DSPCA: a Toolbox for Sparse Principal Component Analysis

Ronny Luss* Alexandre d'Aspremont†

November 29, 2006

Abstract

In this paper, we describe DSPCA, a toolbox for sparse principal component analysis (PCA). Sparse PCA seeks sparse factors, or linear combinations of the data variables, explaining a maximum amount of variance in the data while having only a limited number of nonzero coefficients. We begin with a brief introduction and motivation on sparse PCA, then detail our implementation of the algorithm in d'Aspremont et al. (2005) and finish with a user guide and a set of examples.

Keywords: Sparse principal component analysis, semidefinite programming, clustering.[1]

1 Introduction

This paper focuses on the implementation of the sparse principal component analysis (PCA) relaxation introduced in [dEGJL04]. Sparse methods have had significant impact in many areas of statistics, in particular regression and classification (see [CT05], [DT05] and [Vap95] among others). As in these areas, our motivation for developing this sparse multivariate analysis toolbox is the potential of these methods for yielding statistical results that are both more interpretable and more robust than classical analyses, while giving up little statistical efficiency.

Principal component analysis is a classic tool for analyzing large scale multivariate data. It seeks linear combinations of the data variables (often called factors or principal components) that capture a maximum amount of variance. Numerically, PCA amounts to computing a few dominant eigenvectors of the data's covariance matrix. One of the key shortcomings of PCA however is that these factors are linear combinations of *all* variables, that is, all factor coefficients (or loadings) are non-zero. This means that while PCA facilitates model interpretation and visualization by concentrating the information in a few key factors, the factors themselves are still constructed

*ORFE Department, Princeton University, Princeton, NJ 08544. `rluss@princeton.edu`

†ORFE Department, Princeton University, Princeton, NJ 08544. `aspremon@princeton.edu`

[1] Mathematical Subject Classification: 90C90, 62H25, 65K05.

using *all* observed variables. In many applications of PCA, the coordinate axes have a direct physical interpretation; in finance or biology for example, each axis might correspond to a specific financial asset or gene. In such cases, having only a few nonzero coefficients in the principal components would greatly improve the relevance and interpretability of the factors. In sparse PCA, we seek a trade-off between the two goals of *expressive power* (explaining most of the variance or information in the data) and *interpretability* (making sure that the factors involve only a few coordinate axes or variables).

Earlier methods to produce sparse factors include Cadima and Jolliffe [CJ95] where the loadings with smallest absolute value are thresholded to zero and nonconvex algorithms called SCoTLASS by [JTU03], SLRA [ZZS02, ZZS04] and SPCA by [ZHT04]. This last method works by writing PCA as a regression-type optimization problem and applies LASSO [Tib96], a penalization technique based on the l_1 norm. Very recently, [MWA06b] and [MWA06a] also proposed a greedy approach which seeks globally optimal solutions on small problems and uses a greedy method to approximate the solution of larger ones. No software package implementing these results is available at this point. In what follows, we give a brief introduction to the relaxation of this problem in [dEGJL04] and describe how the smooth optimization algorithm is implemented in the DSPCA numerical package.

Given a covariance matrix $A \in \mathbf{S}_n$, the problem of finding a sparse factor which explains a maximum amount of variance in the data can be written:

$$\begin{array}{ll} \text{maximize} & x^T A x \\ \text{subject to} & \|x\|_2 = 1 \\ & \mathbf{Card}(x) \leq k, \end{array} \tag{1}$$

in the variable $x \in \mathbf{R}^n$ where $\mathbf{Card}(x)$ denotes the cardinality of x and $k > 0$ is a parameter controlling this cardinality. Computing sparse factors with maximum variance is a combinatorial problem and is numerically hard and [dEGJL04] use semidefinite relaxation techniques to compute approximate solutions efficiently by solving the following convex program:

$$\begin{array}{ll} \text{maximize} & \mathbf{Tr}(AX) \\ \text{subject to} & \mathbf{Tr}(X) = 1 \\ & \mathbf{1}^T|X|\mathbf{1} \leq k \\ & X \succeq 0, \end{array} \tag{2}$$

which is a semidefinite program in the variable $X \in \mathbf{S}^n$, where $\mathbf{1}^T|X|\mathbf{1} = \sum_{ij} |X_{ij}|$ can be seen as a convex lower bound on the function $\mathbf{Card}(X)$.

While small instances of problem (2) can be solved efficiently using interior point semidefinite programming solvers such as SEDUMI by [Stu99], the $O(n^2)$ linear constraints make these solvers inefficient for reasonably large instances. Furthermore, interior point methods are geared towards solving small problems with high precision requirements, while here we need to solve very large instances with relatively low precision. In [dEGJL04] it was showed that a smoothing technique due to [Nes05] could be applied to problem (2) to get a complexity estimate of

$$O(n^4\sqrt{\log n}/\epsilon)$$

with a much lower memory cost per iteration. The key numerical step in this algorithm is the computation of a smooth approximation of problem (2) and the gradient of the objective, which amounts to computing a matrix exponential. In section 2, we detail how the particular structure of this problem can be exploited to compute this exponential efficiently and we describe the numerical implementation of this algorithm in DSPCA. In section 3 we describe some numerical examples analyzing gene expression data. Finally, an appendix details installation and compilation procedures on various platforms.

