Se &= \lambda e \\ XX^Te &= \lambda e \\ X^TXX^Te &= \lambda X^Te \\ K\alpha &= \lambda\alpha \end{aligned} \tag{12}$$

where $K = X^TX$ is called as the dot product matrix and $\alpha = X^Te$.

$$\begin{aligned} X^Te &= \alpha \\ XX^Te &= X\alpha \\ Se &= X\alpha \\ \lambda e &= X\alpha \\ e &= \frac{1}{\lambda}X\alpha \end{aligned} \tag{13}$$

Therefore e lies in the span of X the data points. This is a very important property which we will visit again in KPCA. Normalizing e

$$\begin{aligned} |e|^2 &= 1 \\ |X\alpha|^2 &= 1 \\ \alpha^TX^TX\alpha &= 1 \\ \alpha^TK\alpha &= 1 \\ \alpha^T\lambda\alpha &= 1 \\ |\alpha|^2 &= \frac{1}{\lambda} \end{aligned} \tag{14}$$

Finally the principal components can be written as where y is the projection of x on e

$$\begin{aligned} y &= e_Tx \\ y &= \alpha^TX^Tx \\ y &= \sum_{i}^{N}\alpha_i\langle x_i.x\rangle \end{aligned} \tag{15}$$

One thing to be noted is that we do not need e explicitly(we need only the dot product) and also to evaluate the principal components all the data points need to be retained.

PCA has very nice properties like it is the best linear representation in the MSE sense, first c principal components have more variance than any other c components etc. However PCA is still linear. It uses only second order statistics in the form of covariance matrix. The best it can do is to fit an ellipsoid around the data whereas KPCA which we will discuss in the next section has the capability of extracting the nonlinear features from the data.

4. Kernel Principal Component Analysis(KPCA)

The essential idea of KPCA is based on the hope that if we do some non linear mapping of the data points to a higher dimensional(possibly infinite) space we can get better non linear features which are a more natural and compact representation of the data. The computational complexity arising from the high dimensionality mapping is mitigated by using the *kernel trick*. Consider a nonlinear mapping $\phi(.) : \mathrm{R}^d \rightarrow \mathrm{R}^h$ from R^d the space of d dimensional data points to some higher dimensional space $\mathrm{R}^{h.}$. So now every point x_n is mapped to some point $\phi(x_n)$ in a higher dimensional space. Once we have this mapping KPCA is nothing but Linear PCA done on the points in the higher dimensional space.

In order to use the *kernel trick* PCA should be formulated in terms of the dot product matrix as discussed in the previous section. Also as of now assume that the mapped data are zero centered(i.e $\sum_{i=1}^{N} \phi(x_i) = 0$). Although this is not a valid assumption as of now it simplifies our discussion. Towards the end of this section we will show a more general approach[2]. So now in our case the dot product matrix K becomes

$$[K]_{ij} = [\phi(x_i).\phi(x_j)] \tag{16}$$

In literature K is usually referred to as the Gram matrix. Solving the eigen value problem $K\alpha = \lambda\alpha$ gives the corresponding N eigen vectors. Note that the eigen vector needs to be normalized to satisfy $|\alpha|^2 = \frac{1}{\lambda}$ Finally the principal component of any data point x, which is the projection of $\phi(x)$ on e is given by

$$y = \sum_{i}^{N} \alpha_i \langle \phi(x_i).\phi(x) \rangle \tag{17}$$

As mentioned earlier we do not need e explicitly(*we need only the dot product*). The *kernel trick* basically makes use of this fact and replaces the dot product by a kernel function which is more easy to compute than the dot product.

$$k(x, y) = \langle \phi(x).\phi(y) \rangle \tag{18}$$

This allows us to compute the dot product without having to carry out the mapping. The question of which kernel function corresponds to a dot product in some space is answered by the Mercer's theroem which says that k is a continous kernel of a positive integral operator then there exists some mapping where k will act as a dot product. The most commonly used kernels are the polynominal, Gaussian and the tanh kernel which will be discussed in more detail in Section 8.

Now we account for the more general case where the data points in the higher dimensional space need not be zero centered i.e.$\sum_{i=1}^{N} x_i = 0$ does not imply $\sum_{i=1}^{N} \phi(x_i) = 0$. From each of the data point in the higher dimensional space subtract the mean value. Now the centered point is

$$\tilde{\phi}(x_n) = \phi(x_n) - \frac{1}{N}\sum_{i=1}^{N} \phi(x_i) \tag{19}$$

Writing in matrix notation. Let

$$\begin{aligned}
\Phi &= [\phi(x_1)|\phi(x_1)|......|\phi(x_N)] \\
\tilde{\Phi} &= [\tilde{\phi}(x_1)|\tilde{\phi}(x_1)|......|\tilde{\phi}(x_N)] \\
\tilde{\Phi} &= \Phi - \Phi\frac{1_{N\times N}}{N} \\
\tilde{\Phi} &= \Phi(I - \frac{1_{N\times N}}{N})
\end{aligned} \tag{20}$$

where $1_{N\times N}$ is an $N \times N$ matrix with all entries equal to 1. The new Gram matrix is now

$$\begin{aligned}
\tilde{K} &= \tilde{\Phi}^T\tilde{\Phi} \\
\tilde{K} &= (I - \frac{1_{N\times N}}{N})^T K (I - \frac{1_{N\times N}}{N})
\end{aligned} \tag{21}$$

The principal components can now be written as

$$y = \sum_{i}^{N} \alpha_i \langle \phi(x_i).\phi(x) \rangle$$

$$y = \alpha^T \Phi^T \phi(x)$$

$$y = \alpha^T (I - \frac{1_{N\times N}}{N}) (\begin{bmatrix} k(x_1, x) \\ . \\ . \\ . \\ k(x_N, x) \end{bmatrix} - K \frac{1_{N\times 1}}{N}) \tag{22}$$

where $1_{N\times 1}$ is an $N \times 1$ matrix with all entries equal to 1.

