pcaL1: An R Package of Principal Component Analysis using the L1 Norm

Sapan Jot
Virginia Commonwealth University

Follow this and additional works at: https://scholarscompass.vcu.edu/etd

Part of the Physical Sciences and Mathematics Commons

© The Author

Downloaded from
https://scholarscompass.vcu.edu/etd/2488

This Thesis is brought to you for free and open access by the Graduate School at VCU Scholars Compass. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of VCU Scholars Compass. For more information, please contact libcompass@vcu.edu.
pcaL1: An R Package of Principal Component Analysis Methods using the L1 norm

Sapan Jot

May 13, 2011
Abstract

Principal component analysis (PCA) is a dimensionality reduction tool which captures the features of data set in low dimensional subspace. Traditional PCA uses L2-PCA and has much desired orthogonality properties, but is sensitive to outliers. PCA using L1 norm has been proposed as an alternative to counter the effect of outliers. The R environment for statistical computing already provides L2-PCA function \textit{prcomp()}, but there are not many options for L1 norm PCA methods. The goal of the research was to create one R package with different options of PCA methods using L1 norm. So, we choose three different L1-PCA algorithms: PCA-L1 proposed by Kwak [10], L1-PCA* by Brooks et. al. [1], and L1-PCA by Ke and Kanade [9]; to create a package \textit{pcaL1} in R, interfacing with C implementation of these algorithms. An open source software for solving linear problems, CLP, is used to solve the optimization problems for L1-PCA* and L1-PCA. We use this package on human microbiome data to investigate the relationship between people based on colonizing bacteria.
# Contents

1 Introduction .......................... 6
   1.1 PCA with successive directions of maximum variation .... 9
   1.2 PCA with successive directions of minimum variation .... 11
   1.3 PCA using subspace fitting ................................. 12
   1.4 Finding robust estimate of covariance matrix to get principal components ........................................ 14

2 R Package: pcaL1 .................. 16
   2.1 PCA-L1 ........................................ 18
      2.1.1 Implementation in C ............................. 19
      2.1.2 R code and usage .............................. 20
   2.2 $L_1 - PCA^*$ ................................. 22
      2.2.1 Implementation in C ............................. 23
      2.2.2 R code and usage .............................. 23
   2.3 L1-PCA ......................................... 25
      2.3.1 Implementation in C ............................. 27
      2.3.2 R code and usage .............................. 27
   2.4 Complexity ..................................... 28
   2.5 Microbiome Data Example ............................... 30
Appendix A: DESCRIPTION File 41

Appendix B: NAMESPACE File 41

Appendix C: C Code 42

- type.h ........................................ 43
- PCA-L1 ..................................... 46
- L1-PCA* .................................... 53
- L1-PCA .................................... 73

Appendix D: R Code 89

- PCA-L1 ..................................... 90
- L1-PCA* .................................... 91
- L1-PCA .................................... 92

Appendix E: Documentation 94

- pcal1 ....................................... 95
- l1pcastar .................................. 96
- l1pca ...................................... 96
List of Algorithms

1  Kwak’s PCA-$L_1$ for Multiple PCs ........................... 19
2  L1-PCA*: Brooks, Dulá and Boone .......................... 22
3  L1-PCA : Ke and Kanade ................................. 26
# List of Figures

2.1 Directory structure of an R package .......................... 17  
2.2 Flow Chart for PCA-L1 ................................. 21  
2.3 Flow Chart for L1-PCA* ........................................ 24  
2.4 Flow chart for L1-PCA ........................................... 28  
2.5 Bacteria profiles of subjects ................................. 35  
2.6 Steps of Data Collection ................................. 36  
2.7 3-D plots for different PCA methods .................... 37
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Parameters passed to C by <code>pcal1</code></td>
<td>20</td>
</tr>
<tr>
<td>2.2</td>
<td>Arguments in R function <code>pcal1</code></td>
<td>20</td>
</tr>
<tr>
<td>2.3</td>
<td>Parameters passed to C by <code>llpcastar</code></td>
<td>23</td>
</tr>
<tr>
<td>2.4</td>
<td>Arguments in R function <code>llpcastar</code></td>
<td>24</td>
</tr>
<tr>
<td>2.5</td>
<td>Parameters passed to C by <code>llpca</code></td>
<td>27</td>
</tr>
<tr>
<td>2.6</td>
<td>Arguments in R function <code>llpca</code></td>
<td>29</td>
</tr>
<tr>
<td>2.7</td>
<td>Proportion of L1 dispersion explained by first 3 PCs</td>
<td>34</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Principal component analysis (PCA) [8], popular dimensionality reduction tool, transforms the data to a smaller set of variables which are linear combination of the original variable while retaining as much information as possible. It allows for interpretation and better understanding of the different sources of variation. So, naturally the goal of PCA is to find a set of directions that maximizes the variance of the given data. These directions constitute a low-dimensional linear subspace by which the data structure in original input space can be effectively captured. PCA is often the initial data analysis tool, followed by cluster analysis, discriminate analysis or other multivariate techniques.

The classical PCA finds the first component with the largest variance of projected observations. The second component is orthogonal to the first and maximizes the variance of data points projected on it. Proceeding in the similar manner, we find $q$ principal components explaining most of the variation in the data with $q \ll m$, where $m$ is the number of actual components (attributes).

Consider data matrix $X \in \mathbb{R}^{n \times m}$ where $x_i \in \mathbb{R}^m, i = 1, \ldots, n$ is a data vector. Each column of rotation matrix $V_q = [v_1, \ldots, v_q], q \leq m$ is a vector
of principal components loadings that define the first $q$ principal components. The first principal component is given by $v_1^T x$, second by $v_2^T x$, and so on. The $q$ PC loadings vectors are the orthogonal vectors that define a new coordinate system for the data. The PC loading vectors $v_1, \ldots, v_q$ are the eigenvectors of $\Sigma$, a covariance matrix of $X$, corresponding to the $q$ largest eigenvalues. That is, the eigenvalue $\lambda_i$ associated with PC loading $v_i$ is the variance of the data along $v_i$.

For an observation $x_i$, the principal component score is $V_q^T x_i$ and is the point projected into the space of the PC loadings. The reconstruction $x_i$ is $V_q V_q^T x_i$ and is the projected point in terms of the original coordinates.

The conventional PCA decides on a set of basis vectors that define a linear subspace so that the sum of Euclidean distances of points to their projection is minimized. To find a $q$-dimension subspace, L2-PCA minimizes the error function:

$$E_2(V,U) = \| X - VU^T \|_2^2 = \sum_{i=1}^{n} \| x_i - \sum_{k=1}^{q} v_k u_{ki} \|_2^2 = \sum_{i=1}^{n} \sum_{j=1}^{m} \left( x_{ji} - \sum_{k=1}^{m} v_{jk} u_{ki} \right)^2$$ (1.1)

A Singular Value Decomposition is often used as an interchangeable term with PCA since it helps in obtaining PCs efficiently. The singular value decomposition of a matrix $X$ is given by

$$X_{n \times d} = U_{n \times n} \Lambda_{n \times d} V_{d \times d}^T$$

where $U$ is a matrix of right eigenvectors, $V$ is a matrix of left eigenvectors and $\Lambda$ is a diagonal matrix of eigenvalues. The PCA transformation becomes

$$Y^T = X^T V = U \Lambda^T$$
The first column of $Y^T$ has the scores with respect to first principal component (PC), the second column has the scores with respect to the second PC and so on. In order to reduce dimensions, we can project $X$ down by using only first $q$ singular vectors.

The importance of SVD for PCA is thus twofold. First, it is a computationally efficient method of actually finding PCs. Second, $V$ gives us the scaled version of PC scores.

Though useful in many problems, L2-PCA has been known to be sensitive to outliers which is evident from the fact that large norm of an outlier magnifies its effect proportionally. Much research has been done in this respect ([1], [10], [9], [3]) and has led to PCA methods based on L1-norm, which though not rotationally invariant, has robust properties. In 1.1, we replace the L2-norm by L1 to define the problem as:

$$E_1(U, V) = \|X - VU^T\|_1$$

$$= \sum_{i=1}^{n} \|x_i - \sum_{k=1}^{q} v_k u_{ki}\|_1 = \sum_{i=1}^{n} \sum_{j=1}^{m} |x_{ji} - \sum_{k=1}^{q} v_{jk} u_{ki}|$$ (1.2)

where $\| \cdot \|_1$ denotes the L1-norm.

As an optimization problem, **Principal component analysis** methods can be viewed in these different ways:

- Find successive directions of maximum variation
- Find successive directions of minimum variation
- Find the best-fit subspace

Additionally, PCA can be viewed as a problem of finding robust estimate of covariance matrix, the eigenvectors of which are principal components.

All of these approaches return same results i.e. same subspace when using
L2 norm. Since L2 norm is sensitive to outliers, L1 norm has been used as an alternative in all of these approaches to propose “robust” methods for PCA. The results for PCA methods using L1 norm are different for each approach as L1 norm is not rotationally invariant. We take a look at various methods for PCA which use L1 norm in next sections.

1.1 PCA with successive directions of maximum variation

This is the traditional definition of PCA, one which finds the directions of maximum variation. Using optimization, we can write the problem as

$$\max_{\alpha_1} \text{Var}(\alpha_1^T x) = \alpha_1^T \Sigma \alpha_1$$

s.t. $\alpha_1^T \alpha_1 = 1$ \hspace{1cm} (1.3)

If $u_1$ is the optimal solution to the above, we find the solution for second PC by solving

$$\max_{\alpha_2} \text{Var}(\alpha_2^T x) = \alpha_2^T \Sigma \alpha_2$$

s.t. $\alpha_2^T \alpha_2 = 1$ $u_1^T \alpha_2 = 0$ \hspace{1cm} (1.4)

Similarly for the third PC and so on.

Choulakian et. al.[3], [4] uses this approach in centroid method and projection pursuit in order to obtain a robust PCA method. In “projection pursuit” a robust dispersion estimate, instead of variance, is used to find the directions with maximum or minimum variability. They further robustify the method by replacing variance with S-biweight dispersion. They also imbed “extended simple structure” into a family of robust discrete PCA models. The robust
centroid method handles missing data by using weight for each cell, notifying whether the element is missing or not. An R Package pcaPP, implementing the projection pursuit PCA, is available at http://cran.r-project.org/web/packages/pcaPP/index.html.

Rousseeuw [14] uses idea of repeated median regression in using Least Median of Squares (LMS) to find robust method for regression. The LMS estimator is given by

$$\min \theta \med_i (r_i)^2$$

where $\hat{\theta} = \med_i \cdots (\med_{i_{p-1}} \med_{i_p}(i_1, \ldots, i_p)))$ and has a breakdown point of 50%. The disadvantage of LMS method lies in its lack of efficiency because of its $n^{-1/3}$ convergence. The author suggests various alternatives as a remedy for this problem like combining LMS and one-step M estimator methods.

Kwak [10] modifies the idea of finding directions of maximum variation to maximizing the L1 dispersion using the L1-norm in the feature space as following:

$$U^* = \arg \max_U \| U^T X \|_1 = \arg \max_U \sum_{i=1}^{n} \sum_{k=1}^{d} \sum_{j=1}^{q} u_{jk} x_{ji}$$

subject to $U^T U = I_q$ (1.5)

Since the maximization is done on the feature space, the solution to 1.5 is invariant to rotations and since L1-norm is used, the results are more robust to outliers. The author further proposes an algorithm to approximate the idea of PCA-L1 from 1.5. The positive point of proposed algorithm is that number of iterations taken to find the best feature space is dependent on the number of samples $n$, not on the dimension $m$. That is, the data with large number of dimensions can use PCA-L1 without making it computationally intensive. The only downside of the algorithm is that it finds a local maximum solution which on the initialization vector.
We implement Kwak’s PCA-L1 in C and add it in R Package `pcaL1`. The further discussion on implementation of PCA-L1 can be found in section R Package : `pcaL1`.

1.2 PCA with successive directions of minimum variation

The exact reverse of PCA with successive directions of maximum variation is to find the directions with minimum variation. Using optimization, the problem for finding directions of minimum variation is written as follows:

\[
\begin{align*}
\min_{V_{q'}} & \quad tr\left(V_{q'}^T \Sigma V_{q'}\right) \\
\text{s.t.} & \quad V_{q'}^T V_{q'} = I \quad \text{where} \quad V_{q'} \in \mathbb{R}^{m \times q}
\end{align*}
\]

(1.6)

The optimal solution here is \(U_{q'}\), a matrix of last \(q\) columns of \(U\).

Brooks et. al. [1] propose a method called \(L_1 - PCA^*\) which searches for globally optimal best-fit hyperplanes, which minimize the sum of \(L_1\) distances of points to their projections. The method is based on the idea that \(m\) \(L_1\) linear regressions, where each variable becomes independent variable by turn and the regression hyperplane is selected on the basis of smallest total error, can find a best-fit hyperplane. A theorem is proposed using linear programming [15] formulation, which is given by:

\[
\begin{align*}
\min \sum_{i=1}^{n} \| x_i - V \alpha_i \|_1 & \iff \min \| X^T - U V^T \|_1 \\
\text{where} \quad X & \in \mathbb{R}^{n \times m}, U \in \mathbb{R}^{n \times (m-1)}, V \in \mathbb{R}^{m \times (m-1)}
\end{align*}
\]

(1.7)

\(m\) of such regressions problems are solved to obtain the globally optimal hyperplane. An algorithm is proposed based on 1.7, which finds an \(L_1\) best-
fit $m - 1$ dimensional subspace for $m$-dimensional data. We implement this algorithm in our R package. Details on implementation and further discussion can be found in R Package : pcaL1.

1.3 PCA using subspace fitting

For a data matrix $X$ of size $n \times m$, subspace estimation is achieved by matrix factorization:

$$X_{n \times m} = U_{n \times q} V^{T}_{m \times q}$$

The $q$ columns of $V$ are the basis of linear subspace.

The problem of subspace estimation, using optimization, is viewed as a minimization problem of the sum of squared distances of points to their projections onto a linear subspace. The problem is written as

$$\min_{V_q} \sum_{i=1}^{n} \| x_i - V_q (V_q^T x_i) \|_2^2$$

subject to $V_q^T V_q = I$ \{ $x \in \mathbb{R}^d \mid V x = 0$ \}

The optimal solution here is $V_q$.