2 Implementation

DSPCA solves a penalized formulation of problem (2):

$$\begin{array}{ll} \text{maximize} & \mathbf{Tr}(AX) - \rho \mathbf{1}^T |X| \mathbf{1} \\ \text{subject to} & \mathbf{Tr}(X) = 1 \\ & X \succeq 0, \end{array} \tag{3}$$

in the variable $X \in \mathbf{S}_n$. The dual of this program is written:

$$\begin{array}{ll} \text{minimize} & \lambda^{\max}(A+U) \\ \text{subject to} & |U_{ij}| \leq \rho. \end{array} \tag{4}$$

The algorithm in [dEGJL04] then regularizes the objective $f(U) = \lambda^{\max}(A+U)$, replacing it by the smooth (i.e. with Lipschitz continuous gradient) approximation:

$$f_\mu(U) = \mu \log \left(\mathbf{Tr} \exp((A+U)/\mu) \right) - \mu \log n,$$

whose gradient can be computed explicitly as:

$$\nabla f_\mu(U) := \left(\mathbf{Tr} \exp\left((A+U)/\mu\right) \right)^{-1} \exp\left((A+U)/\mu\right).$$

In [dEGJL04], it was shown that solving the smooth problem:

$$\min_{U \in \mathcal{Q}_1} f_\mu(U)$$

with $\mu = \epsilon / 2 \log(n)$ produced an ϵ-approximate solution to (3) and required at most $O(n\sqrt{\log n}/\epsilon)$ iterations. The main step at each iteration is computing the matrix exponential $\exp((A+U)/\mu)$. This is a classic problem in linear algebra (see [MVL03] for a comprehensive survey) and in what follows, we detail three different methods implemented in DSPCA and their relative performance.

2.1 Full eigenvalue decomposition

An exact computation of the matrix exponential can be done through a full eigenvalue decomposition of the matrix. Suppose we decompose $A = VDV^T$ with $V = (v_1, ..., v_n)$ and $D = \mathbf{diag}(d)$, where $d \in \mathbf{R}^n$ are the eigenvalues of A and $v_i \in \mathbf{R}^n$ its eigenvectors. The matrix exponential can then be computed as:

$$\exp(A) = VHV^T,$$

with $H = \mathbf{diag}(h)$ and $h_i = e^{d_i}$, $i = 1, \ldots, n$. While this is a simple procedure, it is also relatively inefficient.

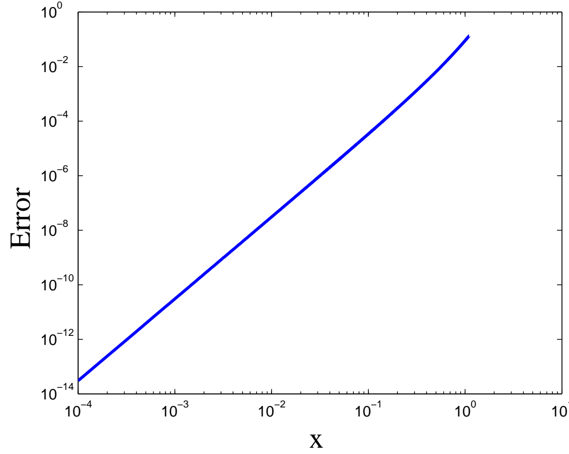

Figure 1: Padé approximation error on e^x.

2.2 Padé approximation

The next method implemented in DSPCA, which is the current method of choice for computing matrix exponentials, is called Padé approximation. This technique approximates the exponential by a rational function in a small neighborhood of zero and uses scaling to bring the eigenvalues into this neighborhood. The (p,q) Padé approximation for $\exp(A)$ is then defined by (see [MVL03]):

$$R_{pq}(A) = [D_{pq}]^{-1} N_{pq}(A), \tag{5}$$

where

$$N_{pq}(A) = \sum_{j=0}^{p} \frac{(p+q-j)!p!}{(p+q)!j!(p-j)!} A^j$$

and

$$D_{pq}(A) = \sum_{j=0}^{q} \frac{(p+q-j)!q!}{(p+q)!j!(q-j)!} (-A)^j.$$

The Padé coefficients, N and D above, can be derived by solving the linear system of equations given in [PTVF92]. Due to roundoff and computational issues, we only use approximations with $p = q$ in practice. A generic graph of the error of a Pade approximation of degree 2 on the interval [0,1] is given in Figure 1. Notice that the approximation completely fails outside of a small neighborhood of zero and the approximation only reaches a high precision near the center. This means that we need to scale down the matrix before computing its exponential using the approximation in (5), and then scale it back to its original size. This scaling and squaring can be done efficiently here using the property that $e^A = (e^{\frac{A}{m}})^m$. First, we scale the matrix A so that $\frac{1}{m}\|A\| \leq 10^{-6}$ and find the smallest integer s such that this is true for $m = 2^s$. We then use the Padé approximation to compute e^A, and simply square the matrix s times to scale it back.