The KPCA algorithm can be summarized as below:

1. *Given N data points in d dimensions let $X = [x_1|x_2|.....|x_N]$ where each column represents one data point.*
2. *Subtract the mean from all the data points.*
3. *Choose an appropriate kernel $k(.,.)$.*
4. *Form the $N \times N$ Gram matrix $[K]_{ij} = [k(x_i, x_j)]$.*
5. *Form the modified Gram matrix*
$$\tilde{K} = (I - \frac{1_{N\times N}}{N})^T K (I - \frac{1_{N\times N}}{N})$$
where where $1_{N\times N}$ is an $N \times N$ matrix with all entries equal to 1
6. *Diagonalize $\tilde{K}$ to get its eigen values λ_n and eigen vectors α_n.*
7. *Normalize $\alpha_n \Leftarrow \frac{\alpha_n}{\sqrt{\lambda_n}}$.*
8. *Retain c eigen vectors corresponding to c largest eigen values such that $\frac{\sum_{j=1}^{c} \lambda_j}{\sum_{j=1}^{N} \lambda_j}$ equals the desired variance to be captured.*
9. *Project the data points on the eigen vectors*
$$y = \alpha^T (I - \frac{1_{N\times N}}{N}) (\begin{bmatrix} k(x_1, x) \\ . \\ . \\ . \\ k(x_N, x) \end{bmatrix} - K \frac{1_{N\times 1}}{N})$$
where $1_{N\times 1}$ is an $N \times 1$ matrix with all entries equal to 1. and use the projections instead of the data points.

5. PCA vs KPCA using a Toy problem

In this section we consider a toy example in 2 dimensions where the N data points were constrained to lie in an annulus. In this case PCA will give two eigen vectors while KPCA will give N eigen vectors. Figure 1 shows the data points and the results from PCA and KPCA. In each plot we have shown the regions of equal principal components. The eigen vectors are perpendicular to the contour lines. It can be seen that PCA does an elliptical fit on the data so that the eigen vectors lie along the major and the minor axes of the ellipses. For the data a more natural axes would be one along the annulus and another perpendicular to it. It can be seen that the fourth and the fifth principal components extracted represent these axes. Figure 2 shows some more kernel principal components. In the above example we used the Gaussian kernel.

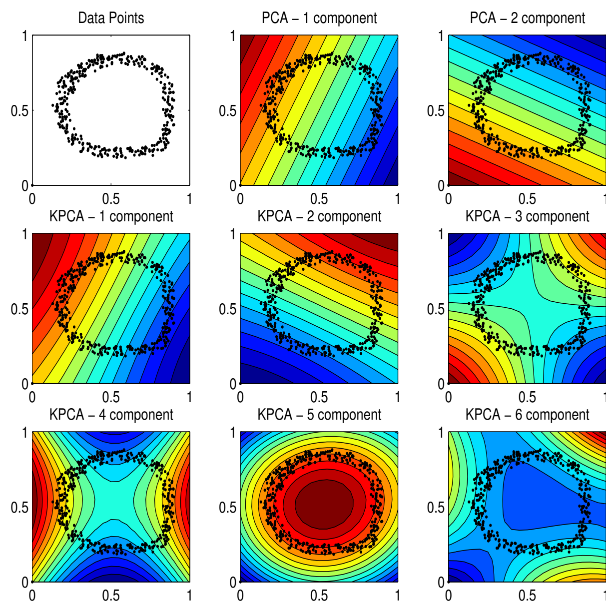

Figure 1. Two dimensional toy example illustrating PCA vs KPCA. For KPCA the Gaussian kernel was used. It can be seen that kernel principal components 4 and 5 are a more compact representation of the data than the principal components extracted by PCA

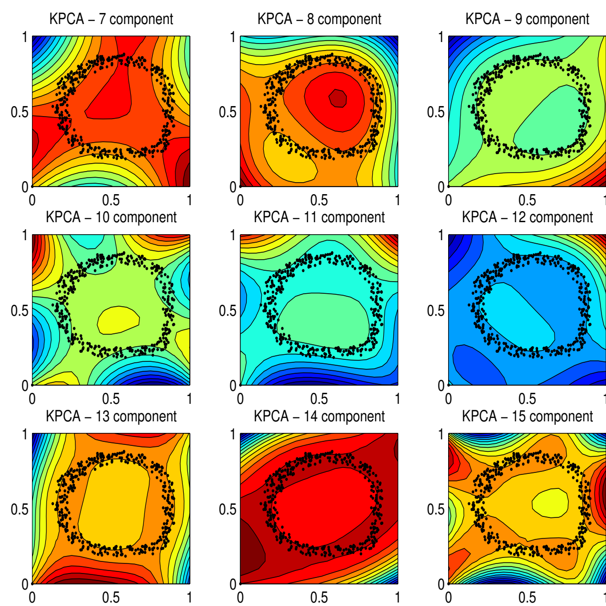

Figure 2. Two dimensional toy example illustrating PCA vs KPCA. For KPCA the Gaussian kernel was used.