Ke and Kanade [9] propose a robust PCA method based on the Iteratively Re-weighted Least Squares (IRLS) with subspace estimation using $L_1$ norm. They use alternate convex programs, to formulate the robust $L_1$ norm matrix factorization, which can be solved efficiently by linear programming. For a given $X$ data matrix, the $L_1$ norm minimization obtained by using MLE is given by

$$E_L(V, U) = \min \| X^T - V U^T \|_1$$

The cost function in above equation is minimized alternatively over $V$ and $U$ to optimize one argument while keeping the other fixed, hence the name alternative.
The optimization problem can be written as:

\[ U^{(t)} = \arg \min_U \| X^T - V^{(t-1)}U^T \|_{L_1} \] (1.8)

\[ V^{(t)} = \arg \min_V \| X^T - VU^{(t)}^T \|_{L_1} \] (1.9)

The cost function of 1.8 can be broken into \( n \) independent small subproblems given by:

\[ u_j = \arg \min_u \| V^{(t-1)}u - x_j \|_1 \] (1.10)

We implement L1-PCA in our R Package. Further discussion about the algorithm and implementation of L1-PCA can be found in R Package: pcaL1.

Maronna [12] extends the idea of “projection pursuit”, proposed by Choulakian [3], for a robust PCA which searches for low-dimensional fit directly instead of one component at a time. An algorithm is proposed, using this concept, to find a \( q \)-dimensional linear subspace that “optimally fits” data along the directions of minimized M-scale estimates or “trimmed squares” scale. The algorithm finds local minimum iteratively, and a random search is used to find the global minimum. Maronna and Yohai [13] extend this idea further to robust estimates for \( X \) with both elementwise and row-wise contamination. A contamination can be defined as the probability \( \pi = 1 - (1 - \varepsilon) \) that a row contains at least a “bad” contamination where \( \varepsilon \) is the probability with which each element is altered at random. The estimates provided by this article can be computed rapidly when number of attributes, \( m \), is large and can also handle missing data very well. Using MM estimation [16], the robust linear regression estimates are obtained. The residuals of these estimates are used to compute a robust error scale and the final M estimate is computed iteratively using the initial one as a starting point.
1.4 Finding robust estimate of covariance matrix to get principal components

For multivariate data, a covariance matrix helps in finding the atypical observation in terms of relation of one variable to another. The solution for the PCA of usual covariance matrix $\Sigma$ is given by an eigen analysis of $\Sigma$. Campbell [2] modifies the analysis by replacing $\Sigma$ by a robust estimator, M-estimator [11], which gives full weight to observations assumed to come from the main body of the data and downweighs the influence of observations with unduly large Mahalanobis distances. An observation is weighted according to its total distance $d_m$, calculated using Mahalanobis distance, from the robust estimate of location. The components of this distance along each eigenvector may lead to an incorrectly downweighted observation which has a large component along one direction and small components along others. So, M-estimators for mean and variance are used to find the principal components with minimum influence of atypical observations.

Another application of replacing covariance matrix has been proposed by Galpin and Hawkins [7]. Conventionally, PCA is applied to an estimate of $\Sigma$ to yield estimates ($\hat{\lambda}_i, \hat{\mu}_i$) of the variance and direction cosines of the population principal components, respectively. Galpin and Hawkins [7] use the idea in opposite direction, i.e. estimates of $\lambda_i$ and $\mu_i$ are used to obtain estimates of $\Sigma$. Variety of techniques are used to determine eigenvectors and eigenvalues; the estimates for eigenvectors are obtained by using QP techniques (Q) or LP techniques (L), and the eigenvalue estimates are determined by directly using the minimum spread (M) or by a scale measure of the projections (P). To find case weights (W), the Mahalanobis distance is used with robust covariance matrix estimate and robust location estimate. All the combinations of techniques
for finding estimates for eigenvalues and eigenvectors are applied on different datasets. The two-fold goal of the method is to provide ‘best’ covariance matrix estimate and to identify outliers. The ‘best’ covariance estimates are given by \( \hat{\Sigma}_{LP}, \hat{\Sigma}_{LPW}, \hat{\Sigma}_{QP}, \) and \( \hat{\Sigma}_{QPW} \) and outliers were identified by \( \hat{\Sigma}_{LPW} \) and \( \hat{\Sigma}_{QPW} \).

Three algorithms, PCA-L1 [10], L1-PCA* [1] and L1-PCA [9], each from a different type of method for PCA are chosen. We implement these algorithms in C and then using C interface for R, we create an R package called \textit{pcaL1}. In chapter 2, we discuss the linear program solver CLP and the details of R Package \textit{pcaL1} with each algorithm implementation as a section. The last section of chap 2 demonstrates the application of \textit{pcaL1} on microbiome Data.
Chapter 2

R Package: pcaL1

As discussed in the previous section, a variety of views have been proposed for PCA, but still there is no one software available which compiles all the different ideas for PCA. As mentioned earlier in section 1.1, “projection pursuit” by Choulakian [3] is the only available L1-norm PCA method in R. So, we write an R Package which implements three different algorithms in one package. The first algorithm implemented is PCA-L1, proposed by Kwak [10], which finds successive directions of maximum variation. Second is $L1 - PCA^*$, proposed by Brooks, Dulá and Boone [1], which finds successive directions of minimum variation. And lastly we implement L1-PCA, proposed by Ke and Kanade [9], which uses subspace estimation to solve PCA. The R package that implements these three PCA techniques is called pcaL1.

Since memory management is difficult in R, we implement our set of algorithms in C and then interface C code with R. The C code in turn uses CLP- a free LP solver-to solve LP’s formulated in 1.5, 1.6, 1.8 and 1.9. CLP is written in C++ and makes use of inheritance properties of OOP extensively. We use a C wrapper for these C++ classes of CLP for all the functions...
needed to solve the above mentioned LPs like \texttt{ClpLoadProblem}, \texttt{ClpOptimize}, and \texttt{ClpGetColSolution}. A reference for the functions in CLP can be found at \url{http://www.coin-or.org/Doxygen/Clp/index.html}.

A manual is available online for “Writing R Extensions”, at \url{cran.r-project.org/doc/manuals/R-exts.html}, to provide guidelines for the R package. A typical R package should have a DESCRIPTION file and three folders, namely src, R and man. A src/ folder contains the source code in C (or C++ or Fortran), an R/ folder contains R code interfacing with the source code in src/ and man/ has the documentation files for the function defined in R/. Optionally, an R package can have a data and an examples folder as well. Figure 2.1 shows the structure of our package pcaL1.

![Directory structure of an R package](image)

Figure 2.1: Directory structure of an R package
A general set of rules/instruction apply on implementation of each algorithm. In R, C interface is used to call the function from C code. This interface allows passing only pointers to the C code. A table is given in R implementation section of each code, which gives the parameters passed to C. In C, a “type.h” file is shared among all the algorithms as this file contains the initialization of structures used for solving the problem for our package. The following three structures are defined:

1. entityinfo: initializes the structure for the data matrix and it’s dimensions.

2. solverinfo: defines the CLP models for the problem.

3. probleminfo: initializes the variables for solving the problem like intermediate matrices for storing the scores, transformed points etc.

Proportion of L1 dispersion explained is used as an equivalent to proportion of variance explained, since all the methods in pcaL1 are minimizing the L1 error function. The formula for calculating the proportion of dispersion Brooks in the $k$-dimensional best-fit hyperplane is given by:

$$\frac{\sum_{i=1}^{n} |x_{ij}^{k} - \bar{x}_{j}^{k}|}{\sum_{j=1}^{m} \sum_{i=1}^{n} |x_{ij} - \bar{x}_{j}|}$$

In R code, we take the absolute sum of scores of each attribute and divide it with total L1 dispersion.

We will discuss each algorithm, it’s implementation and it’s usage in R individually.

## 2.1 PCA-L1

The algorithm for PCA-L1, proposed by Kwak [10], finds the projection vector $u$ that maximizes the L1 objective function in eq.1.5. Kwak [10] also gives greedy
search algorithm to find arbitrary principal components by applying PCA-L1 to the remainder of projected samples. This algorithm for extracting multiple features is given next.

Algorithm 1 Kwak’s PCA-L1 for Multiple PCs
Given a data matrix $X \in \mathbb{R}^{n \times m}$.
1: Set $v_0 = 0, X^0 = X$.
2: for $(j = 1; j \leq q; j++)$ do
3: Set $X^j = X^{j-1} - v_{j-1}^T X^{j-1}$.
4: Initialization: Set $v(0) = \arg\max_{x_i} \| x_i \|_2$. Set $v(0) \leftarrow v(0)/ \| v(0) \|_2$ and $t = 0$.
5: Polarity Check: For $i \in \{1, \ldots, n\}$, if $v^T(t)x_i < 0$, $p_i(t) = -1$, otherwise $p_i(t) = 1$.
6: Flipping and Maximization: Set $t = t + 1$ and $v(t) = \sum_{i=1}^{n} p_i(t-1)x_i$. Set $v(t) \leftarrow v(t)/ \| v(t) \|_2$.
7: Convergence Check:
8: if $v(t) \neq v(t-1)$ then
9: Go to Polarity Check.
10: else
11: if There exists $i$ such that $v^T(t)x_i = 0$ then
12: Set $v(t) \leftarrow (v(t) + \triangle v)/ \| v(t) + \triangle v \|_2$ and go to Polarity Check.
13: Here, $\triangle v$ is small non-zero random vector.
14: end if
15: else
16: Set $v_j \leftarrow v(t)$ and stop.
17: end if
18: end for

2.1.1 Implementation in C

`pcaL1`, function for PCA-L1 in R, calls `PcaL1_R` using .C interface. Table 2.1 shows the parameters passed through C interface to the compiler. The `PcaL1_R` function then calls `allocateMemoryPcaL1`, which allocates the memory for data matrix, for intermediate matrices for storage of results, and for result matrix of principal components. After allocation of memory `solvePcaL1` is called, which implements the PCA-L1 algorithm given in Alg. 1.

To implement PCA-L1, `solvePcaL1` first calculates the $X_j$ for the $j$-th prin-
points | a pointer array for the data matrix.
dataDim | a pointer array for passing the dimensions of data matrix.
q | number of dimensions to project data into.
PCs | pointer array for returning the principal components.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>data: in form of table or a matrix.</td>
</tr>
<tr>
<td>projDim</td>
<td>number of dimensions to project into, default is 1</td>
</tr>
<tr>
<td>center</td>
<td>whether to center the data using Median, default is false</td>
</tr>
<tr>
<td>score</td>
<td>if TRUE returns the Scores matrix, default is FALSE.</td>
</tr>
<tr>
<td>dispersionExp</td>
<td>if TRUE returns the proportion of L1 dispersion explained, default is FALSE.</td>
</tr>
</tbody>
</table>

Table 2.1: Parameters passed to C by `pcal1`

Table 2.2: Arguments in R function `pcal1`

Principal component. Then a vector is initialized based on the point with the largest norm, which is followed by steps of polarity check and checking for convergence by calling `polarityChk` and `ChkConvergence` function respectively, until the LP converges. A flowchart is given in Fig. 2.2 to show the execution of code.

### 2.1.2 R code and usage

R function `pcal1` calls the C code using `.C` interface. Table 2.2 shows the arguments user needs to give to the R function.

Kwak [10] gives a toy problem with outlier to show the implementation of PCA-L1. We use the same data to check for success of our implementation. The data matrix $X$ consists of 11 data points in 2D:

$$X = \begin{bmatrix} -6 & -5 & -4 & -3 & -2 & 10 & 0 & 1 & 2 & 3 & 4 \\ -5 & -4 & -3 & -2 & -1 & 0 & 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$
So, using \textit{pcal1} function of R package \textit{pcaL1}, we get the following result:

\begin{verbatim}
R> pcal1(X, projDim=1, center=TRUE)

$PCs
   [,1]
[1,] 0.8
[2,] 0.6
\end{verbatim}
2.2 $L1 - PCA^*$

PC obtained from $pca1$ matches exactly with the result given in Kwak [10] of $[0.8,0.6]^T$. Brooks et. al. [1] propose an algorithm for $L1$-$PCA^*$, which searches for a globally fit hyperplane using $k$ regressions. For each $k$, the $k^{th}$ variable is fixed as dependent variable and rest of the attributes are used as independent variables to solve the regression. A hyperplane is generated by each of the $k$ regressions and the one with the smallest total error is globally optimal. The LP formulation for finding best fit hyperplane using L1 regressions is given below:

$$j^* = \arg \min_{j=1,\ldots,k} R_j(X) = \min_{\beta, e^+, e^-} \sum_{i=1}^{n} e_i^+ + e_i^-$$

subject to

$$\beta^T x_i + e_i^+ - e_i^- = 0, i = 1, \ldots, n,$$

$$\beta_j = -1,$$

$$e_i^+, e_i^- \geq 0, i = 1, \ldots,$$

where $R_j^*(X)$ has the vector $\beta$ which has the coefficient of best-fit hyperplane $\beta^*$. The algorithm for $L1$-$PCA^*$ is given next:

**Algorithm 2** L1-$PCA^*$: Brooks, Duló and Boone

Given a data matrix $X \in \mathbb{R}^{n \times m}$ with full column rank.

1: Set $X^m = X$; set $V^{m+1} = I$; set $(I^*)^{m+1} = I$.
2: for $(k = m; k > 1; k = k - 1)$ do
3: Set $j^* = \min_j R_j(X^k)$ and $\beta_k = \beta^*$ according to Theorem in [1].
4: SET $Z^k = (X^k)((I^+)^k)^T$.
5: Calculate the SVD of $Z^k$, $Z^k = UAV^T$, and set $V^k$ to be equal to the $k - 1$ columns of $V$ corresponding to the largest values in the diagonal matrix $A$.
6: Calculate the $k^{th}$ principal component $\alpha^k = (\prod_{\ell=m+1}^{k+1} V^\ell) \beta_k / \| \beta_k \|_2$.
7: Set $X^{k-1} = Z^k V^k$.
8: end for
9: Set $\alpha^1 = \prod_{\ell=m+1}^2 V^\ell$. 

22
2.2.1 Implementation in C

R function for L1-PCA* calls the *l1pcaStar* function using .C interface. Table 2.3 shows the parameters passed by .C interface to C compiler. The *l1pcaStar* function first allocates the memory using *allocateMemory* function, and then calls *solveproblem* in order to obtain PC’s using L1-PCA*.

The *solveproblem* function then sets up the CLP model for the problem using *setupClp* and this model is then used to solve k regressions, as proposed in the algorithm. The *solveproblem* function then loops between loading up all the matrices, changing their bounds to create a response, optimizing the loaded problem to get objective value and then storing coefficients of regression to calculate the projected points later. A flow chart for the function in *l1pcaStar* is given in Fig. 2.3.