Padé approximations only requires computing matrix products which can be done very efficiently. The drawback is that a matrix of size n must be squared s times. If n or s get somewhat

large, this can be costly, in which case a full eigenvalue decomposition has better performance. Numerical results illustrating this issue are detailed in the last section.

2.3 Partial eigenvalue decomposition

The first two methods we described for computing the exponential of a matrix are both geared towards producing a solution up to (or close to) machine precision. In most of the sparse PCA problems we solve here, the target precision for the algorithm is of the order 10^{-1}. Computing the gradient up to machine precision in this context is unnecessarily costly. In fact, [d'A05] shows that the optimal convergence of the algorithm in [Nes05] can be achieved using approximate function values $\tilde{f}_\mu(U)$ and gradient $\tilde{\nabla} f_\mu(U)$, provided the error satisfies the following conditions:

$$|f_\mu(U) - \tilde{f}_\mu(U)| \leq \delta \text{ and } |\langle \tilde{\nabla} f_\mu(U) - \nabla f_\mu(U), Y \rangle| \leq \delta, \quad |U_{ij}|, |Y_{ij}| \leq \rho,\ i,j = 1, \ldots, n, \quad (6)$$

where δ is a parameter controlling the approximation error. In practice, this means that only need to compute a few dominant eigenvalues of the matrix $(A+U)/\mu$ to get a sufficient gradient approximation. More precisely, if we denote by $\lambda \in \mathbf{R}^n$ the eigenvalues of $(A+U)/\mu$, condition (6) means that we only need to compute its j largest eigenvalues with j such that:

$$\frac{(n-j)e^{\lambda_j}\sqrt{\sum_{i=1}^j e^{2\lambda_i}}}{(\sum_{i=1}^j e^{\lambda_i})^2} + \frac{\sqrt{n-j}e^{\lambda_j}}{\sum_{i=1}^j e^{\lambda_i}} \leq \frac{\delta}{\rho n}.$$

Computing the j largest eigenvalues of a matrix can be done very efficiently using packages such as ARPACK if $j << n$, and when j becomes larger, the algorithm switches to full eigenvalue decomposition. Finally, the dominant eigenvalues tend to coalesce close to the optimum, thus potentially increasing the number of eigenvalues required at each iteration (see [Pat98] for example). We detail some empirical results on the performance of this technique in the last section.

3 User guide

In this section, we begin by describing the various functions contained in the DSPCA toolbox and their calling sequence. We then study a few practical examples.

3.1 Contents

The package contains two main MATLAB functions: `PrimalDec` and `DSPCA` solving (2) and (3) respectively. Examples and executables for `DSPCA` are contained in the root directory, while the those for `PrimalDec` are in the `/Small Problems` directory.

Solving small problems using SEDUMI The function `PrimalDec` uses SEDUMI by [Stu99] to solve very small instances of (2):

$$\begin{array}{ll} \text{maximize} & \mathbf{Tr}(AX) \\ \text{subject to} & \mathbf{Tr}(X) = 1 \\ & \mathbf{1}^T|X|\mathbf{1} \leq k \\ & X \succeq 0, \end{array}$$

it is mostly intended as a pedagogical tool. A call to `PrimalDec` is made as:

```
>> [resvec,resval,oval]=PrimalDec(A,k)
```

where

- $A \in \mathbf{S}_n$ is the input covariance matrix,
- k is the target cardinality,
- $resvec$ is the first eigenvector x of the matrix X, where X is the optimal solution of problem (2) above,
- $resval$ is the explained variance $(x)^T Ax$,
- $oval$ is the objective value $\mathbf{Tr}(AX)$.

Both parameters to `PrimalDec` are required. When $oval$ and $resval$ match, the semidefinite relaxation is tight (see [dEGJL04] for details).

Solving large problems using smooth optimization The main function of this toolbox is called `DSPCA` and solves problem (3):

$$\begin{array}{ll} \text{maximize} & \mathbf{Tr}(AX) - \rho\mathbf{1}^T|X|\mathbf{1} \\ \text{subject to} & \mathbf{Tr}(X) = 1 \\ & X \succeq 0. \end{array}$$

Its calling sequence is:

```
>> [U,X,x]=DSPCA(A,rho,gapchange,maxiter,info,algo)
```

where:

- $A \in \mathbf{S}_n$ is the input matrix,
- $\rho > 0$ is a parameter controlling sparsity,
- $gapchange$ is the target reduction in duality gap,
- $maxiter$ is the maximum number of iterations,

- $info$ controls verbosity: 0 is silent, $info > 1$ is the frequency of reporting,
- $algo$ controls the method for computing the matrix exponential: 1 is full eigenvalue decomposition (default), 2 is Padé approximation, 3 is partial eigenvalue decomposition,
- X is the matrix X solution of problem (3),
- U is the dual solution,
- x is the dominant eigenvector of X.