6. Computational Complexity of KPCA

KPCA is very attractive for extracting the non-linear features but it comes at a higher computational cost than normal PCA. The two main computationally intensive tasks of KPCA are diagonalizing the modified $N \times N$ Gram matrix and computation of the kernel principal components. AS discussed in previous section the modified Gram matrix is

$$\tilde{K} = (I - \frac{1_{N\times N}}{N})^T K (I - \frac{1_{N\times N}}{N})$$

where $[K]_{ij} = [k(x_i, x_j)]$. This is an $N \times N$ matrix and finding the eigen values and eigen vectors is of $O(N^3)$ complexity. Also the memory storage is of $O(N^2)$. However in most applications all the eigen vectors are not needed. The number of eigen vectors needed is usually $<< N$. In such cases we can use iterative methods to compute the eigen vectors and the eigen values and the memory complexity becomes $O(N^2)$. The kernel principal components for any data x is given by

$$y = \alpha^T (I - \frac{1_{N\times N}}{N})(\begin{bmatrix} k(x_1, x) \\ . \\ . \\ . \\ k(x_N, x) \end{bmatrix} - K\frac{1_{N\times 1}}{N})$$

. The computational complexity is of order $O(N)$ for each point.So for N points the total complexity is of order $O(N^2)$. In both the cases the $O(N^2)$ dependency is due to matrix vector multiplication. In the next few sections we present algorithm to compute these matrix vector multiplication with complexity $O(N)$ or $O(Nlog(N))$.

7. Review of iterative methods for eigen value problems

For a given $N \times N$ matrix if we require only a few eigen values say $p << N$ then we can use the iterative methods. A good review of the different iterative methods for solving eigen value problems can be found in [10]. The important methods are as follows. We only list the methods. More details like convergence proofs can be seen in [10].

- *Power Iteration* This is a simple single vector iteration method which finds the dominant eigen value and the corresponding eigen vector by repeatedly multiplying the matrix by some non zero initial vector. This sequence of vectors when normalized converges to the dominant eigen vector. This method has a very slow convergence and will not converge if the dominant eigen value is complex.

- *Shifted Power method* In some cases instead of using the original matrix it is beneficial to use the shifted version which may give a faster convergence.

- *Inverse Iteration* If the smallest eigen value of the matrix is required rather than the largest.

- *Deflation Techniques* Once the dominant eigen value is found other eigen values can be found by deflation which effectively removes the eigen value found. The two important techniques are the Wielandt Deflation and the Schur Wielandt Deflation.

- *Subspace Iteration* Simplest method for computing many eigen values and eigen vectors is simultaneous iteration(also known as subspace iteration) which repeatedly multiples matrix times matrix of initial starting vectors. This this commonly used for large sparse matrices and can be seen as a block generalization of the power methods. Converges to the subspace determined by the p largest eigen values. Can use QR iteration here.

- *Krylov subspace methods* Reduce matrix to Hesenberg form using only matrix vector multiplication. These methods are based on the porjection methods onto Krylov subspaces. The three most important methods in this class are Arnoldi Iteration, Lanczos Iteration and the Jacobi Iteration.

The Gram matrix in our case is real symmetric. Real symmetric matrices are probably the *easiest* for these iterative methods. The most commonly used method for Hermitian matrices is the Lanczos algorithm which is

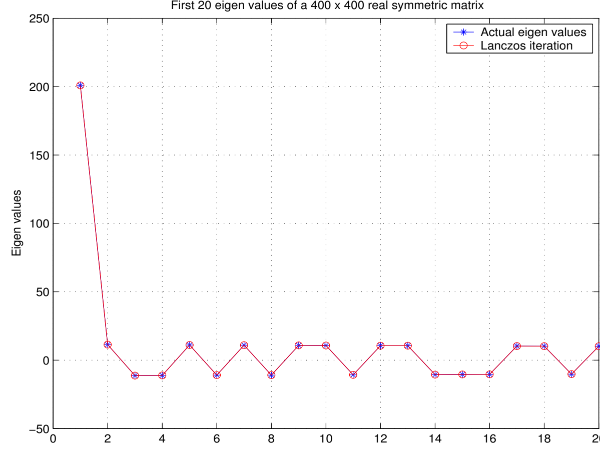

Figure 3. This figure shows the first 10 eigen values obtained by the Lanczos iteration as compared with the eigen values obtained by direct diagonalization using the *eig* function in MATLAB. The results are for a real symmetric 400 test matrix. It can be seen that the Lanczos iteration gives a good approximation to the actual eigen values.

basically a version of Arnoldi's iteration for Hermitian matricies. The Lanczos iteration is as follows(say we need k eigen values $k << N$ of matrix A)[10]:

Start Choose an initial vector v_1 of norm unity. Set $\beta_1 = 0$, $v_0 = 0$

Iterate for $j = 1, 2, ...k$ do

$$\begin{aligned} w_j &= Av_j - \beta_j v_{j-1} \\ \alpha_j &= (w_j, v_j) \\ w_j &= w_j - \alpha_j v_j \\ \beta_{j+1} &= ||w_j||_2 \\ v_{j+1} &= w_j/\beta_{j+1} \end{aligned} \tag{23}$$

α_j and β_j are the diagonal and the subdiagonal entries of symmetric tridiagonal matrix T_k. Lanczos method does not produce the eigen values directly but produces the tridiagonal matrix T_k whos eigen values and eigen vectors are to the determined by some other method. In general if only a few eigen values are needed T_k is a very small matrix as compared to the original matrix and can be diagonalized directly. If the Lanczos algorithm were run till $k = N$ then the resulting tridiagonal matrix would be orthogonally similar to A. The above algorithm gurantees that the vectors v_i are orthogonal. However in reality exact orthogonality of these vectors is observed only at the beginning of the process[10]. Also rounding error causes loss of orthogonality. The problem can be overcome by reorthogonalizing the vectors as needed or do some partial or selective orthogonalization. Figure 3 shows the first 20 eigen values obtained by the Lanczos iteration using the *eigs* function in MATLAB as compared with the eigen values obtained by direct diagonalization using the *eig* function in MATLAB. The results are for a real symmetric 400 test matrix. It can be seen that the Lanczos iteration gives a good approximation to the actual eigen values.