2.2.2 R code and usage

As mentioned earlier, .C function of R is used to interface with C code for the algorithm. R function passes the following parameters to the C compiler:

To check if the algorithm has been implemented successfully, we run the

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>points</td>
<td>a pointer array of data points.</td>
</tr>
<tr>
<td>dataDim</td>
<td>dimension of the data matrix.</td>
</tr>
<tr>
<td>q</td>
<td>number of dimensions to project into.</td>
</tr>
<tr>
<td>getScores</td>
<td>return Scores matrix if TRUE. Default is FALSE.</td>
</tr>
<tr>
<td>getProjPoints</td>
<td>return matrix of Projected Points if TRUE. Default is FALSE.</td>
</tr>
<tr>
<td>PCs</td>
<td>a pointer array to return principal components.</td>
</tr>
<tr>
<td>Scores</td>
<td>a pointer array for returning Scores matrix if getScores is TRUE.</td>
</tr>
<tr>
<td>projpoints</td>
<td>a pointer array to return projected points if getProjPoints is TRUE.</td>
</tr>
</tbody>
</table>

Table 2.3: Parameters passed to C by *l1pcaStar*
Figure 2.3: Flow Chart for L1-PCA

<table>
<thead>
<tr>
<th>X</th>
<th>Data, in form of matrix or table.</th>
</tr>
</thead>
<tbody>
<tr>
<td>projDim</td>
<td>no. of dimensions to project into, default is 1.</td>
</tr>
<tr>
<td>center</td>
<td>centers the data using median is TRUE, default is FALSE.</td>
</tr>
<tr>
<td>getScore</td>
<td>if TRUE returns scores matrix, default is FALSE.</td>
</tr>
<tr>
<td>getProjPoints</td>
<td>True if want projected points, default is FALSE.</td>
</tr>
<tr>
<td>dispersionExp</td>
<td>returns L1 dispersion explained for each PC if TRUE, default is FALSE.</td>
</tr>
</tbody>
</table>

Table 2.4: Arguments in R function `Upcastar`
code on toy problem given in Brooks et. al. [1]. The three dimensional data
matrix is given below:

\[
(X^3)^T = \begin{bmatrix}
-1.17 & 0.53 & -1.02 & 1.12 & 2.08 & -1.61 & 1.17 & 2.00 & 3.00 & 3.00 \\
1.20 & 0.24 & 0.40 & 1.36 & -1.82 & 0.53 & -1.52 & -1.03 & -2.00 & 3.00 \\
-0.30 & -1.00 & 1.11 & -1.69 & -0.76 & 0.99 & 0.71 & -1.44 & -1.00 & 3.00
\end{bmatrix}.
\]

The results of l1pcastar agree with results of toy problem implemented with
CPLEX (another optimization solver) in Brooks et. al. [1]. The result from
l1pcastar is shown below:

R> l1pcastar(X, projDim=2, center=T, getScore=T)

[,1] [,2]
[1,] 0.8  0.04
[2,] -0.53 -0.4  
[3,] -0.27 0.92

2.3 L1-PCA

L1-PCA is proposed by Ke and Kanade [9] which uses alternative convex min-
imization, a scheme that minimizes the cost function alternatively over \( U \) or
\( V \), each time optimizing one argument while keeping the other one fixed. Ke
and Kanade [9] suggest the decomposition of eq.1.8 into \( n \) independent small
subproblems in order to optimize \( u_i \), where \( u_i \) is the \( i \)-th column of \( U^T \). The
decomposed optimization problem can be written as:

\[
\text{for } t = 1, \ldots, \text{convergence :}
\]

\[
\text{for } i = 1, \ldots, n:
\min_{\delta, u_i} \delta_1^T \delta
\]

\[
s.t. \ -\delta \leq V^{(t-1)} u_i - x_i \leq \delta
\]

The formulation of LP for finding rotation matrix \( V \) is given below:

\[
\min \| X^T - V^{(t)} U^{(t)T} \|_1
\]

\[
\Rightarrow \min \sum \lambda_i^+ + \lambda_i^-
\]

\[
s.t. \ X_i - V^{(t)} u_i^{(t)T} + \lambda_i^+ - \lambda_i^- = 0
\]

\[
\lambda_i^+, \lambda_i^- \geq 0
\]

Since the algorithm proposed is iterative, Ke and Kanade [9] suggest random initialization of \( V \) in the beginning but to give it a good start, we use solution of L2-PCA in R to get \( V \) which is then send to L1-PCA as initialized \( V \). The cost function converges when the difference of parameters between iterations is small enough.

\begin{algorithm}
\caption{L1-PCA : Ke and Kanade}
1: Initialization: \( V^0, \Sigma^0 = I \).
2: for \( t = 1, \ldots, \text{convergence} \) do
3: \( U^{(t)} = \arg \min_U \| X^T - V^{(t-1)} \Sigma^{(t-1)} U^T \|_{L_1} \).
4: \( V^{(t)} = \arg \min_V \| X^T - V \Sigma^{(t-1)} U^{(t)T} \|_{L_1} \).
5: Normalize \( V \)
6: end for
7: Output: \( V \leftarrow V \Sigma^{1/2}, U \leftarrow U \Sigma^{1/2} \)
\end{algorithm}
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>points</td>
<td>a pointer array for data matrix</td>
</tr>
<tr>
<td>dataDim</td>
<td>dimension of data matrix</td>
</tr>
<tr>
<td>q</td>
<td>number of dimensions to project into</td>
</tr>
<tr>
<td>tolerance</td>
<td>user-defined tolerance for the convergence.</td>
</tr>
<tr>
<td></td>
<td>Default is .001.</td>
</tr>
<tr>
<td>iterations</td>
<td>user-defined number of iterations before returning the result. Default is 10.</td>
</tr>
<tr>
<td>initV</td>
<td>a pointer array for initial V (result of L2-PCA).</td>
</tr>
<tr>
<td>getScores</td>
<td>whether to calculate scores or not. Default is FALSE.</td>
</tr>
<tr>
<td>PCs</td>
<td>an array of pointers to return PCs.</td>
</tr>
<tr>
<td>Scores</td>
<td>a pointer array for returning Scores.</td>
</tr>
</tbody>
</table>

Table 2.5: Parameters passed to C by l1pca

2.3.1 Implementation in C

R function for L1-PCA, l1pca, interfaces with C by calling the function l1pca from C code. R passes a set of parameters to the function through .C interface, which are given in table 2.5.

V is initialized in R as the rotation matrix from solution of L2-PCA and is send to C using .C interface. Memory is allocated for by allocateMemory2 function on call by l1Pca. Then solveL1Pca is run to implement the algorithm shown in Alg 3. First of all, CLP model is set up for U and V, functions solveforU and solveforV are called iteratively until the results converge. The columns of U are returned as scores and columns of V as principal components. A flow chart is shown in Fig.2.4 to depict the steps of C code.

2.3.2 R code and usage

R function, l1pca, is the function user calls to get principal components from L1-PCA algorithm (Alg 3) while using pcaL1 package. User gives the arguments shown in table 2.6 to function.

The results of l1pca function from R package pcaL1 for toy problem from
Kwak [10] returns PC of \{0.7808, 0.6247\}, which is very close to the expected result, with 73% of L1 dispersion explained. Also for toy problem in Brooks et. al. [1], the results of \textit{l1pca} are in compliance with the results of \textit{l1pcastar}.

### 2.4 Complexity

The complexity of PCA-L1 is $O(nm \times n_{it})$, where $n_{it}$ is the number of iterations. From the Flipping and Maximization step in pseudo code 1, it can be noticed that once the polarity of a point has been changed, it will not revert back. So, the number of iterations is bounded by number of points and the complexity for PCA-L1 becomes $O(n^2 m)$ in worst case scenario.
The complexity of L1-PCA* is $O(m^2) \times O(LP)$, where $LP$ is the complexity of linear program. From Chvátal [5], the complexity of an LP is approximated as $O(c \log v)$, where $c$ is the number of constraints in LP and $v$ is the number of variables. So, from eq. (2.1) for $k$ L1 regression, which are LP with $n$ constraints and $2n+k$ variables, for each dimension, the complexity for L1-PCA* algorithm can be approximated as $O\left(\sum_{k=1}^{m} kn \log(2n+k)\right)$ which is slightly more efficient than $O\left(m^2 n \log(2n+m)\right)$. We can see that algorithm L1-PCA* can be favored over PCA-L1 when $n$, number of entities, is large.

The complexity of linear programs in algorithm of L1-PCA can be obtained using Chvátal [5] again. In $n$ LPs of eq.2.2, there are $qm$ number of variables and $2m$ constraints and in LP of 2.3, we have $2nm + mq$ variables and $nm$ constraints. So, the complexity of L1-PCA becomes $O\left[\left(\left(nm \log(2nm + mq) + 2nm \log(qm)\right) \times n_{it}\right)\right]$, where $n_{it}$ is the number of iterations. The complexity, after algebraic reduction, can be approximated by $O\left(\left(nm \log(2nm + mq)\right) \times n_{it}\right)$. If the $n_{it}$, number of iterations, is equal to $m$ then L1-PCA performs better than L1-PCA* in terms of complexity, and if $n_{it}$ is equal to $n$ then L1-
PCA takes lesser time than PCA-L1.

2.5 Microbiome Data Example

We demonstrate the usage of R package : pcaL1 on “Micorbiome data”. A 496 \times 34 in size, has bacteria profile of subjects (in this case humans) i.e. each bar in figure 2.5 shows the percentage of different bacteria in each subject’s sample. The colors of different bacterias has been manually identified to analyze these colored clusters for relationships amongst them. All of the methods in R package are used on this data and the results are plotted, with the help of rgl[6] (an R Package for 3d plots), and discussed to show the different results from different L1-PCA algorithms.

The samples of bacteria are collected at different clinics and is sent to microbiology lab from there to look for the bacterias under study. The results from the lab then undergo DNA sequencing to associate DNA of various bacteria. At this point, each subject has about 20-30,000 DNA sequences associated with him/her. These sequences are put in percentage form after identifying the bacteria and the data is set up for analysis. A pictorial representation of data collection is shown in Fig 2.6.

The first column of data is manually identified labels, so we put the values of this label in a separate vector which are used later to color code the points in 3-D plots. The code for that is shown below:

```r
mydata <- read.table("clpcode/exec/microbiometypes.txt")
mylabels <- mydata[,1]
mypcadata <- mydata[,c(1)]
```
The code for calling pcaL1 functions and results on R are shown below:

R> mykwak <- pcaL1(mypcadata, projDim=3, score=TRUE, var=TRUE)
R> mykwak$PCs
   [,1]        [,2]        [,3]
[1,] 0.2456316461 -4.122662e-01 -0.420780990
[2,] 0.6922667374  4.532892e-01  0.474504660
[3,] 0.4868120050  2.043177e-01 -0.656833206
[4,] 0.4235189959 -7.595785e-01  0.324092902
[5,] 0.1096279853  4.394477e-03 -0.153313731
[6,] 0.0759128653  5.124354e-02 -0.083999800
   .      .      .
   .      .      .
   .      .      .
[27,] 0.0029555109  1.570871e-03 -0.002690254
[28,] 0.0022701248  1.753319e-03 -0.002962853
[29,] 0.0020519217  1.475746e-03 -0.001779449
[30,] 0.0018875863  1.538587e-03 -0.002830030
[31,] 0.0017154425  7.478593e-05 -0.002334359
[32,] 0.0008679204  6.843594e-04 -0.001272212
[33,] 0.0949796432 -1.513827e-02 -0.078562795

R> mykwak$PropL1Dispersion
[1] 0.3442172 0.3095441 0.3030566

R> mylpcastar <- l1pcastar(mypcadata, projDim=3, getScore=TRUE, getProjPoints=TRUE, propVar=TRUE)
R> myl1pcastar$PCs

<table>
<thead>
<tr>
<th></th>
<th>[,1]</th>
<th>[,2]</th>
<th>[,3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-0.0071982563</td>
<td>1.015766e-01</td>
<td>-6.641151e-01</td>
</tr>
<tr>
<td>2</td>
<td>0.0012351239</td>
<td>9.894476e-01</td>
<td>1.319859e-01</td>
</tr>
<tr>
<td>3</td>
<td>0.0080964628</td>
<td>-8.959534e-02</td>
<td>7.217552e-01</td>
</tr>
<tr>
<td>4</td>
<td>-0.9969178902</td>
<td>8.043031e-05</td>
<td>1.901229e-02</td>
</tr>
<tr>
<td>5</td>
<td>-0.0488186528</td>
<td>-2.574058e-03</td>
<td>-4.499122e-02</td>
</tr>
<tr>
<td>6</td>
<td>-0.0407661533</td>
<td>9.624811e-04</td>
<td>-9.847082e-02</td>
</tr>
<tr>
<td>7</td>
<td>0.0076607688</td>
<td>3.075361e-02</td>
<td>3.672275e-02</td>
</tr>
<tr>
<td>8</td>
<td>0.0212358508</td>
<td>-1.484202e-02</td>
<td>2.073030e-02</td>
</tr>
<tr>
<td>28</td>
<td>0.0006744257</td>
<td>1.307122e-04</td>
<td>5.791225e-04</td>
</tr>
<tr>
<td>29</td>
<td>0.0003653182</td>
<td>3.169510e-04</td>
<td>-5.010227e-04</td>
</tr>
<tr>
<td>30</td>
<td>0.0066456248</td>
<td>3.086885e-03</td>
<td>1.011392e-02</td>
</tr>
<tr>
<td>31</td>
<td>0.0000918366</td>
<td>6.337891e-04</td>
<td>-1.065300e-03</td>
</tr>
<tr>
<td>32</td>
<td>-0.0002221122</td>
<td>-1.676717e-04</td>
<td>5.908053e-06</td>
</tr>
<tr>
<td>33</td>
<td>-0.0137127842</td>
<td>-9.519876e-03</td>
<td>9.274510e-04</td>
</tr>
</tbody>
</table>

R> myl1pcastar$PropL1Dispersion

[1] 0.3721951 0.3509983 0.2396433

R> myl1pca <- l1pca(mypcadata, projdim=3, iteration=20, center=TRUE, getScore=TRUE, getProjPoints=TRUE, dispersionExp=TRUE)
From the results, we can notice that the maximum L1-dispersion explained along the first three principal components is captured by PCA-L1, then L1-PCA^*. Table 2.7 compares the proportion of L1 dispersion explained by each method for first three PCs. The total L1-dispersion explained by L1-PCA^* is
<table>
<thead>
<tr>
<th>PC</th>
<th>PCA-L1</th>
<th>L1-PCA*</th>
<th>L1-PCA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.3442172</td>
<td>0.3721951</td>
<td>0.3280341</td>
</tr>
<tr>
<td>2</td>
<td>0.3095441</td>
<td>0.3509983</td>
<td>0.3048935</td>
</tr>
<tr>
<td>3</td>
<td>0.3030566</td>
<td>0.2396433</td>
<td>0.3024189</td>
</tr>
<tr>
<td>TOTAL</td>
<td>0.9568179</td>
<td>0.9628367</td>
<td>0.9353465</td>
</tr>
</tbody>
</table>

Table 2.7: Proportion of L1 dispersion explained by first 3 PCs

maximum among the three methods for microbiome data with L1-PCA explaining the least proportion.