All parameters are required except for the matrix exponential algorithm option. Note that `DSPCA` is a Matlab wrapper to the mex function `sparse_rank_one_mex` and both `DSPCA.m` and the appropriate `sparse_rank_one_mex` executable must be in your Matlab path. There are also several parameters more specific to the algorithms that can be changed inside the DSPCA wrapper function that we do not describe here.

Clustering The directory `Clustering` contains three functions used to measure clustering performance. `RandPartition.m` computes the Rand index $R(X, Y)$ between partitions X and Y. `CalculateRands.m` is used to measure clustering performance in the examples that follow. Given a number of partitions k, it runs K-means clustering 1000 times on a given data set and averages the Rand index comparing the resulting partitions with the true partition. Finally `separation.m` computes the distance between the centers of two sets of points.

3.2 Examples

In this section, we detail a simple example using the two functions detailed above on a randomly generated covariance matrix. We construct a sparse rank one covariance matrix A and add noise:

```
>> n=10; ratio=2;
>> testvec=[1 0 1 0 1 0 1 0 1 0];
>> testvec=testvec/(norm(testvec));
>> A=rand(n,n);
>> A=A'*A/n+ratio*testvec'*testvec;
```

Such a small example can be solved with the function `PrimalDec` using SEDUMI:

```
>> [resvec,resval,oval]=PrimalDec(A,4)

resvec =

    0.3687
    0.0000
    0.2034
    0.0000
    0.5729
```

```
    0.0000
    0.1736
    0.0000
    0.6815
    0.0000


resval =

    2.8412


oval =

    2.8412
```

Here, because $resval$ and $oval$ are equal, we know that the relaxation is tight. We then run the large-scale function `DSPCA` on this small example as a comparison.

```
>> [X,U,u] = DSPCA(A,.5,.5e-2,1000,100,1);
DSPCA starting ...
Iter: 0.000e+000 Obj: 4.1381e+000 Gap: 4.4199e+000 CPU Time: 0h 0m 0s
Iter: 1.000e+002 Obj: 9.7366e-001 Gap: 2.0434e-001 CPU Time: 0h 0m 0s
Iter: 2.000e+002 Obj: 9.6834e-001 Gap: 4.8100e-002 CPU Time: 0h 0m 0s
Iter: 2.050e+002 Obj: 9.6833e-001 Gap: 1.2607e-002 CPU Time: 0h 0m 0s
```

and compare the result with the first eigenvector of A:

```
    DSPCA      PCA
    0.4466    0.3977
    0.0014    0.1934
    0.3991    0.3866
    0.0036    0.1352
    0.4792    0.4132
    0.0082    0.2215
    0.4134    0.3938
    0.0012    0.1672
    0.4906    0.4130
    0.0014    0.2516
```

Finally, the directory `Small Problems` also contains the files `PitpropsTest.m` and `CardversusKPLots.m` which implement the examples detailed in [dEGJL04].

4 Applications

In this section, we use the DSPCA function to analyze large sets of gene expression data. We also discuss applications of sparse PCA to class discovery and feature selection.

4.1 Gene expression data

We test the performance of DSPCA on covariance matrices generated from a gene expression data set from [ABN+99] on colon cancer. The data set is comprised of 62 tissue samples, 22 from normal colon tissues and 40 from cancerous colon tissues with 2000 genes measured in each sample. We preprocess the data by taking the log of all data intensities and then normalize the log data of each sample to be mean zero and standard deviation one, which is a classic procedure to minimize experimental effects (see [HK05]).

We first run the code for increasingly large problems using each of the three methods described in section 2 (full eigenvalue decomposition, Padé approximation and partial eigenvalue decomposition). Figure 2 depicts the results of these tests on a 3.0 GHz CPU in a loglog plot of runtime (in seconds) versus problem dimension.

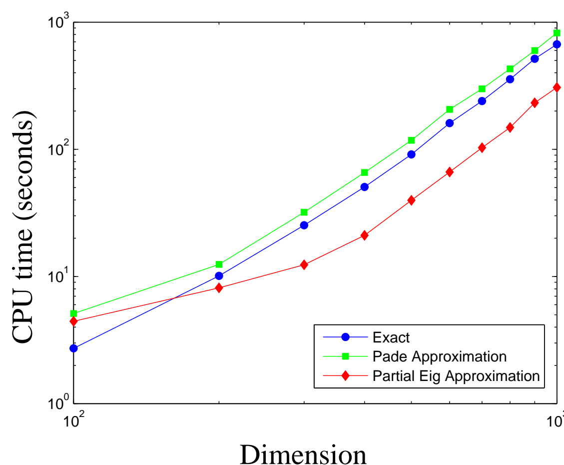

Figure 2: Running time comparison.