Lanczos iteration provides a good approximation for the extreme eigen values and they require only one matrix-vector multiplication per step. In the next few sections we show that for matrix which involve the polynomial and the Gaussian kernel the matrix-vector multiplication can be done with $O(N)$ or $O(Nlog(N))$. Also most of the commercially available software have of the option of providing the function which does the matrix-vector product. So we can write a fast efficient matrix-vector multiplication code which can effectively fasten up the iterative procedure.

8. Different kernels and their properties

The three most popular kernels are the Polynomial, Gaussian and the tanh kernel. The polynomial kernel of degree d is given by

$$k(x,y) = (x.y + c)^d \tag{24}$$

where c is a constant whose choice depends on the range of the data. It can be seen that the polynomial kernel picks up correlations upto order d.

The Gaussian or the radial basis function kernel is given by

$$k(x,y) = exp(-\frac{||x-y||^2}{2\sigma^2}) \tag{25}$$

where the parameter σ controls the support region of the kernel.

The tanh kernel which is mostly used in Neural network type application is given by

$$k(x,y) = \tanh((x.y) + b) \tag{26}$$

This kernel is not actually positive definite and is not used in practice.

As can be seen from previous sections the most important part is the matrix vector multiplication where each entry of the matrix is of the form $[K]_{ij} = [k(x_i, x_j)]$ where the kernel function can be one of the above.

9. Compress operator

In this section we define an operator called the compression operator which is very crucial for the development of fast methods. Let a be d dimensional vector and define an operator called $Compress(a^n)$. We require the property that

$$a^n \cdot b^n = Compress\left(a^n\right) \cdot Compress\left(b^n\right) \tag{27}$$

Consider in two dimensions

$$a^n \cdot b^n = \left(a \cdot b\right)^n = \left(a_1 b_1 + a_2 b_2\right)^n = a_1^n b_1^n + \begin{pmatrix} n \\ 1 \end{pmatrix} a_1^{n-1} b_1^{n-1} a_2 b_2 + \begin{pmatrix} n \\ 2 \end{pmatrix} a_1^{n-2} b_1^{n-2} a_2^2 b_2^2 + ... + a_2^n b_2^n$$

Now Let us define

$$Compress\left(a^n\right) = \left(a_1^n, \sqrt{\begin{pmatrix} n \\ 1 \end{pmatrix}} a_1^{n-1} a_2, \sqrt{\begin{pmatrix} n \\ 2 \end{pmatrix}} a_1^{n-2} a_2^2, ..., a_2^n\right)$$

$$Compress\left(b^n\right) = \left(b_1^n, \sqrt{\begin{pmatrix} n \\ 1 \end{pmatrix}} b_1^{n-1} b_2, \sqrt{\begin{pmatrix} n \\ 2 \end{pmatrix}} b_1^{n-2} b_2^2, ..., b_2^n\right)$$

Note that the length of the compressed vector is $n+1$ and not 2^n. Similarly we can define the compression operator in 3D

$$\begin{aligned}
a^n \cdot b^n &= \left(a \cdot b\right)^n = \left(a_1 b_1 + a_2 b_2 + a_3 b_3\right)^n \\
&= \left[\left(a_1 b_1 + a_2 b_2\right) + a_3 b_3\right]^n = \sum_{m=0}^{n} \begin{pmatrix} n \\ m \end{pmatrix} \left(a_1 b_1 + a_2 b_2\right)^{n-m} a_3^m b_3^m \\
&= \sum_{m=0}^{n} \sum_{l=0}^{n-m} \begin{pmatrix} n \\ m \end{pmatrix} \begin{pmatrix} n-m \\ l \end{pmatrix} a_1^{n-m-l} b_1^{n-m-l} a_2^l b_2^l a_3^m b_3^m \\
&= a_1^n b_1^n + \begin{pmatrix} n \\ 1 \end{pmatrix} a_1^{n-1} b_1^{n-1} a_2 b_2 + \begin{pmatrix} n \\ 2 \end{pmatrix} a_1^{n-2} b_1^{n-2} a_2^2 b_2^2 + ... + a_2^n b_2^n \\
&+ \begin{pmatrix} n \\ 1 \end{pmatrix} a_1^{n-1} b_1^{n-1} a_3 b_3 + \begin{pmatrix} n \\ 1 \end{pmatrix} \begin{pmatrix} n-1 \\ 1 \end{pmatrix} a_1^{n-2} b_1^{n-2} a_2 b_2 a_3 b_3 + ... + a_3^n b_3^n,
\end{aligned}$$

$$Compress\left(a^n\right) = \left(a_1^n, \sqrt{\begin{pmatrix} n \\ 1 \end{pmatrix}} a_1^{n-1} a_2, \sqrt{\begin{pmatrix} n \\ 2 \end{pmatrix}} a_1^{n-2} a_2^2, ..., a_2^n, \sqrt{\begin{pmatrix} n \\ 1 \end{pmatrix}} a_1^{n-1} a_3, ..., a_3^n\right)$$

For any dimension d the compression operator can be defined as

$$(a_1 + a_2 + ... + a_d)^n = \sum_{n_1+n_2+..+n_d=n} (n; n_1, n_2...n_d) a_1^{n_1} a_2^{n_2} ....a_d^{n_d} \tag{28}$$

where $(n; n_1, n_2...n_d)$ is the multinomial coefficient

$$(n; n_1, n_2...n_d) = \frac{n!}{n_1!n_2!....n_d!} \tag{29}$$

$$Compress(a^n) = (a_1^n, \sqrt{(n; n-1, 1, 0, ..., 0)} a_1^{n_1} a_2, ..., \sqrt{(n; n_1, n_2...n_d)} a_1^{n_1} a_2^{n_2} ....a_d^{n_d}, ..., a_d^n) \tag{30}$$

It can be proved by induction that in general for d dimensions the length of the compressed vector is $(\frac{n+d-1}{n}$.