The scatter plots using scores for each method, including L2-PCA, are shown in the Fig 2.7. Each dot represent a person with color denoting the type of bacteria with maximum percentage for that person. The clusters obtained from scatter plots can help in inferring the relationships between people colonized with (traditionally considered) healthy, less healthy and unhealthy bacterias.

The results of relationships between different bacterias will be used further in microbiome project to draw conclusions on the regional, genetic and ethnicity related occurrences of the bacteria.
Figure 2.5: Bacteria profiles of subjects
Figure 2.6: Steps of Data Collection
Figure 2.7: 3-D plots for different PCA methods.
Bibliography


[9] Q. Ke and T. Kanade. Robust l1 norm factorization in the presence of outliers and missing data by alternative convex programming. In *Proceedings*


Appendices
DESCRIPTION

Package: pcaL1
Version: 0.05
Date: 2011-01-01
Title: Implementation of 3 PCA methods using L1 norm
License: GPL-2

Author@R: c(person("Sapan Jot", "Developer", email="jots@vcu.edu"), person("J.P. Brooks", "Advisor"))

Author: Sapan Jot <jots@vcu.edu>, with contributions from J.P. Brooks <jpbrooks@vcu.edu>.

Maintainer: Sapan Jot <jots@vcu.edu>

Description: This package provides 3 different principal component analysis methods using L1 norm. First is L1-PCA* by Dr. J. P. Brooks, Dr. Dula and Dr. Boone. Second is PCA-L1 by N. Kwak and third is L1-PCA by Ke and Kanade.

Packaged: 2011-04-08 19:00:34 UTC; jpbrooks
NAMESPACE

elexportPattern ("^\[\^\:\[\]\]"")
UseDynLib(pcaL1, PcaL1_R, l1pca, l1pcastar)
C Code

type.h

src/type.h

#include "Clp_C_Interface.h"
#include <sys/times.h>
#include <stdio.h>
#include <assert.h>
#include <float.h>
#define PATHLENGTH 500 /* length of path name describing location of data file*/
#define OBJ_INIT 1000000.0 /* initial incumbent value */
#define VERBOSITY 0 /* 1− print projdim, rotation matrix, average error, 1 3− projected points in terms of original coords, 4−CPX screen output on, all objective values, best betas, projected points at each iteration, orthogonal direction, columns of rotation matrix at each iteration, initial SVD info 5−CPLEX log file, 6− write subproblem to a file, 7− write points to screen*/
#define NMAX 64 /* maximum possible block size, see dgeqrf.f in LAPACK */
#define PROJPTS 0 /* if 1, get projected q−dim points in terms of original coordinates */
#define SCORES 0 /* if 1, get scores, projected points in new coordinates */
#define PREV 0 /* if 1, get product of Vˆk matrices */
#define PREVBETA 1 /* if 1, get product of (Vˆk Beta) matrices */
#define EPSILON 0.0000001 /* check if objective is 0, check if wT == wTold */

/* input data */
struct entityinfo {
    int numentities; /*# of rows in your data*/
int numattributes; /*# of columns*/
double *points; /* actual data points in column major
  format— in a single array*/
}

typedef struct entityinfo ENTITYINFO, *ENTITYINFOptr;

/* main CPLEX variables (others are in cutcallbackinfo)
*/

struct solverinfo {
  struct Clp_Simplex *model;
  int status;
  char *errmsg;
  struct Clp_Simplex *modelU;
  struct Clp_Simplex *modelV;
};
typedef struct solverinfo SOLVERINFO, *SOLVERINFOptr;

/* problem info */

struct probleminfo {
  int rcnt; /*# of constraints in LP*/
  int nzcnt; /*number of coefficients to change*/
  int *matbeg; /* starting index of column*/
  int *matind; /* list of row–indices for each value in
  matval*/
  double *matval; /*coefficients for the linear
  constraints*/
  double *rhs; /*vector of RHS–0 for all of them*/
  int q; /* projected dimension*/
  int *betaind; /* index of betas*/
  int *eplusind; /* indices for e+*/
  int *eminusind; /* indices for e–*/
  double objective; /* stores the objective CLP returns*/
  int projdim; /* index for the loop–dimension we’re
  projecting into (k–1)*/
  int *bestdir; /* stores the best direction for every
  dimension*/
  int solstat; /* solution status – 0–optimal*/
  const double *beta; /* betas–coefficients for datapoints
  in constraint*/
  double *b; /* matrix of PC’s*/
  double minobjective; /* stores the best objective for k
  linear regression*/
  int numcols; /* # of columns for constraint matrix*/
  int i;
  int j;
int k;
int l; /* variable for loop on regression − j (article) */
int status; /* checking for the success of function */
double *obj; /* coefficients of objective function */
double *lb; /* lower bound for variables */
double *ub; /* upper bound for variables */
char **colname; /* variable names */
double *xpluslambda; /* Z−projected points on the subspace in terms of original co-ordinates */
double *xpluslambda2; /* copy of Z */
double *work; /* used in Fortran routine 'dgesvd' */
int lwork; /* used in Fortran routine 'dgesvd' */
double *S; /* eigenvalues of Z* */
double *VT; /* U matrix */
double *a; /* normalized beta */
double *preVjT; /* product of V matrix */
double *temppreVjT; /* used for obtaining the product of V matrices */
double *VBeta; /* product of VjT with Identity matrix (which has a row of betas for best direction dimension) */
double *preVBeta; /* transformation for finding the new score of a point */
double *temppreVBeta;
double *tempPC; /**/
double *projpoints; /* projected points in the subspace in terms of original co-ordinates */
int numprojdim; /* number of projection directions with "L1 variation" greater than 1 */
int numfactors; /* min (#of attributes, # of points) */
double *scores; /* stores scores−projected points in terms of new coordinates */
int getScores; /* 0 for no scores; default is 0 */
int getProjPoints; /* 0 for no projected point calculations; default is 0 */

/* variables for PCA− L1 */

int iter; /* T: no of iterations to find w(PC) */
double *polarity; /* polarities for each column in data matrix */
int dotConv; /* stores if w.x(i)=0 */
double *wT; /* Tth w */
double *wTOld; /* (T−1)th w */
double *PCs; /* all the w’s */
double *Xj; /* points for the Xj matrix – See Kwak*/
double innerprod; /* w^T x */
int convergent; /* is pcaL1 convergent */

/* for L1-PCA*/

double *V; /* rotation matrix */
double *Vtemp; /* temp V matrix*/
double *nv; /* normalized diagonal vector for V*/
double *U; /* scores matrix */
double *Utemp; /* temp U matrix*/
double *nu; /* normalized diagonal vector for U*/
double *sigma;
double tolerance; /* user defined epsilon*/
int iterations; /* user defined no. of iterations for algorithm*/
double maxdifference; /* max difference between elements of U and V between iterations*/
double *rhsL; /* lower bound for constraints */
double *rhsU; /* upper bound for constraints */
int *xinda; /* index of constraints when solving for U */
int *xindb; /* index of constraints when solving for U */
int *uind; /* index of variables when solving for U */
double *Usol; /* solution to problem when solving for U, including delta’s */
int **rowind; /* index of constraints when solving for V */
int **vind; /* index for variables when solving for V */
double *Vsol; /* solution to problem when solving for V, including lambda’s */
double udiff; /* checking for convergence of U */
double vdiff; /* checking for convergence of V */

}

typedef struct probleminfo PROBLEMINFO, *PROBLEMINFOptr;

PCA-L1

tscl/PcaL1Pkg.c

#include <stdlib.h>
#include <math.h>
#include "mrand.h"
#include "type.h"
#include <R.h>

int initialize(ENTITYINFOptr entityinfo, PROBLEMINFOptr probleminfo);
int polarityChk(ENTITYINFOptr entityinfo, PROBLEMINFOptr probleminfo);
int ChkConvergence(PROBLEMINFOptr probleminfo, ENTITYINFOptr entityinfo);

int solvePcaL1(ENTITYINFOptr entityinfo, PROBLEMINFOptr probleminfo)
{

    int numattributes = entityinfo->numattributes;
    int numentities = entityinfo->numentities;
    double *points = entityinfo->points;

    int status = probleminfo->status;
    double *wT = probleminfo->wT;
    double *PCs = probleminfo->PCs;
    int q = probleminfo->q;
    int i = probleminfo->i;
    int j = probleminfo->j;
    int k = probleminfo->k;
    int l = probleminfo->l;
    double innerprod = probleminfo->innerprod;

    int position;

    for (k = 0; k < q; ++k) {
        if (k!=0)/\*compute new points for Xj*/ {
            for (i = 0; i < numentities; ++i) {
                innerprod = 0.0;
                for (l = 0; l < numattributes; ++l) {
                    innerprod += wT[l] * points[numattributes*i+l];
                }
                for(j = 0; j < numattributes; ++j) {
                    position = numattributes*i + j;
                    points[position] = points[position] - wT[j] * innerprod;
                }
            }
        }
    }
}
status = initialize(entityinfo, probleminfo);
if(status) {
    fprintf(stderr, "Unable to initialize\n");
    return 1;
}
probleminfo->convergent = 0;
while (probleminfo->convergent == 0) {
    status = polarityChk(entityinfo, probleminfo);
    if(status) {
        fprintf(stderr, "Polarity Check failed\n");
        return 1;
    }
    status = ChkConvergence(probleminfo, entityinfo);
    if(status) {
        fprintf(stderr, "Convergence Check failed\n");
        return 1;
    }
}
for (j = 0; j < numattributes ; ++j) {
    PCs[numattributes * k + j] = wT[j]; /* save PCs*/
}
return 0;

int initialize(ENTITYINFOptr entityinfo, PROBLEMINFOptr
    probleminfo)
{
    int numentities = entityinfo->numentities;
    int numattributes = entityinfo->numattributes;
    double *points = entityinfo->points;

    int i = probleminfo->i;
    int j = probleminfo->j;
    double x;
    double xSum;
    double maxSum=0.0;
    int argmax;

    for (i=0; i < numentities; ++i) {
        xSum = 0;
        for (j=0; j<numattributes; ++j) {
            x = points[numattributes * i + j]; /* get the point*/
            xSum=xSum+x*x; /* create sum*/
        }
    }
xSum=sqrt(xSum);
if(maxSum < xSum)/*compare to the max*/
{
    maxSum=xSum; /*save the max sum*/
    argmax = i; /*save the observations*/
}
}
for (j = 0; j < numattributes; ++j){
    probleminfo->wT[j]=points[numattributes * argmax + j ]/maxSum;
}

probleminfo->iter = 0;
return 0;
}
int polarityChk(ENTITYINFOptr entityinfo , PROBLEMINFOptr probleminfo)
{
    double *polarity = probleminfo -> polarity;
    int iter = probleminfo -> iter;
    int i = probleminfo ->i;
    int j = probleminfo ->j;
    double dotProd = 0.0;
    double wTSum = 0.0;
    double normalizer;

    int numattributes = entityinfo ->numattributes;
    int numentities = entityinfo ->numentities;
    double *points = entityinfo ->points;

    probleminfo->dotConv=0;

    for (i = 0; i < numentities ; ++i){
        dotProd = 0.0;
        for (j = 0; j < numattributes ; ++j){
            dotProd += probleminfo->wT[j]*points[numattributes *
                i + j ];/*obtaining dot product of wT and ith observation */
        }
        if (dotProd < 0.0)
            polarity[i] = -1.0;
        else {
            if (dotProd >= 0.0){
                probleminfo->dotConv=1; /*saving for convergence
                    check*/
            }
        }
    }
polarity[i] = 1.0;
}
}
iter = iter + 1;
for(i = 0; i < numattributes; ++i){
    probleminfo->wTOld[i] = probleminfo->wT[i]; /* saving w(T-1)*/
    probleminfo->wT[i] = 0.0;
}
for (i = 0; i < numattributes; ++i){
    for (j = 0; j < numentities; ++j){
        probleminfo->wT[i] += polarity[j] * points[numattributes * j + i]; /* new wT*/
    }
}
for (j = 0; j < numattributes; ++j){
    wTSum += probleminfo->wT[j] * probleminfo->wT[j];
}

normalizer = sqrt(wTSum);
for (j = 0; j < numattributes; ++j){
    probleminfo->wT[j] = probleminfo->wT[j] / normalizer;
}
return 0;
}

int ChkConvergence(PROBLEMINFOptr probleminfo,
                     ENTITYINFOptr entityinfo)
{
    double *wTOld = probleminfo->wTOld;
    int j = probleminfo->j;
    int dotConv = probleminfo->dotConv;

    double wTSum = 0.0;
    double Normalizer = 0.0;

    int numattributes = entityinfo->numattributes;

    probleminfo->convergent = 1;

    for (j = 0; j < numattributes; ++j){
        if (probleminfo->wT[j] - wTOld[j] > (EPSILON)){
            probleminfo->convergent = 0;
        }
    }
    return 0;
}
if (dotConv == 1) /* if wT(Xi) = 0, then add a small non-zero random vector and do Polarity Check again */
{
    probleminfo->convergent = 0;
    for (j = 0; j < numattributes; ++j) {
        probleminfo->wT[j] = probleminfo->wT[j] + mrand(1);
        wTSum = wTSum + probleminfo->wT[j] * probleminfo->wT[j];
    }
    Normalizer = sqrt(wTSum);
    for (j = 0; j < numattributes; ++j) {
        probleminfo->wT[j] = probleminfo->wT[j] / Normalizer;
    }
    dotConv = 0; /* reset dotConv for next iteration */
}
return 0;

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <R.h>
#include "type.h"

int allocateMemoryPcaL1 (ENTITYINFOptr entityinfo, PROBLEMINFOptr probleminfo);
int solvePcaL1 (ENTITYINFOptr entityinfo, PROBLEMINFOptr probleminfo);

static void free_and_null (char **ptr);
void PcaL1_R (double *points, int *dataDim, int *q, double *PCs)
{
    int i;
    ENTITYINFO entityinfo;
    PROBLEMINFO probleminfo;

    int status = probleminfo.status;
    probleminfo.polarity = NULL;
    probleminfo.wT = NULL;

    for (i = 0; i < *dataDim; ++i) {
        double value = points[i];
        double iWeight = wT[i];

        if (value < 0) {
            probleminfo.polarity = NULL;
            probleminfo.wT[i] = NULL;
            dotConv = 0;
        } else {
            probleminfo.polarity = 1;
            probleminfo.wT[i] = 1;
            dotConv = 1;
        }
    }

    if (dotConv == 1) /* if wT(Xi) = 0, then add a small non-zero random vector and do Polarity Check again */
    {
        probleminfo->convergent = 0;
        for (j = 0; j < numattributes; ++j) {
            probleminfo->wT[j] = probleminfo->wT[j] + mrand(1);
            wTSum = wTSum + problenfeinfo->wT[j] * probleminfo->wT[j];
        }
        Normalizer = sqrt(wTSum);
        for (j = 0; j < numattributes; ++j) {
            probleminfo->wT[j] = probleminfo->wT[j] / Normalizer;
        }
        dotConv = 0; /* reset dotConv for next iteration */
    }
    return 0;
}
probleminfo.wTOld = NULL;
probleminfo.PCs = PCs;

entityinfo.numentities = dataDim[1];
entityinfo.numattributes = dataDim[0];

probleminfo.q = *q; /* desired number of PCs */
entityinfo.points = points; /* transpose of data matrix */

status = allocateMemoryPcaL1(&entityinfo, &probleminfo);
if (status) {
    fprintf(stderr, "Unable to allocate memory\n");
    goto TERMINATE;
}

status = solvePcaL1(&entityinfo, &probleminfo);
if (status) {
    fprintf(stderr, "Unable to solve. Terminating... or done\n");
    goto TERMINATE;
}