In these tests, partial eigenvalue is about twice as fast as the other methods for large problem sizes. Surprisingly, computing the matrix exponential using Padé approximation is slower than performing a full eigenvalue decomposition. This probably reflects the high numerical cost of matrix multiplications necessary for scaling and squaring in Padé approximations. Solving a problem of size 1000 requires about three minutes.

Another important question to study is how the number of eigenvalues needed for the partial eigenvalue decomposition impacts performance. Figure 3 (left) shows how computing time (in seconds) varies as the number of eigenvalues needed at each iteration (shown as the percentage of total eigenvalues) increases across iterations. The four tests are all of dimension 1000 and vary in degrees of sparsity, more sparsity (ie. higher ρ and less genes) requiring more eigenvalues as the algorithm proceeds. The graph on the right side of Figure 3 shows how the number of eigenvalues required in the matrix exponential computation increases when the duality gap decreases (due to the fact that the largest eigenvalues tend to coalesce near the optimum).

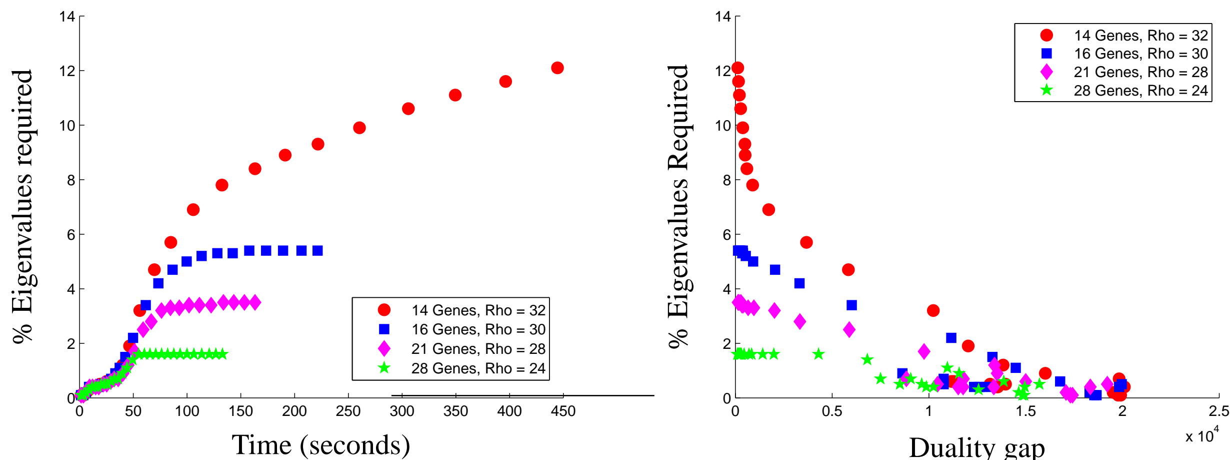

Figure 3: Eigenvalues vs CPU time (left) & duality gap vs eigenvalues (right).

4.2 Class discovery

In this section, we compare the class discovery (clustering) performance of sparse PCA to that of standard PCA. We analyze the colon cancer data set of [ABN$^+$99] as well as a lymphoma data set from [AED$^+$00]. The lymphoma data set consists of 3 classes of lymphoma denoted DLCL, FL and CL. The top half of Figure 4 displays the clustering effects of using two factors on the colon cancer data while the bottom half displays the results of the lymphoma data. Clusters are represented using PCA factors on the left, and sparse factors from DSPCA on the right. For the colon cancer data, the second factor has greater power in predicting the class of each sample, while for the lymphoma data, the first factor classifies DLCL and the second factor differentiates between CL and FL. In this example, we observe that DSPCA maintains good cluster separation while requiring far fewer genes.

We then analyze the similarity of the clusters derived from PCA and DSPCA. We first cluster the data using K-means clustering, then use the Rand index to compare the partitions obtained from PCA or DSPCA to the true partitions. The Rand index measure the similarity between two clusters X and Y and is computed as the ratio

$$R(X,Y) = \frac{p+q}{\binom{n}{2}}$$

where p is the number of elements that are in the same partition in X and in the same partition in Y, q is the number of elements not in the same partition in X and not in the same partition in Y, and $\binom{n}{2}$ is total number of element pairs. The results for the Rand index for varying levels of sparsity are plotted in Figure 5. The Rand index of standard PCA is .654 for colon cancer (.804 for lymphoma) as marked in Figure 5. The Rand index for the DSPCA factors of colon cancer using 13 genes is .669 and is the leftmost point above the PCA line. However, DSPCA on the lymphoma data does not achieve a high Rand index with very sparse factors and it takes about 50 genes per factor to get good clusters.