10. Fast Algorithm for polynomial kernel

In this section we develop a fast algorithm for evaluating the matrix vector multiplication $v = Ku$ where $[K]_{ij} = [k(x_i, x_j)]$ for the polynomial kernel of order p $k(x, y) = (x.y + c)^p$ i.e we want to evaluate v_i for $i = 1.....N$

$$v_i = \sum_{j=1}^{N} k(x_i, x_j) u_j$$

$$v_i = \sum_{j=1}^{N} (x_i.x_j + c)^p u_j \tag{31}$$

Using the Newton's Binomial theorem we can write

$$v_i = \sum_{j=1}^{N} u_j \sum_{k=0}^{p} \begin{pmatrix} p \\ k \end{pmatrix} (x_i.x_j)^k c^{p-k}$$

$$v_i = \sum_{k=0}^{p} \begin{pmatrix} p \\ k \end{pmatrix} c^{p-k} \sum_{j=1}^{N} u_j (x_i.x_j)^k \tag{32}$$

Using the compress operator discussed in the previous section we can write

$$v_i = \sum_{k=0}^{p} \begin{pmatrix} p \\ k \end{pmatrix} c^{p-k} \sum_{j=1}^{N} u_j comp(x_i^k).comp(x_j^k)$$

$$v_i = \sum_{k=0}^{p} comp(x_i^k).[\sum_{j=1}^{N} u_j \begin{pmatrix} p \\ k \end{pmatrix} c^{p-k} comp(x_j^k)]$$

$$v_i = \sum_{k=0}^{p} comp(x_i^k).A_k \tag{33}$$

where

$$A_k = \sum_{j=1}^{N} u_j \begin{pmatrix} p \\ k \end{pmatrix} c^{p-k} comp(x_j^k) \tag{34}$$

The A_ks depend only on j and not on the index i. So all the A_k can be computed in one loop for all k and then we can use these to compute v_i for all n. Computation of each A_k involves the summation over N terms. Let each term which is in the summation for computing A_k need P operations. Then each A_k computation is of the order $O(PN)$. There are $p + 1$ such A_k's. So the total complexity is $O(P(p + 1)N)$. computing v_i is of $O(P(p+1)N)$. Therefore the total complexity is $O(2(p+1)PN)$. Generally p is small so the complexity is of the order $O(PN)$ while the straightforward computation is of $O(N^2)$. Figure 4 shows the CPU time as measured in

MATLAB for $d = 2$ for N varying from 100 to 1000. The degree of the polynomial kernel $p = 2$ and $c = 1$. In this formulation there were no approximations and ideally one would expect that there would be no error between the straightforward and the fast method, But however due to numerical error we do have some error.Figure 5 shows the maximum absolute error between the straightforward and the fast method for N varying between 10^2 and 10^3 for the same case.

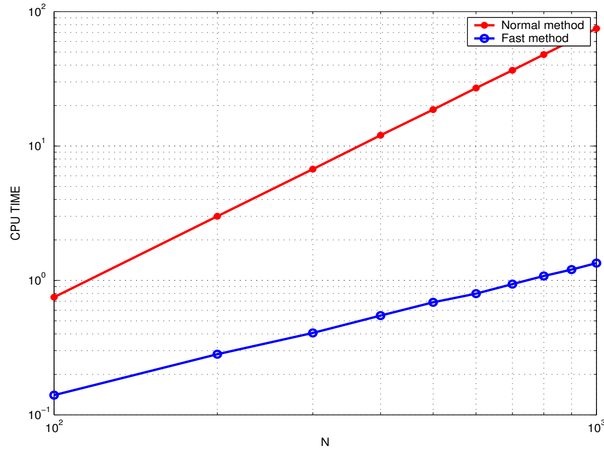

Figure 4. CPU time as measured in MATLAB for $d = 2$ for N varying from 100 to 1000. The degree of the polynomial kernel $p = 2$ and $c = 1$.

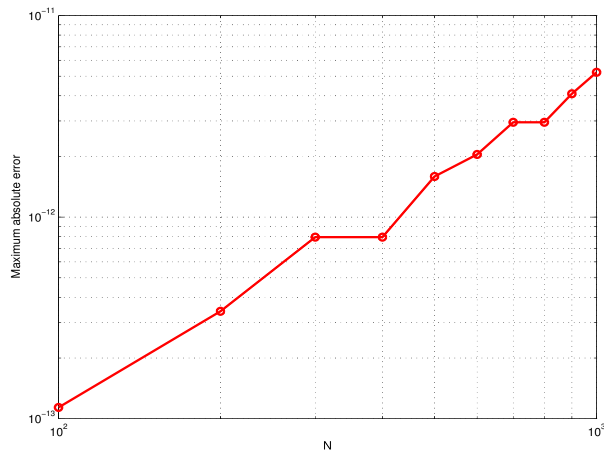

Figure 5. The maximum absolute error between the straightforward and the fast method for N varying between 10^2 and 10^3.The degree of the polynomial kernel $p = 2$ and $c = 1$.