TERMINATE:

/* free_and_null ((char **) &entityinfo.points); */
free_and_null ((char **) &probleminfo.polarity);
free_and_null ((char **) &probleminfo.wT);
free_and_null ((char **) &probleminfo.wTOld);
/* free_and_null ((char **) &probleminfo.PCs); */

}  

class static void free_and_null (char **ptr)
{
    if( *ptr != NULL){
        free (*ptr);
        *ptr = NULL;
    }
}

int allocateMemoryPcaL1 (ENTITYINFOptr entityinfo,
PROBLEMINFOptr probleminfo) {
  int numentities = entityinfo->numentities;
  int numattributes = entityinfo->numattributes;

  probleminfo->polarity = (double *) malloc (numentities * sizeof(double));
  probleminfo->wT = (double *) malloc (numattributes * sizeof(double));
  probleminfo->wTOld = (double *) malloc (numattributes * sizeof(double));
  /* probleminfo->PCs = (double *) malloc (probleminfo->q * numentities * sizeof(double)); */

  return 0;
}

L1-PCA*

src/l1pcaStar.c
#include "Clp_C_Interface.h"
#include <stdlib.h>
#include <math.h>
#include <R.h>
#include "type.h"

int solveproblem (ENTITYINFOptr entityinfo, SOLVERINFOptr solverinfo, PROBLEMINFOptr probleminfo);

int initialSVD (ENTITYINFOptr entityinfo, PROBLEMINFOptr probleminfo);
int setupCLP (SOLVERINFOptr solverinfo);
int loadClpProblem (ENTITYINFOptr entityinfo, SOLVERINFOptr solverinfo, PROBLEMINFOptr probleminfo);
int changeBounds (SOLVERINFOptr solverinfo, PROBLEMINFOptr probleminfo, int l);
int optimize (SOLVERINFOptr solverinfo, PROBLEMINFOptr probleminfo);
int getBeta (SOLVERINFOptr solverinfo, ENTITYINFOptr entityinfo, PROBLEMINFOptr probleminfo, int l);
int getProjectedPoints (ENTITYINFOptr entityinfo, PROBLEMINFOptr probleminfo);
int dgesvd (char jobu, char jobvt, int m, int n, double * A, int lda, double *S, double *VT, int ldu, double *Umat, int ldvt, double *work, int lwork); /* SVD */
void dgemm (char transa, char transb, int m, int n, int k,
  double alpha, double *A, int lda, double *B, int ldb,
  double beta, double *C, int ldc); /* multiply A *B = C */
void dgemv (char trans, int m, int n, double alpha,
  double *A, int lda, double *x, int incx, double beta,
  double *y, int incy); /* multiply Ax = y */

int solveproblem (ENTITYINFOptr entityinfo, SOLVERINFOptr solverinfo, PROBLEMINFOptr probleminfo) {

  int status = probleminfo->status;
  int i = probleminfo->i;
  int j = probleminfo->j;
  int l = probleminfo->l;
  double minobjective = probleminfo->minobjective;
  int numprojdim = probleminfo->numprojdim;
  int numfactors = probleminfo->numfactors;

  int numentities = entityinfo->numentities;
  int numattributes = entityinfo->numattributes;
  numprojdim = 0;

  status = setupCLP (solverinfo); /* initialize CLP */
  if (status) {
    fprintf (stderr, "Error setting up CLP\n");
    return 1;
  }

  /* if numattributes > numentities, project into numentities space, then set numattributes = numentities and continue */
  if (numattributes > numentities) {
    status = initialSVD (entityinfo, probleminfo);
    if (status) {
      fprintf (stderr, "Error with initial SVD\n");
      return 1;
    }
  }

  for (probleminfo->projdim = numfactors - 1; probleminfo
    ->projdim > 0; --probleminfo->projdim) { /* the main loop */
    minobjective = (OBJ_INIT); /* objective of the best l1 regression */

54
Clp_setDualObjectiveLimit(solverinfo->model,
    minobjective);

status = loadClpProblem(entityinfo, solverinfo,
    probleminfo); /* set up columns/variables */
if (status) {
    fprintf(stderr, "Error with initial load of problem\n");
    return 1;
}

fprintf(stdout, "%d", probleminfo->projdim);
fflush(stdout);
if ((VERBOSITY) >= 1) {
    fprintf(stdout, "projdim %d\n", probleminfo->projdim);
    fflush(stdout);
}

for (l = 0; l <= probleminfo->projdim; ++l) { /* solve l1 regression for each direction, lth attribute is response */
    if ((VERBOSITY) >= 1) {
        fprintf(stdout, "l%d minobjective %f\n", l, minobjective);
        fflush(stdout);
    }

    status = changeBounds(solverinfo, probleminfo, l);
    /* set beta_l = -1, others unbounded */
    if (status) {
        fprintf(stderr, "Error changing beta bounds\n");
        return 1;
    }

    status = optimize(solverinfo, probleminfo); /* solve with Clp, get objective value */
    if (status) {
        fprintf(stderr, "Error solving l1 regression subproblem\n");
        return 1;
    }

    if (minobjective > probleminfo->objective) { /* determine if objective is best so far */
        probleminfo->bestdir[probleminfo->projdim] = 1;
    }
minobjective = probleminfo->objective;
probleminfo->minobjective = minobjective;
if ((VERBOSITY) >= 4) {
    fprintf (stdout, "new\_best\_dir\_%d\_new\_best\_objective\_%f\n", 1, probleminfo->objective);
    fflush (stdout);
}

status = getBeta(solverinfo, entityinfo, probleminfo, l); /* store coefficients of regression */
if (status) {
    fprintf (stderr, "Error storing beta's\n");
    return 1;
}

Clp_setDualObjectiveLimit(solverinfo->model, minobjective);
}

if (minobjective <= (EPSILON)) { /* if error is 0, go to next dimension */
    l = probleminfo->projdim + 1;
}
} /* end loop on l1 regressions */
if ((VERBOSITY) >= 1) {
    fprintf (stdout, "avg error %f\n", minobjective/((double) entityinfo->numentities));
    fflush (stdout);
}
if (minobjective/((double) entityinfo->numentities) < 1.0) {
    ++numprojdim;
}

if ((VERBOSITY) >= 4) {
    fprintf (stdout, "bestdir\_%d\_best\_objective\_%f\n",
            probleminfo->bestdir[probleminfo->projdim],
            minobjective);
    fflush (stdout);
}

status = getProjectedPoints(entityinfo, probleminfo);
/* get projected points */
if (status) {
    fprintf (stderr, "Error getting projected points, \n"}
int setupCLP (SOLVERINFOptr solverinfo) {
    solverinfo->model=Clp_newModel();
    Clp_setLogLevel (solverinfo->model, 0);
    return 0;
} /* end setupCLP */

int initialSVD (ENTITYINFOptr entityinfo , PROBLEMINFOptr
    probleminfo ) {
    int i = probleminfo->i;
    int j = probleminfo->j;
    int status = probleminfo->status;
    double *work = probleminfo->work;
    int lwork = probleminfo->lwork;
    double *S = probleminfo->S;
    double *VT = probleminfo->VT;

    int numattributes = entityinfo->numattributes;
    int numentities = entityinfo->numentities;
    double *points = entityinfo->points;

    if ((VERBOSITY) >= 7) {
        fprintf (stdout , "points\n");
        for (i = 0; i < numattributes; ++i) {
            for (j = 0; j < numentities; ++j) {
                fprintf (stdout , "%f", points [numattributes * j + i]);
            }
        }
        fprintf (stdout , "\n");
    }

    status = dgesvd ( 'A' , 'A' , numattributes , numentities ,
        points , numattributes , S , probleminfo->preVjT ,
        numattributes , VT , numentities , work , lwork); /* get
    A_q = preVjT, points gets destroyed here; The

    or done\n"; return 1;
}
columns of pre$V_jT$ define the new subspace */
if (status) { /* pre$V_jT$ is the first projdim columns
of U, derived by doing an SVD on points (which has
dimension numattributes $\times$ numentities). It's like
doing PCA on the transpose of the data matrix; $U*S$
has the scores */
  fprintf(stderr, "Error getting SVD, error \%d\n", status);
  fflush(stderr);
  return 1;
}
for (i = 0; i < numentities; ++i) {
  for (j = 0; j < numentities; ++j) {
    entityinfo->points[numentities * j + i] =
    probleminfo->VT[numentities * j + i] * S[i];
  }
}
return 0;
} /* end initial SVD */

int loadClpProblem (ENTITYINFOptr entityinfo,
  SOLVERINFOptr solverinfo, PROBLEMINFOptr probleminfo)
{
  int numcols = probleminfo->numcols;
  int i = probleminfo->i;
  int j = probleminfo->j;
  double *obj = probleminfo->obj;
  char **colname = probleminfo->colname;
  int rcnt = probleminfo->rcnt;
  int nzcnt = probleminfo->nzcnt;
  double *rhs = probleminfo->rhs;
  int *matind = probleminfo->matind;
  double *matval = probleminfo->matval;
  int *matbeg = probleminfo->matbeg;
  int projdim = probleminfo->projdim;
  double *points = entityinfo->points;

  int numentities = entityinfo->numentities;

  rcnt = numentities;
  for (i = 0; i < numentities; ++i) {

\texttt{\textbf{rhs}[i] = 0.0;}
}

\texttt{\textbf{nzcnt} = 0;}
\texttt{\textbf{numcols} = 0;}
\texttt{\textbf{for} (j = 0; j < (projdim + 1); ++j) \{}
\texttt{\textbf{matbeg}[\textbf{numcols}] = \textbf{nzcnt};}
\texttt{\textbf{probleminfo}->betaind[j] = \textbf{numcols};}
\texttt{\textbf{obj}[\textbf{numcols}] = 0.0;}
\texttt{\textbf{probleminfo}->lb[\textbf{numcols}] = -(COIN\_DBL\_MAX);}
\texttt{\textbf{probleminfo}->ub[\textbf{numcols}] = COIN\_DBL\_MAX;}
\texttt{\textbf{printf} (\textbf{colname}[\textbf{numcols}], ”beta\_%d”, j);}
\texttt{\textbf{for} (i = 0; i < \textbf{numentities}; ++i) \{}
\texttt{\textbf{if} (\textbf{points}[i*(projdim + 1) + j] != 0.0) \{}
\texttt{\textbf{matind}[\textbf{nzcnt}] = i;}
\texttt{\textbf{matval}[\textbf{nzcnt}] = \textbf{points}[i \times (projdim + 1) + j];}
\texttt{++\textbf{nzcnt};}
\texttt{\}}}
\texttt{\textbf{++\textbf{numcols}};}
\texttt{\}}
\texttt{\textbf{for} (i = 0; i < \textbf{numentities}; ++i) \{}
\texttt{\textbf{matbeg}[\textbf{numcols}] = \textbf{nzcnt};}
\texttt{\textbf{probleminfo}->eplusind[i] = \textbf{numcols};}
\texttt{\textbf{obj}[\textbf{numcols}] = 1.0;}
\texttt{\textbf{probleminfo}->lb[\textbf{numcols}] = 0.0;}
\texttt{\textbf{probleminfo}->ub[\textbf{numcols}] = (COIN\_DBL\_MAX);}
\texttt{\textbf{printf} (\textbf{colname}[\textbf{numcols}], ”eplus\_%d”, i);}
\texttt{\textbf{matind}[\textbf{nzcnt}] = i;}
\texttt{\textbf{matval}[\textbf{nzcnt}] = 1.0;}
\texttt{++\textbf{nzcnt};}
\texttt{++\textbf{numcols};}
\texttt{\}}
\texttt{\textbf{for} (i = 0; i < \textbf{numentities}; ++i) \{}
\texttt{\textbf{matbeg}[\textbf{numcols}] = \textbf{nzcnt};}
\texttt{\textbf{probleminfo}->eminusind[i] = \textbf{numcols};}
\texttt{\textbf{obj}[\textbf{numcols}] = 1.0;}
\texttt{\textbf{probleminfo}->lb[\textbf{numcols}] = 0.0;}
\texttt{\textbf{probleminfo}->ub[\textbf{numcols}] = (COIN\_DBL\_MAX); /* (CPX\_INFBOUND); */}
\texttt{\textbf{printf} (\textbf{colname}[\textbf{numcols}], ”eminus\_%d”, i);}
\texttt{\textbf{matind}[\textbf{nzcnt}] = i;}
\texttt{\textbf{matval}[\textbf{nzcnt}] = -1.0;}
\texttt{++\textbf{nzcnt};}
\texttt{++\textbf{numcols};}
\texttt{\}\}}
matbeg[numcols] = nzcnt;

Clp_loadProblem(solverinfo->model, numcols, rcnt, matbeg, matind, matval, probleminfo->lb, probleminfo->ub, obj, rhs, rhs);

return 0;
} /* end loadClpProblem */

int changeBounds(SOLVERINFOptr solverinfo, PROBLEMINFOptr probleminfo, int l)
{
    int j = probleminfo->j;
    int projdim = probleminfo->projdim;
    int *betaInd = probleminfo->betaInd;

    for (j = 0; j < (projdim + 1); ++j) {
        probleminfo->lb[betaInd[j]] = -(COIN_DBL_MAX);
        probleminfo->ub[betaInd[j]] = COIN_DBL_MAX;
    }

    probleminfo->lb[betaInd[1]] = -1.0;
    probleminfo->ub[betaInd[1]] = -1.0;

    Clp_chgColumnLower(solverinfo->model, probleminfo->lb);
    Clp_chgColumnUpper(solverinfo->model, probleminfo->ub);