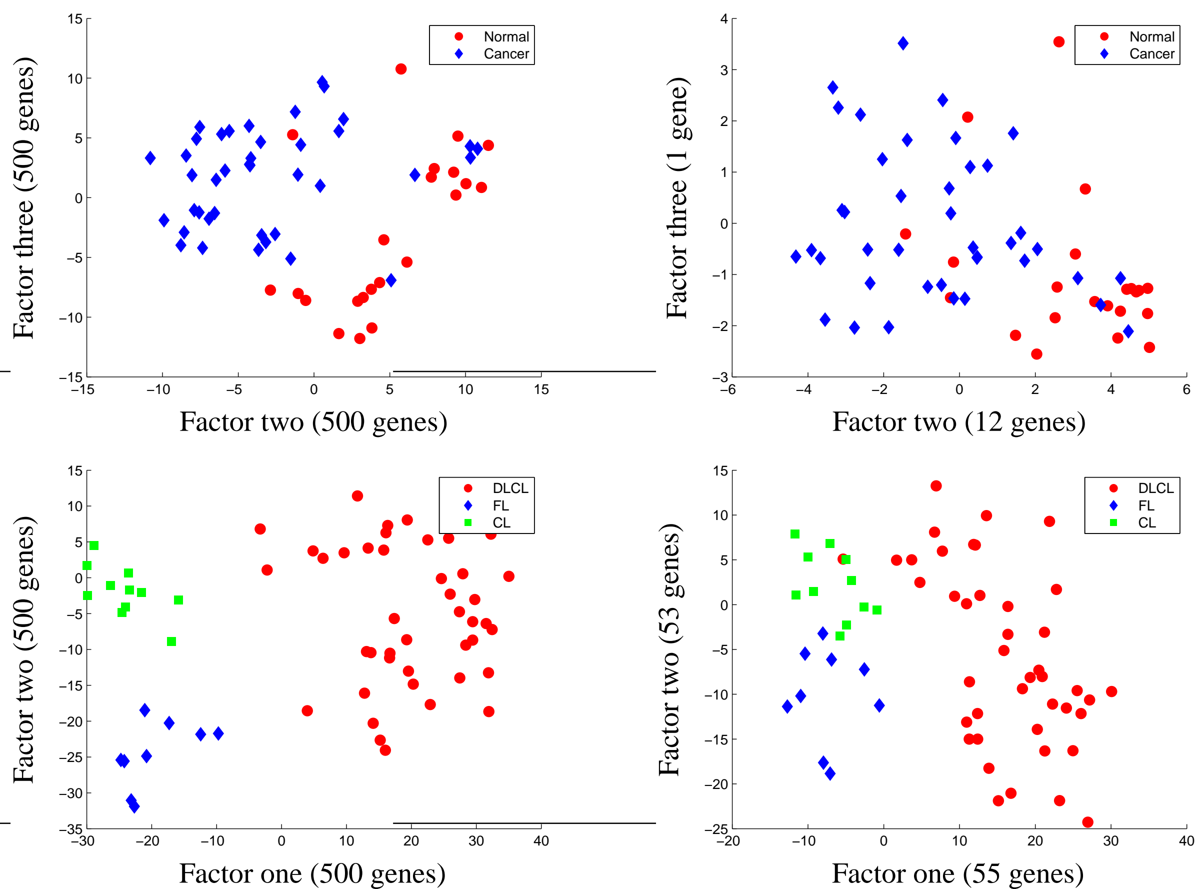

Figure 4: Clustering: using PCA (left) and DSPCA (right) on the colon cancer (top) and lymphoma (bottom) data sets.