Figure 6 shows the CPU time as measured in MATLAB for $d = 2$ for N varying from 100 to 1000 for different polynomial degree p. It can be seen that as p increases the breakeven point increases. Also one more thing to be noted is that the fast method is fast only if P is the number of operations required to compute each term in A_k is very less than N. This could be a problem in higher dimensions where the compression operator involves very high factorials and computing them would require a very large number of operations.[TO BE DISCUSSED IN MORE DETAIL].

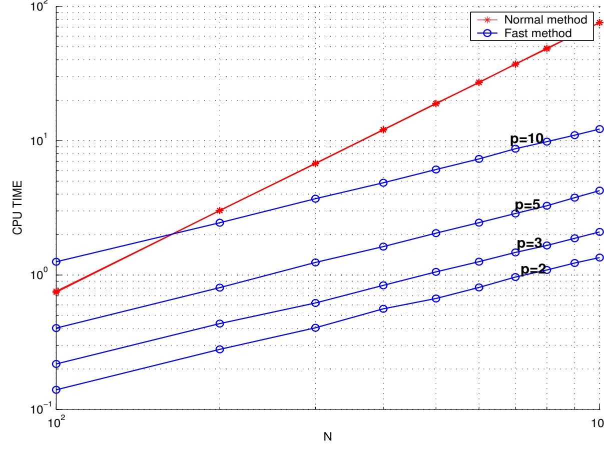

Figure 6. CPU time as measured in MATLAB for $d = 2$ for N varying from 100 to 1000 for different polynomial degree p.

11. Fast Algorithm for Gaussian kernel

n this section we develop a fast algorithm for evaluating the matrix vector multiplication $v = Ku$ where $[K]_{ij} = [k(x_i, x_j)]$ for the Gaussian kernel $k(x, y) = exp(-\frac{||x-y||^2}{2\sigma^2})$ i.e we want to evaluate v_i for $i = 1.....N$

$$v_i = \sum_{j=1}^{N} k(x_i, x_j)u_j$$

$$v_i = \sum_{j=1}^{N} exp(-\frac{||x_i - x_j||^2}{2\sigma^2})u_j \tag{35}$$

Let us consider the Taylor series expansion of $k(x_i, x_j)$ around x_*.

$$K(x_i, x_j) = \exp(\tfrac{-|x_i-x_j|^2}{2\sigma^2}) \qquad i = 1 \cdots N$$

$$= \exp(\tfrac{-|x_i-x_*+x_*-x_j|^2}{2\sigma^2})$$

$$= \exp(\tfrac{-|(x_i-x_*)-(x_j-x_*)|^2}{2\sigma^2})$$

$$= \exp\left(\tfrac{-|x_i-x_*|^2}{2\sigma^2}\right) \exp\left(\tfrac{-|x_j-x_*|^2}{2\sigma^2}\right) \exp\left(\tfrac{1}{\sigma^2}\left(x_i - x_*\right).\left(x_j - x_*\right)\right)$$

$$= \exp\left(\tfrac{-|x_i-x_*|^2}{2\sigma^2}\right) \exp\left(\tfrac{-|x_j-x_*|^2}{2\sigma^2}\right) \textstyle\sum_{m=0}^{\infty} \frac{1}{\sigma^{2m}m!} \left(x_i - x_*\right)^m \cdot \left(x_j - x_*\right)^m$$

truncating upto p terms,

$$= \exp\left(\tfrac{-|x_i-x_*|^2}{2\sigma^2}\right) \exp\left(\tfrac{-|x_j-x_*|^2}{2\sigma^2}\right) [\textstyle\sum_{m=0}^{p-1} \frac{1}{\sigma^{2m}m!} \left(x_i - x_*\right)^m \cdot \left(x_j - x_*\right)^m + error_p]$$

Using the compress operator

$$=\exp\left(\tfrac{-|x_i-x_*|^2}{2\sigma^2}\right) \exp\left(\tfrac{-|x_j-x_*|^2}{2\sigma^2}\right) [\textstyle\sum_{m=0}^{p-1} \frac{1}{\sigma^{2m}m!} Compress((x_i - x_*)^m) \cdot Compress((x_j - x_*)^m) + error_p]$$

The error due to truncation can be evaluated as..

$$|error_p| \le \exp\left(\tfrac{-|x_i-x_*|^2}{2\sigma^2}\right) \exp\left(\tfrac{-|x_j-x_*|^2}{2\sigma^2}\right) \tfrac{1}{p!} \sup_{x_j} \left|((x_j - x_*).\nabla)^p \exp\left(\tfrac{1}{\sigma^2}\left(x_i - x_*\right).\left(x_j - x_*\right)\right)\right|$$

Since $\exp\left(\frac{-|x_i-x_*|^2}{2\sigma^2}\right) \le 1, \exp\left(\frac{-|x_i-x_*|^2}{2\sigma^2}\right) \le 1,$

$|error_p| \leq \frac{1}{p!}\sup_{x_j}\left|((x_j - x_*).\nabla)^p \exp\left(\frac{1}{\sigma^2}(x_i - x_*).(x_j - x_*)\right)\right|$

$\quad = \frac{|x_j - x_*|^p |x_i - x_*|^p}{\sigma^{2p} p!}\sup_{x_j}\left|\exp\left(\frac{1}{\sigma^2}(x_i - x_*).(x_j - x_*)\right)\right|$

since $|(x_i - x_*)\cdot(x_j - x_*)| \leq |(x_i - x_*)||(x_j - x_*)|$

$\quad \leq \frac{|x_j - x_*|^p |x_i - x_*|^p}{\sigma^{2p} p!}\sup_{x_j}\exp\left(\frac{1}{\sigma^2}|(x_i - x_*)||(x_j - x_*)|\right)$