    /* Clp_writeMps(solverinfo->model, "test.mps"); */
    return 0;
} /* end changeBounds */

int optimize(SOLVERINFOptr solverinfo, PROBLEMINFOptr probleminfo)
{
    int status = probleminfo->status;
    int solstat = probleminfo->solstat;

    status = Clp_dual(solverinfo->model, 0);
    if (status) {
        /* fprintf(stdout, "error solving dual simplex\n");
            return 1; */
    }

    solstat = Clp_status(solverinfo->model);

    return 0;
} /* end optimize */

/* end lpsol */
if (solstat != 0) {
    // printf (stdout, "solstat %d\n", solstat);
    return 1;*/
}

probleminfo->objective = Clp_getObjValue(solverinfo->model);
if ((VERBOSITY) >= 4) {
    fprintf (stdout, "objective_value %f\n", probleminfo->objective);
    fflush (stdout);
}

return 0;
} /* end optimize */

int getBeta (SOLVERINFOptr solverinfo, ENTITYINFOptr entityinfo, PROBLEMINFOptr probleminfo, int 1) {

    int i = probleminfo->i;
    /* int status = probleminfo->status; */
    int numfactors = probleminfo->numfactors;

    probleminfo->beta = Clp_getColSolution (solverinfo->model);

    if ((VERBOSITY) >= 4) {
        for (i = 0; i < numfactors; ++i) {
            fprintf (stdout, "beta[%d] %f\n", i, probleminfo->beta[i]);
            fflush (stdout);
        }
    }
    return 0;
} /* end getBeta */

int getProjectedPoints (ENTITYINFOptr entityinfo, PROBLEMINFOptr probleminfo) {

    int numattributes = entityinfo->numattributes;
    int numentities = entityinfo->numentities;
    double *points = entityinfo->points;

    int i = probleminfo->i;
    int j = probleminfo->j;
int status = probleminfo->status;
int projdim = probleminfo->projdim;
int *bestdir = probleminfo->bestdir;
double *xplslambda = probleminfo->xplslambda;
double *xplslambda2 = probleminfo->xplslambda2;
const double *beta = probleminfo->beta;
double *work = probleminfo->work;
int lwork = probleminfo->lwork;
double *S = probleminfo->S;
double *VT = probleminfo->VT;
double *a = probleminfo->a;
double *temppreVjT = probleminfo->temppreVjT;
double *temppreVBeta = probleminfo->temppreVBeta;
double *tempPC = probleminfo->tempPC;
int q = probleminfo->q;

for (i = 0; i < numentities; ++i) { /* get projected points, in terms of coordinates of current subspace */
xplslambda[i * (projdim + 1) + bestdir[projdim]] = 0.0;
for (j = 0; j <= projdim; ++j) {
    if (j != bestdir[projdim]) {
        xplslambda[i * (projdim + 1) + bestdir[projdim]] +=
            beta[j] * points[i * (projdim + 1) + j];
        xplslambda[i * (projdim + 1) + j] = points[i * (projdim + 1) + j];
    }
}
for (i = 0; i < numentities; ++i) {
    for (j = 0; j <= projdim; ++j) {
        xplslambda2[i * (projdim + 1) + j] = xplslambda[i * (projdim + 1) + j];
    }
}
if (((VERBOSITY) < 4) { /* get V^jT, xplslambda gets destroyed here; The columns of VjT define the new subspace */
    status = dgesvd ( 'A', 'N', projdim + 1, numentities,
        xplslambda, projdim + 1, S, probleminfo->VjT,
        projdim + 1, VT, numentities, work, lwork); /* get V^jT, xplslambda gets destroyed here; The columns of VjT define the new subspace */
    if (status) { /* V^jT is the first projdim columns of U, derived by doing an SVD on xplslambda (which has dimension projdim + 1 X numentities). It's like doing PCA on the transpose of the data matrix */

62
#define printf (stderr, "Error getting SVD, error %d\n", status);
fflush (stderr);
return 1;
}

else {
    status = dgesvd ( 'A', 'S', projdim + 1, numentities,
                    xpluslambda, projdim + 1, S, probleminfo->VjT,
                    projdim + 1, VT, numattributes, work, lwork); /* get
VjT, xpluslambda gets destroyed here; The
columns of VjT define the new subspace */
    if (status) { /* VjT is the first projdim columns of
U, derived by doing an SVD on xpluslambda (which
has dimension projdim + 1 X numentities). It's
like doing PCA on the transpose of the data matrix */
        printf (stderr, "Error getting SVD, error %d\n", status);
        fflush (stderr);
        return 1;
    }
    for (i = 0; i < projdim + 1; ++i) {
        a[i] = probleminfo->VjT[projdim * (projdim + 1) + i];
        /* saving normalized beta*/
    }
    for (i = 0; i < numattributes; ++i) {
        tempPC[i] = 0.0;
    }
    if ((VERBOSITY) >= 4) {
        for (i = 0; i < numattributes * (projdim + 1); ++i) {
            fprintf (stdout, "preVjT[%d] %f\n", i, probleminfo
->preVjT[i]); /* product of all previous VjT */
        }
    }
    if (probleminfo->getProjPoints == 1) {
        if (projdim == q) { /* print projected points in
original coordinates*/
            for (i = 0; i < numattributes * numentities; ++i) {
                probleminfo->projpoints[i] = 0.0;
            }
            dgemm ('N', 'N', numattributes, numentities, projdim
+ 1, 1.0, probleminfo->preVjT, numattributes,
xpluslambda2, projdim + 1, 1.0, probleminfo->projpoints, numattributes); /* get projected points in original coordinates by multiplying preVjT by xpluslambda, projected points are columns */

```
dgemv ( 'N', numattributes, projdim + 1, 1.0, probleminfo->preVjT, numattributes, a, 1, 1.0, tempPC, 1): /* multiply old VjT's by a, to get projdim th PC */
for (j = 0; j < numattributes; ++j) {
    probleminfo->b[numattributes * projdim + j] = tempPC[j];
}
/* new VBeta times previous VBeta */
for (i = 0; i < numattributes * (projdim + 1); ++i) {
    temppreVBeta[i] = 0.0;
}
for (i = 0; i < projdim; ++i) {
    for (j = 0; j < projdim + 1; ++j) {
        if (j != bestdir[projdim]) {
        } else {
            probleminfo->VBeta[projdim * j + i] = 0.0;
        }
    }
}
```
dgemm ('N', 'N', projdim, numattributes, projdim + 1, 1.0, probleminfo->VBeta, projdim, probleminfo->preVBeta, projdim + 1, 0.0, temppreVBeta, projdim);
/* get preVBeta = projdim by numattributes */

for (i = 0; i < numattributes * projdim; ++i) {
    probleminfo->preVBeta[i] = temppreVBeta[i];
}
/* previous VjT’s times VjT */
for (i = 0; i < numattributes * (projdim + 1); ++i) {
    temppreVjT[i] = 0.0;
}
dgemm ('N', 'N', numattributes, projdim, projdim + 1, 1.0, probleminfo->preVjT, numattributes, probleminfo->VjT, projdim + 1, 1.0, temppreVjT, numattributes);
/* get preVjT = numattributes by projdim*/

for (i = 0; i < numattributes * projdim; ++i) {
    probleminfo->preVjT[i] = temppreVjT[i];
}
if (projdim == 1) { /* get 1st PC */
    for (i = 0; i < numattributes; ++i) {
        probleminfo->b[i] = probleminfo->preVjT[i];
    }
}

for (i = 0; i < projdim * numentities; ++i) {
    entityinfo->points[i] = 0.0;
}
dgemm ('T', 'N', projdim, numentities, projdim + 1, 1.0, probleminfo->VjT, projdim + 1, xpluslambda2, projdim + 1, 1.0, entityinfo->points, projdim);
/* get new points in new coordinates by multiplying Vj (transpose of VjT) by xpluslambda */

/* get scores— projected points in terms of new coordinates */
if ((probleminfo->getScores == 1) && (projdim == q)) {
  for (i = 0; i < numentities; ++i) {
    for (j = 0; j < projdim; ++j) {
      probleminfo->scores[i * projdim + j] = entityinfo->points[i * projdim + j];
    }
  }
}
return 0;
} /* end getProjectedPoints */

int dgesvd (char jobu, char jobvt, int m, int n, double *A, int lda, double *S, double *VT, int ldu, double *Umat, int ldvt, double *work, int lwork) { /* SVD */
  extern void dgesvd_(const char *jobup, const char *jobvtp, const int *mp, const int *np, double *A, const int *ldap, double *S, double *U, const int *ldup, double *Umat, int *ldvtp, double *work, int *lworkp, int *infop);

  int info;
  dgesvd_(&jobu, &jobvt, &m, &n, A, &lda, S, VT, &ldu, Umat, &ldvt, work, &lwork, &info);
  return info;
} /* end dgesvd, SVD */

void dgemm (char transa, char transb, int m, int n, int k, double alpha, double *A, int lda, double *B, int ldb, double beta, double *C, int ldc) { /* multiply A *B = C */
  extern void dgemm_(const char *transap, const char *transbp, const int *mp, const int *np, const int *kp, double *alphap, double *A, const int *ldap, double *B, const int *ldbp, const double *betap, double *C, const int *ldcp);
  dgemm_(&transa, &transb, &m, &n, &k, &alpha, A, &lda, B, &ldb, &beta, C, &ldc);


```c
void dgemv (char trans, int m, int n, double alpha, double *A, int lda, double *x, int incx, double beta, double *y, int incy) { /* multiply Ax = y */
    extern void dgemv_(const char *transp, const int *mp, const int *np, double *alphap, double *A, const int *ldap, double *x, const int *incxp, const double *betap, double *y, const int *incy);
    dgemv_(&trans, &m, &n, &alpha, A, &lda, x, &incx, &beta, y, &incy);
} /* end dgemv, multiply Ax */
```

```c
int allocateMemory (ENTITYINFOptr entityinfo,
    SOLVERINFOptr solverinfo, PROBLEMINFOptr probleminfo);

int solveproblem (ENTITYINFOptr entityinfo, SOLVERINFOptr solverinfo, PROBLEMINFOptr probleminfo);
/* int solvePcaL1(IOINFOptr ioinfo, ENTITYINFOptr entityinfo, PROBLEMINFOptr probleminfo); */

static void
free_and_null (char **ptr);

void l1pcaStar (double *points, int *dataDim, int *q, int *getScores, int *getProjPoints, double *PCs, double *Scores, double *projpoints)
{
    ENTITYINFO entityinfo;
    SOLVERINFO solverinfo;
    PROBLEMINFO probleminfo;

    int status = probleminfo.status;

    status = 0;
```
solverinfo.model = NULL;
probleminfo.betaind = NULL;
probleminfo.eplusind = NULL;
probleminfo.eminusind = NULL;
probleminfo.beta = NULL;
probleminfo.bestdir = NULL;
probleminfo.xpluslambda = NULL;
probleminfo.xpluslambda2 = NULL;
probleminfo.S = NULL;
probleminfo.VT = NULL;
probleminfo.VjT = NULL;
probleminfo.preVjT = NULL;
probleminfo.temppreVjT = NULL;
probleminfo.VBeta = NULL;
probleminfo.preVBeta = NULL;
probleminfo.temppreVBeta = NULL;
probleminfo.tempPC = NULL;
probleminfo.rhs = NULL;
probleminfo.matbeg = NULL;
probleminfo.matind = NULL;
probleminfo.obj = NULL;
probleminfo.lb = NULL;
probleminfo.ub = NULL;
probleminfo.colname = NULL;
probleminfo.a = NULL;
probleminfo.b = PCs;
probleminfo.projpoints = projpoints;
probleminfo.scores = Scores;
probleminfo.work = NULL;

/* probleminfo.scores = NULL; */
probleminfo.getScores = *getScores;
probleminfo.getProjPoints = *getProjPoints;

entityinfo.numerities = dataDim[1];
extityinfo.numattributes = dataDim[0];

if (entityinfo.numattributes <= entityinfo.numerities)
    {
        probleminfo.numfactors = entityinfo.numattributes;
    }
else
    {
        probleminfo.numfactors = entityinfo.numerities;
    }

68
probleminfo.q = *q; /* desired number of PCs */

entityinfo.points = points; /* transpose of data matrix */

if ((VERBOSITY) >= 2) {
    fprintf(stdout, "n%d
", dataDim[1], dataDim [0]);
    fprintf(stdout, "getScores%d
", *getScores, *getProjPoints);
    fprintf(stdout, "getScores%d
", probleminfo.getScores, probleminfo.getProjPoints);
    fprintf(stdout, "numfactors%d
", probleminfo.numfactors, probleminfo.q);
}

status = allocateMemory(&entityinfo, &solverinfo, &probleminfo); /* at the end of this file */

if (status) {
    fprintf(stderr, "Unable to allocate memory\n");
    goto TERMINATE;
}

status = solveproblem(&entityinfo, &solverinfo, &probleminfo); /* in l1pcaStar.c */

if (status) {
    fprintf(stderr, "Unable to solve or done\n");
    goto TERMINATE;
}

/* PCs = probleminfo.b;
Scores = probleminfo.scores;
projpoints = probleminfo.projpoints;*/

TERMINATE:

free_and_null ((char **) &probleminfo.betaind);
free_and_null ((char **) &probleminfo.eplusind);
free_and_null ((char **) &probleminfo.eminusind);
/* free_and_null ((char **) &probleminfo.beta);*/
free_and_null ((char **) &probleminfo.bestdir);
free_and_null ((char **) &probleminfo.xpluslambda);
free_and_null ((char **) &probleminfo.xpluslambda2);
free_and_null ((char **) &probleminfo.S);
free_and_null ((char **) &probleminfo.VT);
free_and_null ((char **) &probleminfo.VjT);
free_and_null ((char **) &probleminfo.preVjT);
free_and_null ((char **) &probleminfo.temppreVjT);
free_and_null ((char **) &probleminfo.VBeta);
free_and_null ((char **) &probleminfo.preVBeta);
free_and_null ((char **) &probleminfo.temppreVBeta);
free_and_null ((char **) &probleminfo.rhs);
free_and_null ((char **) &probleminfo.matbeg);
free_and_null ((char **) &probleminfo.matval);
free_and_null ((char **) &probleminfo.matind);
free_and_null ((char **) &probleminfo.obj);
free_and_null ((char **) &probleminfo.lb);
free_and_null ((char **) &probleminfo.ub);

for (probleminfo.i = 0; probleminfo.i < probleminfo.numfactors + 2 * entityinfo.numentities; ++
probleminfo.i) {
    free_and_null ((char **) &probleminfo.colname[probleminfo.i]);
}
free_and_null ((char **) &probleminfo.colname);
free_and_null ((char **) &probleminfo.a);
free_and_null ((char **) &probleminfo.work);

if ( solverinfo.model != NULL ) {
    Clp_deleteModel (solverinfo.model);
    if ( status ) {
        fprintf (stderr, "Clp_delete failed, error code %d
            .\n", status);
    }
}

static void
free_and_null (char **ptr)
{
    if ( *ptr != NULL ) {
        free (*ptr);
        *ptr = NULL;
    }
} /* END free_and_null */

int allocateMemory (ENTITYINFOptr entityinfo, 
    SOLVERINFOptr solverinfo, PROBLEMINFOptr probleminfo)
{
    int numentities = entityinfo->numentities;
int numattributes = entityinfo->numattributes;

int i = probleminfo->i;
int j = probleminfo->j;
int numfactors = probleminfo->numfactors;