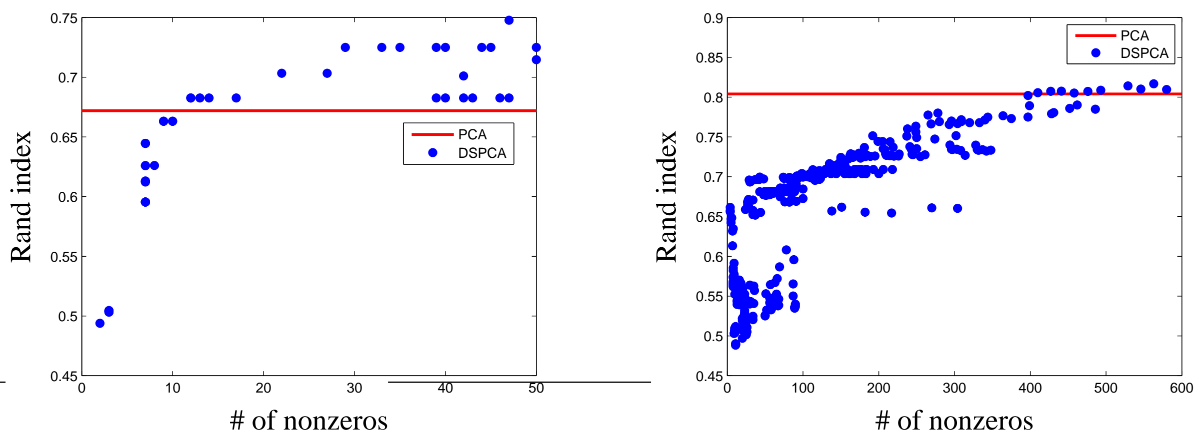

Figure 5: Rand index versus sparsity: colon cancer (left) & lymphoma (right).

For lymphoma, we can also look at another measure of cluster validity. We measure the impact of sparsity on the separation between the true clusters, defined as the distance between the cluster centers. Figure 6 shows hows this separation varies with the sparsity of the factors. The lymphoma clusters with 108 genes have a separation of 63, after which separation drops sharply. Notice that the separation of CL and FL is very small to begin with and the sharp decrease in separation is mostly due to CL and FL getting closer to DLCL.

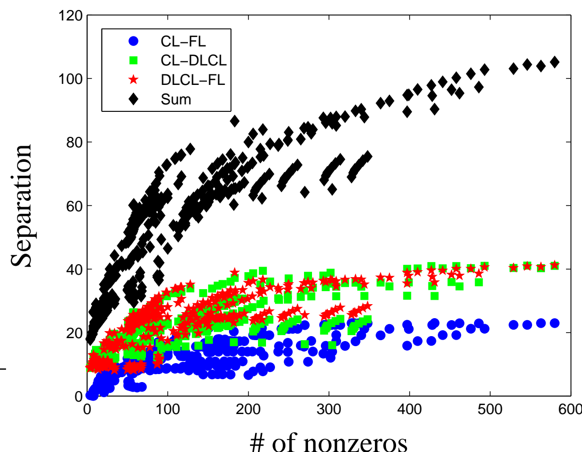

Figure 6: Separation vs sparsity: lymphoma.

4.3 Feature selection

Using the sparse factor analysis derived above, our next objective is to track the action of the remaining genes. See [GWBV02] for an example illustrating the possible relation of selected genes of this data set to colon cancer using Recursive Feature Elimination with Support Vector Machines (RFE-SVM) for feature selection. We compared the list of genes in the sparse factors produced above to genes selected by [HK05] and the Rankgene software in [SMP$^+$03]. Ten genes ranked in the top 100 of all eight methods in Rankgene and the RFE-SVM results. Five of these genes were selected by Factor 2 of the DSPCA results with twelve genes. A sixth gene was picked up by Rankgene but not listed by RFE-SVM. These genes are given in Table 4.3.

5 Appendix

The DSPCA package contains binaries for different platforms. The correct binaries (mex files) together with the wrapper function `DSPCA.m` should be placed in a directory within the MATLAB path.

DSPCA rank	Rankgene	GAN	Description
2	8.6	J02854	Myosin regulator light chain 2, smooth muscle isoform (human)
5	18.9	T92451	Tropomyosin, fibroblast and epithelial muscle-type (human)
6	31.5	T60155	Actin, aortic smooth muscle (human)
7	25.1	H43887	Complement factor D precursor (H. sapiens)
9	2.1	M63391	Human desmin gene, complete cds
10	32.3	T47377	S-100P Protein (Human).

Table 1: 6 genes selected by DSPCA and Rankgene

5.1 Installation & sources

The source code, binaries and examples can be downloaded from:

`http://www.princeton.edu/~aspremon/DSPCA.htm`

The code has been tested with MATLAB 6.1 to R2006b on Windows, Mac OS X, and Linux. The small scale example uses SEDUMI by [Stu99]. Precompiled binaries for the large scale code are provided for the same 3 operating systems. Simply copy the `.dll`, `.mexw32`, `.mexmac` or `.mexglx` file and the wrapper `DSPCA.m` into your working directory or add them to the MATLAB path. Unfortunately, these binaries are only compatible with the latest versions of MATLAB and should be recompiled when used with older ones. A MATLAB script `CompileCode.m` will compile the code directly from MATLAB. This has not been tested yet on platforms other than WIN32, Mac, or Linux and the header file `sparsesvd.h` should be adapted to the particular version of BLAS/LAPACK available on your system.

5.2 Win32

The Windows version was built with MS VC++ and a project file is provided together with the source files. The mex files can also be built directly using `CompileCode.m`. Here, the code uses the BLAS and LAPACK libraries provided in the MATLAB installation. Again, simply update the paths in the project settings or in the `CompileCode.m` file to reflect differences in your MATLAB installation and/or compiler.