Assuming all the data are scaled to lie in a d dimensional hypercube and x_* is chosen to lie in the center of the hyper cuber then

$|x_i - x_*|^p \leq \left(\frac{\sqrt{d}}{2}\right)^p |x_j - x_*|^p \leq \left(\frac{\sqrt{d}}{2}\right)^p$

$|error_p| \leq \frac{|x_j - x_*|^p |x_i - x_*|^p}{\sigma^{2p} p!}\sup_{x_j}\exp\left(\frac{1}{\sigma^2}|(x_i - x_*)||(x_j - x_*)|\right)$

$|error_p| \leq \frac{\left(\frac{\sqrt{d}}{2}\right)^p\left(\frac{\sqrt{d}}{2}\right)^p}{\sigma^{2p} p!}\exp\left(\frac{1}{\sigma^2}\left(\frac{\sqrt{d}}{2}\right)\left(\frac{\sqrt{d}}{2}\right)\right)$

$|error_p| \leq \frac{\left(\frac{d}{4}\right)^p}{\sigma^{2p} p!}\exp\left(\frac{d}{4\sigma^2}\right)$

The absolute error in computation does not exceed $|error_p|$.

For fast summation,

$v_i = \sum_{j=1}^{N} K(x_i, x_j)\, u_j$

$v_i = \sum_{j=1}^{N} \exp\left(\frac{-|x_i - x_j|^2}{2\sigma^2}\right) u_j$

$v_i = \sum_{j=1}^{N} \exp\left(\frac{-|x_i - x_*|^2}{2\sigma^2}\right)\exp\left(\frac{-|x_j - x_*|^2}{2\sigma^2}\right)[\sum_{m=0}^{p-1}\frac{1}{\sigma^{2m} m!}(x_i - x_*)^m \cdot (x_j - x_*)^m + error_p]u_j$

$v_i = \sum_{j=1}^{N} \exp\left(\frac{-|x_i - x_*|^2}{2\sigma^2}\right)\exp\left(\frac{-|x_j - x_*|^2}{2\sigma^2}\right)[\sum_{m=0}^{p-1}\frac{1}{\sigma^{2m} m!} Compress((x_i - x_*)^m) \cdot Compress((x_j - x_*)^m) + error_p]\, u_j$

$v_i = \sum_{m=0}^{p-1}\frac{1}{\sigma^{2m} m!}\exp\left(\frac{-|x_i - x_*|^2}{2\sigma^2}\right) Compress((x_i - x_*)^m)\cdot\sum_{j=1}^{N}\exp\left(\frac{-|x_j - x_*|^2}{2\sigma^2}\right) Compress((x_j - x_*)^m)\ u_j + \sum_{j=1}^{N}\exp\left(\frac{-|x_i - x_*|^2}{2\sigma^2}\right)\exp\left(\frac{-|x_j - x_*|^2}{2\sigma^2}\right) error_p\ u_j$

$v_i = \sum_{m=0}^{p-1}\exp\left(\frac{-|x_i - x_*|^2}{2\sigma^2}\right) Compress((x_i - x_*)^m)\cdot C_m + \sum_{j=1}^{N}\exp\left(\frac{-|x_i - x_*|^2}{2\sigma^2}\right)\exp\left(\frac{-|x_j - x_*|^2}{2\sigma^2}\right) error_p\ u_j$

where

$C_m = \frac{1}{\sigma^{2m} m!}\sum_{j=1}^{N} u_j \exp\left(\frac{-|x_j - x_*|^2}{2\sigma^2}\right) Compress((x_j - x_*)^m)$

The absolute error in fast computation of v_i

$\in = \left|\sum_{j=1}^{N}\exp\left(\frac{-|x_i - x_*|^2}{2\sigma^2}\right)\exp\left(\frac{-|x_j - x_*|^2}{2\sigma^2}\right) error_p\ u_j\right|$

$\in <= \sum_{j=1}^{N}|error_p|\ |u_j|$

$\in <= N\,|error_p|$

$\in <= N\frac{\left(\frac{d}{4}\right)^p}{\sigma^{2p} p!}\exp\left(\frac{d}{4\sigma^2}\right)$

Figure 7 shows the maximum absolute error between the straightforward and the fast method for $N = 10^3$ and the truncation constant p varying between 1 and 11. Also shown is the theoretical bound for the corresponding

p. It can be seen that the theoretical error always bounds the computed error. Figure 8shows the CPU time required by the straightforward and the fast method.In all cases the truncation number was varied with N according to the theoretical estimate of the error for an accuracy of 10^{-6}. Figure 9 shows the maximum absolute error between the straightforward and the fast method for N varying between 10^2 and $10^3$and the truncation constant varied with N according to the theoretical estimate of the error for an accuracy of 10^{-6}. The second plot show the corresponding truncation number as a function of N.

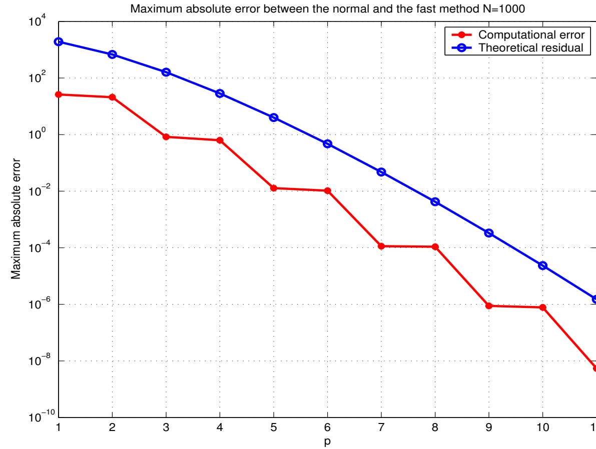

Figure 7. This plot shows the maximum absolute error between the straightforward and the fast method for $N = 10^3$ and the truncation constant p varying between 1 and 11. Also shown is the theoretical bound for the corresponding p. It can be seen that the theoretical error always bounds the computed error.