/* initialize solution vectors */
/* problenfo->b = (double **) malloc (numattributes * sizeof (double *));*/
for (j = 0; j < numattributes; ++j) {
    problenfo->b[j] = (double *) malloc (numattributes * sizeof (double));
}

/* allocate memory for best regression */
/* problenfo->beta = (double *) malloc ((numfactors + 2*numentities) * sizeof (double));*/
/* keep track of best linear regression */
probleminfo->bestdir = (int *) malloc (numfactors * sizeof (int)); /* to keep track of projections */

/* allocate memory for columns */
probleminfo->betaind = (int *) malloc (numfactors * sizeof (int));
probleminfo->eplusind = (int *) malloc (numentities * sizeof (int));
probleminfo->eminusind = (int *) malloc (numentities * sizeof (int));

probleminfo->obj = (double *) malloc (((numfactors + 2*numentities) * sizeof (double)));
probleminfo->lb = (double *) malloc (((numfactors + 2*numentities) * sizeof (double)));
probleminfo->ub = (double *) malloc (((numfactors + 2*numentities) * sizeof (double)));
probleminfo->colname = (char **) malloc (((numfactors + 2*numentities) * sizeof (char *)));
for (i = 0; i < numfactors + 2*numentities; ++i) {
    probleminfo->colname[i] = (char *) malloc (20 * sizeof (char));
}

/* allocate memory for constraints */
probleminfo->rhs = (double *) malloc (numentities * sizeof (double));
probleminfo->matbeg = (int *) malloc ((numfactors + 2 * numentities + 1) * sizeof (int));
probleminfo->matind = (int *) malloc ((numfactors * numentities + 2 * numentities) * sizeof (int));
probleminfo->matval = (double *) malloc ((numfactors * numentities + 2 * numentities) * sizeof (double));

/* allocate memory for chgbds—no more changing one bound at a time */

/* allocate memory for projected points */
probleminfo->xpluslambda = (double *) malloc (numentities * numfactors * sizeof (double));
probleminfo->xpluslambdad = (double *) malloc (numentities * numfactors * sizeof (double));
probleminfo->work = (double *) malloc (numentities * (NNMAX) * sizeof (double));
probleminfo->lwork = numentities * (NNMAX);
probleminfo->S = (double *) malloc (numattributes * sizeof (double));
probleminfo->VjT = (double *) malloc (numattributes * numattributes * sizeof (double));
probleminfo->VT = (double *) malloc (numentities * numentities * sizeof (double));
probleminfo->a = (double *) malloc (numfactors * sizeof (double));
probleminfo->VBeta = (double *) malloc (numattributes * numattributes * sizeof (double));
probleminfo->preVjT = (double *) malloc (numattributes * numattributes * sizeof (double));
probleminfo->preVBeta = (double *) malloc (numattributes * numattributes * sizeof (double));

for (i = 0; i < numattributes; ++i) {
  for (j = 0; j < numattributes; ++j) {
    if (i != j) {
      probleminfo->preVjT[i * numattributes + j] = 0.0;
      probleminfo->preVBeta[i * numattributes + j] = 0.0;
    }
    else {
      probleminfo->preVjT[i * numattributes + j] = 1.0;
      probleminfo->preVBeta[i * numattributes + j] = 1.0;
    }
  }
}
L1-PCA

src/l1pca.c

#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "type.h"
#include "mrand.h"

/* int initializeV (ENTITYINFOptr entityinfo,
    PROBLEMINFOptr probleminfo); */
int setupCLPforL1Pca (SOLVERINFOptr solverinfo);
int solveforU (ENTITYINFOptr entityinfo, SOLVERINFOptr solverinfo,
    PROBLEMINFOptr probleminfo);
int solveforV (ENTITYINFOptr entityinfo, SOLVERINFOptr solverinfo,
    PROBLEMINFOptr probleminfo);
int normalize (ENTITYINFOptr entityinfo, PROBLEMINFOptr probleminfo);
int dgesvd1 (char jobu, char jobvt, int m, int n, double *
    A, int lda, double *S, double *U, int ldu, double *VT,
    int ldvt, double *work, int lwork); /* SVD */
int solveL1Pca (ENTITYINFOptr entityinfo, SOLVERINFOptr
    solverinfo, PROBLEMINFOptr probleminfo);

int solveL1Pca (ENTITYINFOptr entityinfo, SOLVERINFOptr
    solverinfo, PROBLEMINFOptr probleminfo) {
    int status = probleminfo->status;

    status = setupCLPforL1Pca (solverinfo); /* initialize 
    CLP */
if (status) {
    fprintf(stderr, "Error setting up CLP\n");
    return 1;
}

/* status = initializeV(entityinfo, probleminfo); */
/* set V to a random matrix */
/* if (status) {
    fprintf(stderr, "Error initializing V\n");
    return 1;
} */
probleminfo->maxdifference = (COIN_DBL_MAX);

probleminfo->iter = probleminfo->iterations;
fprintf(stderr, "probleminfo->iterations:%d\n",
        probleminfo->iterations);
while (((probleminfo->iter > 0) && (probleminfo->
        maxdifference > probleminfo->tolerance)) {

    /* fprintf(stderr, "probleminfo->maxdifference %f\n",
            probleminfo->maxdifference); */
    probleminfo->maxdifference = 0.0;
    status = solveforU(entityinfo, solverinfo, 
        probleminfo); /* solve for U, matrix of scores */
    if (status) {
        fprintf(stderr, "Error solving for U\n");
        return 1;
    }
    /* solve for V, rotation matrix */
    status = solveforV(entityinfo, solverinfo, 
        probleminfo); /* solve for V, rotation matrix */
    if (status) {
        fprintf(stderr, "Error solving for V\n");
        return 1;
    }
    status = normalize(entityinfo, probleminfo);
    if(status){
        fprintf(stderr, "Error normalizing\n");
        return 1;
    }
    fprintf(stderr, "probleminfo->iter %d\n",
            probleminfo->iter);
    --probleminfo->iter;
}
fprintf(stderr, "number of iterations:%d\n",
        probleminfo->iter);
return 0;
} /*end solveproblem */

int setupCLPforL1PCA (SOLVERINFOptr solverinfo) {
  solverinfo->modelU=Clp_newModel();
  solverinfo->modelV=Clp_newModel();

  Clp_setLogLevel(solverinfo->modelU, 0);
  Clp_setLogLevel(solverinfo->modelV, 0);
  return 0;
} /* end setupCLP */

/* int initialize V(ENTITYINFOptr entityinfo,
 PROBLEMINFOptr probleminfo) {
 int j = probleminfo->j;
 int k = probleminfo->k;
 int q = probleminfo->q;
 int numattributes = entityinfo->numattributes;*/

/* initialize V to a random matrix; another option
 would be SVD */
/*for (k = 0; k < numattributes; ++k) {
 for (j = 0; j < q; ++j) {
 probleminfo->V[numattributes*j+k] = mrand(1);
 }
*/
/*for (k=0; k<q; ++k) {
 probleminfo->sigma[k] = 1.0;
 }*/
/*fprintf(stderr, "\n");
 return 0;
} */ /* end initialize V */

int solveforU (ENTITYINFOptr entityinfo, SOLVERINFOptr solverinfo, PROBLEMINFOptr probleminfo) {
 int status = probleminfo->status;
 int numcols = probleminfo->numcols;
 int i = probleminfo->i;
 int j = probleminfo->j;
 int k = probleminfo->k;
 int q = probleminfo->q;
 double *obj = probleminfo->obj;
 double *lb = probleminfo->lb;
 double *ub = probleminfo->ub;
 int rcnt = probleminfo->rcnt;
int nzcnt = probleminfo->nzcnt;
double *rhsL = probleminfo->rhsL;
double *rhsU = probleminfo->rhsU;
int *matind = probleminfo->matind;
double *matval = probleminfo->matval;
int *matbeg = probleminfo->matbeg;
int *xinda = probleminfo->xinda;
int *xindb = probleminfo->xindb;
double *initV = probleminfo->initV;
double *V = probleminfo->V;
int solstat = probleminfo->solstat;
int *uind = probleminfo->uind;

const double *Usol;
double *points = entityinfo->points;
int numentities = entityinfo->numentities;
int numattributes = entityinfo->numattributes;

double temp = 0.0;
double temp1 = 0.0;

/* add rows */
for (i = 0; i < numentities; ++i) {
    rcnt = 0;
    for (j = 0; j < numattributes; ++j) {
        xinda[j] = rcnt;
        rhsL[rcnt] = -(COIN_DBL_MAX);
        rhsU[rcnt] = points[i*numattributes + j];
        ++rcnt;
        rhsL[rcnt] = points[i*numattributes + j];
        rhsU[rcnt] = (COIN_DBL_MAX);
        xindb[j] = rcnt;
        ++rcnt;
    }

    nzcnt = 0;
    numcols = 0;
    /* add u i variables */
    for (k = 0; k < q; ++k) {
        matbeg[numcols] = nzcnt;
        obj[numcols] = 0.0;
        lb[numcols] = -(COIN_DBL_MAX);
        ub[numcols] = COIN_DBL_MAX;
        uind[k] = numcols;
        for (j = 0; j < numattributes; ++j) {
            matind[nzcnt] = xinda[j];
            if (probleminfo->iter != probleminfo->iterations)
            matval[nzcnt] = V[numattributes*k + j];
            ++nzcnt;
        }
    }

    // rest of the code...
matind[nzcnt] = xindb[j];
matval[nzcnt] = V[numattributes*k +j];

} else{
  matval[nzcnt] = initV[numattributes*k +j];
  ++nzcnt;
  matind[nzcnt] = xindb[j];
  matval[nzcnt] = initV[numattributes*k +j];
  /*printf(stdout, "initV[%d] %f\n", numattributes*k+j, V[numattributes*k+j]);*/
}

matbeg[numcols] = nzcnt;
for (j = 0; j < numattributes; ++j) {
  matbeg[numcols] = nzcnt;
  obj[numcols] = 1.0;
  lb[numcols] = 0.0;
  ub[numcols] = COIN_DBL_MAX;
  matind[nzcnt] = xinda[j];
  matval[nzcnt] = -1.0;
  ++nzcnt;
  matind[nzcnt] = xindb[j];
  matval[nzcnt] = 1.0;
  ++nzcnt;
  ++numcols;
}
matbeg[numcols] = nzcnt;

Clp_loadProblem(solverinfo->modelU, numcols, rcnt,
      matbeg, matind, matval, lb, ub, obj, rhsL, rhsU);

status = Clp_dual(solverinfo->modelU, 0);
if (status) {
      printf(stdout, "error\nsolving\nfor U\n");;
      return 1;
}

solstat=Clp_status(solverinfo->modelU);
if (solstat != 0) {

printf (stdout, "solstat %d\n", solstat);
  return 1;
}
/* objval = Clp_getObjValue(solverinfo->modelU);
printf(stdout, "obj val in U %f\n", objval);*/
Usol = Clp_getColSolution(solverinfo->modelU);
for (k = 0; k < q; ++k) {
  probleminfo->Utemp[k*numentities+i] = Usol[uind[k]];
  /* printf(stderr, "SolU i %d k %d U %f\n", i, k,
    probleminfo->Utemp[k*numentities + i]);*/
}
for (i = 0; i < numentities; ++i)
  for (j = 0; j < numattributes; ++j)
    temp = 0.0;
  for (k = 0; k < q; ++k) {
    temp += (V[numattributes*k + j]* probleminfo->
      Utemp[k*numentities+i]);
  }
  temp1 +=fabs(points[i*numattributes+j]-temp);
}
printf(stdout, "obj val in U %f\n", temp1);
return 0;
} /* end solveforU */

int solveforV (ENTITYINFOptr entityinfo, SOLVERINFOptr solverinfo, PROBLEMINFOptr probleminfo) {
  int status = probleminfo->status;
  int numcols = probleminfo->numcols;
  int i = probleminfo->i;
  int j = probleminfo->j;
  int k = probleminfo->k;
  int q = probleminfo->q;
  double *obj = probleminfo->obj;
  double *lb = probleminfo->lb;
  double *ub = probleminfo->ub;
  int rcnt = probleminfo->rcnt;
  int nzcnt = probleminfo->nzcnt;
  double *rhsL = probleminfo->rhsL;
  double *rhsU = probleminfo->rhsU;
  int *matind = probleminfo->matind;
  double *matval = probleminfo->matval;
  int *matbeg = probleminfo->matbeg;
  printf(stdout, "solstat %d\n", solstat);
  return 1;
}
int **rowind = probleminfo->rowind;
double *U = probleminfo->U;
int solstat = probleminfo->solstat;
int **vind = probleminfo->vind;
const double *Vsol;
double *Utemp = probleminfo->Utemp;
double *points = entityinfo->points;
int numentities = entityinfo->numentities;
int numattributes = entityinfo->numattributes;
int iter = probleminfo->iter;
int iterations = probleminfo->iterations;
double objval = 0.0;

/* add rows */
rcnt = 0;
for (i = 0; i < numentities; ++i) {
    for (j = 0; j < numattributes; ++j) {
        rowind[i][j] = rcnt;
        rhsL[rcnt] = 0.0 - points[i*numattributes + j];
        rhsU[rcnt] = 0.0 - points[i*numattributes + j];
        ++rcnt;
    }
}
nzcnt = 0;
numcols = 0;

/* add lambda variables */
for (i = 0; i < numentities; ++i) {
    for (j = 0; j < numattributes; ++j) {
        matbeg[numcols] = nzcnt;
        obj[numcols] = 1.0;
        lb[numcols] = 0.0;
        ub[numcols] = COIN_DBL_MAX;
        matind[nzcnt] = rowind[i][j];
        matval[nzcnt] = 1.0;
        ++nzcnt;
        ++numcols;
        matbeg[numcols] = nzcnt;
        obj[numcols] = 1.0;
        lb[numcols] = 0.0;
        ub[numcols] = COIN_DBL_MAX;
        matind[nzcnt] = rowind[i][j];
        matval[nzcnt] = -1.0;
        ++nzcnt;
        ++numcols;
    }
}