5.3 Mac OS X

The Mac OS X version was built using gcc 4.0 and Xcode. The Xcode project is provided together with the source files. Simply update the ”search paths” in the project to reflect differences in the MATLAB installation on your machine. The code uses the (vector-optimized) BLAS and LAPACK implementations in the Apple provided Accelerate framework. Note that Accelerate uses a mix of CBLAS and f2c'd LAPACK, hence the difference in calling sequences.

5.4 Linux

Linux binaries should compile directly using `CompileCode.m` but path and settings may vary with the type of distribution.

Acknowledgements

The authors would like to acknowledge support from grants NSF DMS-0625352, Eurocontrol C20083E/BM/05 and a gift from Google, Inc.

References

[ABN+99] A. Alon, N. Barkai, D. A. Notterman, K. Gish, S. Ybarra, D. Mack, and A. J. Levine. Broad patterns of gene expression revealed by clustering analysis of tumor and normal colon tissues probed by oligonucleotide arrays. *Cell Biology*, 96:6745–6750, 1999.

[AED+00] A. Alizadeh, M. Eisen, R. Davis, C. Ma, I. Lossos, and A. Rosenwald. Distinct types of diffuse large b-cell lymphoma identified by gene expression profiling. *Nature*, 403:503–511, 2000.

[CJ95] J. Cadima and I. T. Jolliffe. Loadings and correlations in the interpretation of principal components. *Journal of Applied Statistics*, 22:203–214, 1995.

[CT05] E. Candès and T. Tao. Decoding by linear programming. *ArXiv: math.MG/0502327*, 2005.

[d'A05] A. d'Aspremont. Smooth optimization for sparse semidefinite programs. *ArXiv: math.OC/0512344*, 2005.

[dEGJL04] A. d'Aspremont, L. El Ghaoui, M.I. Jordan, and G. R. G. Lanckriet. A direct formulation for sparse PCA using semidefinite programming. *To appear in SIAM Review*, 2004.

[DT05] D. L. Donoho and J. Tanner. Sparse nonnegative solutions of underdetermined linear equations by linear programming. *Proceedings of the National Academy of Sciences*, 102(27):9446–9451, 2005.

[GWBV02] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik. Gene selection for cancer classification using support vector machines. *Machine Learning*, 46:389–422, 2002.

[HK05] T. M. Huang and V. Kecman. Gene extraction for cancer diagnosis by support vector machines-an improvement. *Artificial Intelligence in Medicine*, 35:185–194, 2005.

[JTU03] I. T. Jolliffe, N.T. Trendafilov, and M. Uddin. A modified principal component technique based on the LASSO. *Journal of Computational and Graphical Statistics*, 12:531–547, 2003.

[MVL03] C. Moler and C. Van Loan. Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM Review*, 45(1):3–49, 2003.

[MWA06a] B. Moghaddam, Y. Weiss, and S. Avidan. Generalized spectral bounds for sparse LDA. In *International Conference on Machine Learning*, 2006.

[MWA06b] B. Moghaddam, Y. Weiss, and S. Avidan. Spectral bounds for sparse PCA: Exact and greedy algorithms. *Advances in Neural Information Processing Systems*, 18, 2006.

[Nes05] Y. Nesterov. Smooth minimization of nonsmooth functions. *Mathematical Programming, Series A*, 103:127–152, 2005.

[Pat98] G. Pataki. On the rank of extreme matrices in semidefinite programs and the multiplicity of optimal eigenvalues. *Mathematics of Operations Research*, 23(2):339–358, 1998.

[PTVF92] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in Fortran 77: The Art of Scientific Computing, Second Edition*. Cambridge University Press, 1992.

[SMP+03] Y. Su, T. M. Murali, V. Pavlovic, M. Schaffer, and S. Kasif. Rankgene: identification of diagnostic genes based on expression data. *Bioinformatics*, 19:1578–1579, 2003.

[Stu99] J. Sturm. Using SEDUMI 1.0x, a MATLAB toolbox for optimization over symmetric cones. *Optimization Methods and Software*, 11:625–653, 1999.

[Tib96] R. Tibshirani. Regression shrinkage and selection via the LASSO. *Journal of the Royal statistical society, series B*, 58(1):267–288, 1996.

[Vap95] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer, 1995.

[ZHT04] H. Zou, T. Hastie, and R. Tibshirani. Sparse principal component analysis. *To appear in Journal of Computational and Graphical Statistics*, 2004.

[ZZS02] Z. Zhang, H. Zha, and H. Simon. Low rank approximations with sparse factors I: basic algorithms and error analysis. *SIAM journal on matrix analysis and its applications*, 23(3):706–727, 2002.

[ZZS04] Z. Zhang, H. Zha, and H. Simon. Low rank approximations with sparse factors II: penalized methods with discrete Newton–like iterations. *SIAM journal on matrix analysis and its applications*, 25(4):901–920, 2004.