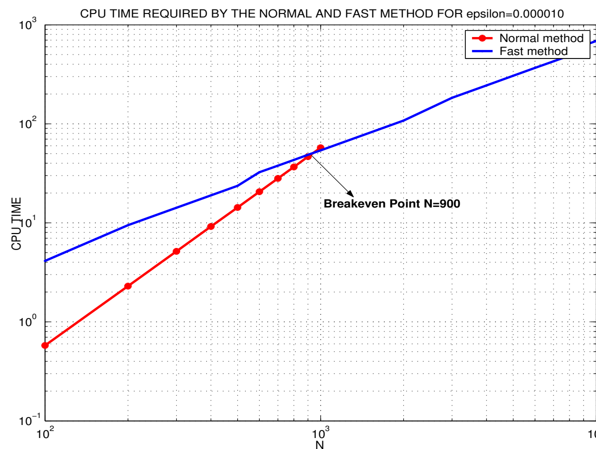

Figure 8. This plot shows the CPU time required by the straightforward and the fast method.In all cases the truncation number was varied with N according to the theoretical estimate of the error for an accuracy of 10^{-6}.

12. Example application

Kernel PCA has been used in a wide range of applications. One typical application where PCA has been conventionally use is the problem of extracting features for face detection and recognition. It has been shown that KPCA performs lot better in classification tasks than using features extracted by linear PCA. Also these tasks like face detection involve huge databases used for training and it is very computationally intensive to diagonalize the Gram matrix. The fast methods can be used in such case.

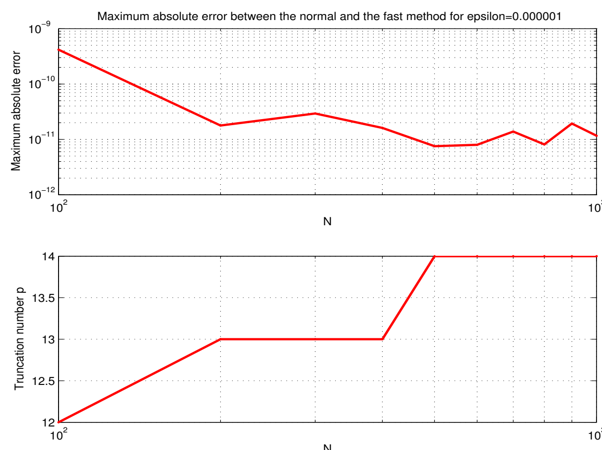

Figure 9. The first plot shows the maximum absolute error between the straightforward and the fast method for N varying between 10^2 and 10^3 and $M = 2N$ and the truncation constant varied with N according to the theoretical estimate of the error for an accuracy of 10^{-6}. The second plot show the corresponding truncation number as a function of N.

13. Application to KLDA,KBDA,KICA

Same idea can be extended to other machine learning algorithms like Kernel Linear Discriminant Analysis(KLDA), Kernel Biased Discriminant Analysis(KBDA), Kernel Independent Component Analysis(KICA) which can be proposed as a generalized eigen value problem.

14. Future work

- Demonstrate for a good application example like face detection.
- For higher dimensions evaluation of factorials could be expensive.(Solution??)
- Can use a MLFMM for the Gaussian kernel.
- Explore other kernels like the Epanichekov kernel.

References

[1] D. Achlioptas, F. McSherry, and B. Sch. Sampling techniques for kernel methods.

[2] A. Smola B. Schölkopf and K.-R. Müller. Nonlinear component analysis as a kernel eigenvalue problem. Technical Report 44, Max-Planck-Institut fr biologische Kybernetik, Tbingen, December 1996.

[3] Alexander J. Smola Bernhard Schölkopf. *Learning with kernels.* The MIT Press, Cambridge,Massachusetts, 2002.

[4] Francesco Camastra. Kernel methods for computer vision: Theory and applications.

[5] M. Girolami. *Neural Computation*, volume 14(3), chapter Orthogonal Series Density Estimation and the Kernel Eigenvalue Problem, pages 669 – 688. MIT Press, 2001.

[6] Hotelling H. Analysis of a complex of statistical variables in principal compenents. *Journal of Educational Psychology*, 26, 1933.

[7] S. Mika, B. Schölkopf, A. J. Smola, K.-R. Müller, M. Scholz, and G. Rätsch. Kernel PCA and de–noising in feature spaces. In M. S. Kearns, S. A. Solla, and D. A. Cohn, editors, *Advances in Neural Information Processing Systems 11.* MIT Press, 1999.

[8] Perry Moerland. An on-line em algorithm applied to kernel pca.

[9] Roman Rosipal and Mark Girolami. An expectation-maximization approach to nonlinear component analysis. *Neural Computation*, 13(3):505–510, 2001.

[10] Y. Saad. *Numerical Methods for Large Eigenvalue Problems.* Manchester University Press, Manchester, UK, 1992.

[11] L. Sirovitch and M. Kirby. Low dimensional procedure for the characterization of human faces. *J. Opt. Soc. Am.*, 2:519 – 524, 1987.

[12] M. Turk and A. Pentland. Eigenfaces for recognition. *Journal of Neuroscience*, 3(1):71 – 86, 1991.