/* add V variables */
for (j = 0; j < numattributes; ++j) {
    for (k = 0; k < q; ++k) {
        matbeg[numcols] = nzcnt;
        vind[j][k] = numcols;
        lb[numcols] = -(COIN_DBL_MAX);
        ub[numcols] = COIN_DBL_MAX;
        obj[numcols] = 0.0;
        for (i = 0; i < numentities; ++i) {
            matind[nzcnt] = rowind[i][j];
            matval[nzcnt] = 0.0 - Utemp[k*numentities + i];
            /* if (iter != iterations) { 
                matval[nzcnt] = 0.0 - U[k*numentities + i]; 
            } */
            ++nzcnt;
        }
        ++numcols;
    }
    matbeg[numcols] = nzcnt;

    Clp_loadProblem(solverinfo->modelV, numcols, rcnt,
                    matbeg, matind, matval, lb, ub, obj, rhsL, rhsU);
    status = Clp_dual(solverinfo->modelV, 0);
    if (status) {
        fprintf (stdout, "error _ solving _ dual _ simplex _ for _ V\n"
                 );
        return 1;
    }

    solstat = Clp_status(solverinfo->modelV);
    if (solstat != 0) {
        fprintf (stdout, "solstat _%d\n", solstat);
        return 1;
    }

    objval = Clp_getObjValue(solverinfo->modelV);
    fprintf (stdout, "obj _ val _ for _ V\n", objval);
    Vsol = Clp_getColSolution (solverinfo->modelV);
    for (j = 0; j < numattributes; ++j) {
        for (k = 0; k < q; ++k) {
            probleminfo->Vtemp[numattributes*k+j] = Vsol[vind[j][k]];
        }
    }
}
return 0;

} /* end solveforV */

int normalize(ENTITYINFOptr entityinfo, PROBLEMINFOptr probleminfo) {

double *nv = probleminfo->nv;
/* double *nu = probleminfo->nu; */
double *sigma = probleminfo->sigma; /*
int i = probleminfo->i;
int j = probleminfo->j;
int k = probleminfo->k;
int numattributes = entityinfo->numattributes;
int numentities = entityinfo->numentities;
double udiff = probleminfo->udiff;
int iter = probleminfo->iter;
int iterations = probleminfo->iterations;
double vdiff = probleminfo->vdiff;
double *Vtemp = probleminfo->Vtemp;
double *Utemp = probleminfo->Utemp;
int q = probleminfo->q;

double temp2 = 0.0;

dobjval = 0.0;

double objval = 0.0;

for (i=0; i < q; ++i) {/* calculate nv and nu*/
    nv[i] = 0.0;
    for (j=0; j < numattributes; ++j){
        nv[i] += Vtemp[i*numattributes+j] * Vtemp[i*numattributes+j];
        /* printf(stderr, "nv[%d] %f\n", i, nv[i]); */
    }
    /*nu[i] = 0.0;
    for(j=0; j < numentities; ++j){
        nu[i] += Utemp[i*numentities+j] * Utemp[i*numentities+j];
        printf(stderr, "nu[%d] %f\n", i, nu[i]);
    }*/
}

sigma[i] = nv[i] * sigma[i] * nu[i];

printf(stderr, "sigma[%d] %f \n", i, sigma[i]);
}

for (i=0; i < q; ++i){
    for (j=0; j < numattributes; ++j){
        if (nv[i] > (EPSILON)) {
            Vtemp[i*numattributes+j] = Vtemp[i*numattributes+
j] * (1/sqrt(nv[i]));

} /* fprintf (stderr, "i %d j %d V %f\n", i, j,
        probleminfo->Vtemp[i*numattributes + j]);*/
/* probleminfo->Vtemp[i * numattributes + j] =
    probleminfo->Vtemp[i * numattributes + j] * sqrt
    (sigma[i]);
fprintf (stderr, "VSigma i %d j %d V %f\n", i, j,
    probleminfo->Vtemp[i*numattributes + j]); */

} /* calculating maxdifference after V*/
vdiff = 0.0;
for (j = 0; j < numattributes; ++j) {
    for (k = 0; k < q; ++k) {
        vdiff = fabs(Vtemp[numattributes*k+j] -
        probleminfo->V[numattributes*k+j]);
        /* fprintf(stderr, "vdiff %f\n", vdiff);*/
        if (vdiff > probleminfo->maxdifference)
            probleminfo->maxdifference = vdiff;
        probleminfo->V[numattributes*k+j] = Vtemp[
            numattributes*k+j];
        /* fprintf(stderr, "normalized V[%d] %f \n",
            numattributes*k+j , probleminfo->V[
            numattributes* k+j]); */
    }
    /*for(j = 0; j < numentities; ++j){
        probleminfo->Utemp[i * numentities + j] = Utemp[i *
            numentities + j] * (1/nu[i]);
        fprintf (stderr, "i %d j %d U %f\n", i, j,
            probleminfo->Utemp[i*numentities + j]);
        probleminfo->Utemp[i * numentities + j] =
            probleminfo->Utemp[i * numentities + j] * sqrt(
            sigma[i]);
        fprintf (stderr, "USigma i %d j %d U %f\n", i, j,
            probleminfo->Utemp[i*numentities + j]);
    }*/
}
/*for(j = 0; j < numentities; ++j){
    probleminfo->Utemp[i * numentities + j] = Utemp[i *
        numentities + j] * (1/nu[i]);
    fprintf (stderr, "i %d j %d U %f\n", i, j,
        probleminfo->Utemp[i*numentities + j]);
    probleminfo->Utemp[i * numentities + j] =
        probleminfo->Utemp[i * numentities + j] * sqrt(
        sigma[i]);
    fprintf (stderr, "USigma i %d j %d U %f\n", i, j,
        probleminfo->Utemp[i*numentities + j]);
}*/
fprintf(stderr, " maxdiff after normalizing V,%f\n",
        probleminfo->maxdifference);

/* calculating mxdifference after U*/
udiff = 0.0;
for(i = 0; i < numentities; ++i){
for (k = 0; k < q; ++k) {
    if (iter != iterations) {
        udiff = fabs(Utemp[k*numentities+i] - probleminfo
                      ->U[k*numentities+i]);
        if (udiff > probleminfo->maxdifference) {
            probleminfo->maxdifference = udiff;
        }
    }
    probleminfo->U[k*numentities+i] = Utemp[k*
                                           numentities+i];
    /*fprintf(stderr, "i%d k%d U%f\n", i, k,
              probleminfo->U[k*numentities + i]);*/
}
printf(stderr, "maxdiff after normalizing U%f\n", probleminfo->maxdifference);
/*fprintf(stderr, "normalized\n");*/
for (i=0; i < numentities; ++i){
    for (j=0; j < numattributes; ++j){
        temp2 = 0.0;
        for (k=0;k < q; ++k){
            temp2 += (probleminfo->V[numattributes*k + j]*
                      probleminfo->U[k*numentities+i]);
        }
        objval += fabs(entitiesinfo->points[i*numattributes+j
                                 ]-temp2);
    }
}
printf(stdout, "final obj%f\n", objval);

return 0;
}

int dgesvd1 (char jobu, char jobvt, int m, int n, double *
               A, int lda, double *S, double *U, int ldu, double *
               VT , int ldvt, double *work, int lwork) {
    /* SVD */
    extern void dgesvd((const char *jobup, const char *
                         jobvtp, const int *mp, const int *np, double *A, int
                         *ldap, double *S, double *U, const int *ldup,
                         double *VT, int *ldvt, double *work, int *lworkp,
                         int *infop);

    int info;
    dgesvd_(&jobu, &jobvt, &m, &n, A, &lda, S, U, &ldu, VT
               , &ldvt, work, &lwork, &info);

83
```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "type.h"

int allocateMemory2 (ENTITYINFOptr entityinfo, 
                     SOLVERINFOptr solverinfo, PROBLEMINFOptr probleminfo);

int solveL1Pca (ENTITYINFOptr entityinfo, SOLVERINFOptr 
                solverinfo, PROBLEMINFOptr probleminfo);

int solvePcaL1 (IOINFOptr ioinfo, ENTITYINFOptr 
                entityinfo, PROBLEMINFOptr probleminfo); /*

static void free_and_null (char **ptr);

void l1pca (double *points, int *dataDim, int *q, double *tolerance, 
            int *iterations, double *initV, int *getScores, double *PCs, double *Scores)
{

    ENTITYINFO entityinfo;
    SOLVERINFO solverinfo;
    PROBLEMINFO probleminfo;

    int status = probleminfo.status;
    int i = probleminfo.i;
    status = 0;

    entityinfo.points = points;
    entityinfo.numentities = dataDim[1];
    entityinfo.numattributes= dataDim[0];

    solverinfo.modelU = NULL;
    solverinfo.modelV = NULL;

    probleminfo.q = *q;
    probleminfo.initV = initV;
fprintf(stdout, "initV %p", probleminfo.initV);
    printf("%p\n", initV, probleminfo.initV);
    probleminfo.getScores = *getScores;
```
probleminfo.tolerance = *tolerance;
probleminfo.iterations = *iterations;
probleminfo.rhsL = NULL;
probleminfo.rhsU = NULL;
probleminfo.matbeg = NULL;
probleminfo.matval = NULL;
probleminfo.matind = NULL;
probleminfo.obj = NULL;
probleminfo.ub = NULL;
probleminfo.lb = NULL;
probleminfo.colname = NULL;
probleminfo.nu = NULL;
probleminfo.nv = NULL;
/*probleminfo.sigma = NULL;*/
probleminfo.Vtemp = NULL;
probleminfo.Utemp = NULL;
probleminfo.rowind = NULL;
probleminfo.xinda = NULL;
probleminfo.xindb = NULL;
probleminfo.vind = NULL;
/*probleminfo.Vsol = NULL;
probleminfo.Usol = NULL;*/
probleminfo.V = PCs;
/* printf(stdout, "V %p PCs %p \n", probleminfo.V, PCs) */
probleminfo.U = Scores;
/* printf(stdout, "U %p Scores %p \n", probleminfo.U, Scores); */
status = allocateMemory2 (&entityinfo, &solverinfo, &probleminfo); /* at the end of this file */
if (status) {
    fprintf(stderr, "Unable to allocate memory\n");
goto TERMINATE;
}

status = solveL1Pca (&entityinfo, &solverinfo, &probleminfo); /* in solveproblem.c*/
if (status) {
    fprintf(stderr, "Unable to solve. Terminating... or done\n");
goto TERMINATE;
}

TERMINATE:
free_and_null ((char **) &probleminfo.rhsL);
free_and_null ((char **) &probleminfo.rhsU);
free_and_null ((char **) &probleminfo.matbeg);
free_and_null ((char **) &probleminfo.matval);
free_and_null ((char **) &probleminfo.matind);
free_and_null ((char **) &probleminfo.obj);
free_and_null ((char **) &probleminfo.lb);

printf(stdout, "colname\n");
free_and_null ((char **) &probleminfo_ub);

for (i = 0; i < 2*entityinfo.numentities*entityinfo.numattributes + entityinfo.numattributes*probleminfo.q; ++i) {
    free_and_null ((char **) &probleminfo.colname[i]);
}
free_and_null ((char **) &probleminfo.colname);
free_and_null ((char **) &probleminfo.nu);
free_and_null ((char **) &probleminfo.nv);
free_and_null (/*free_and_null ((char **) &probleminfo.sigma);*/);
free_and_null ((char **) &probleminfo.Vtemp);
free_and_null ((char **) &probleminfo.uind);
free_and_null (/*free_and_null ((char **) &probleminfo.Vsol);*/);
free_and_null ((char **) &probleminfo.Utemp);
for (i = 0; i < entityinfo.numentities; ++i) {
    free_and_null ((char **) &probleminfo.rowind[i]);
}
free_and_null ((char **) &probleminfo.rowind);
free_and_null ((char **) &probleminfo.xinda);
free_and_null ((char **) &probleminfo.xindb);
for (i = 0; i < entityinfo.numattributes; ++i) {
    free_and_null ((char **) &probleminfo.vind[i]);
}
free_and_null ((char **) &probleminfo.vind);
if (solverinfo.modelU != NULL) {
    Clp_deleteModel (solverinfo.modelU);
    if (status) {
        fprintf(stderr, "Clp_delete failed, error code%d.
", status);
    }
}
if (solverinfo.modelV != NULL) {
    Clp_deleteModel (solverinfo.modelV);
    if (status) {
        fprintf(stderr, "Clp_delete failed, error code%d.
", status);
    }
}
```c
static void free_and_null (char **ptr)
{
    if (*ptr != NULL) {
        free (*ptr);
        *ptr = NULL;
    }
} /* END free_and_null */

int allocateMemory2 (ENTITYINFOptr entityinfo,
    SOLVERINFOptr solverinfo, PROBLEMINFOptr probleminfo)
{
    int numentities = entityinfo->numentities;
    int numattributes = entityinfo->numattributes;
    int numcols = probleminfo->numcols;
    int q = probleminfo->q;  
    int rcnt = probleminfo->rcnt;  
    int nzcnt = probleminfo->nzcnt;

    int i = probleminfo->i;  
    int j = probleminfo->j;

    /* allocate memory for columns */
    numcols = 2 * numentities * numattributes + numattributes * q;
    /* number of columns when solving for V */
    probleminfo->obj = (double *) malloc (numcols * sizeof (double));
    probleminfo->lb = (double *) malloc (numcols * sizeof (double));
    probleminfo->ub = (double *) malloc (numcols * sizeof (double));
    probleminfo->colname = (char **) malloc (numcols * sizeof (char *));
    for (i = 0; i < numcols; ++i) {
        probleminfo->colname[i] = (char *) malloc (20 * sizeof (char));
    }

    /* allocate memory for constraints */

87
rcnt = numentities*numattributes; /* number of rows when solving for V */

nzcnt = 2*numentities*numattributes + numentities*numattributes*q; /* number of non-zeros when solving for V */

probleminfo->rhsL = (double *) malloc (rcnt * sizeof (double));
probleminfo->rhsU = (double *) malloc (rcnt * sizeof (double));
probleminfo->xinda = (int *) malloc (rcnt * sizeof (int));
probleminfo->xindb = (int *) malloc (rcnt * sizeof (int));
probleminfo->matbeg = (int *) malloc ((numcols+1) * sizeof (int));
probleminfo->matind = (int *) malloc (nzcnt * sizeof (int));
probleminfo->matval = (double *) malloc (nzcnt * sizeof (double));

probleminfo->uind = (int *) malloc(q*sizeof(int));

probleminfo->Vtemp = (double *) malloc (numattributes*q * sizeof (double));
probleminfo->Utemp = (double *) malloc (numentities*q* sizeof (double));
probleminfo->nu = (double *) malloc (q*sizeof(double));
probleminfo->nv = (double *) malloc (q*sizeof(double));

/* probleninfo->sigma = (double *) malloc (q*sizeof(double)); */

probleminfo->rowind = (int **) malloc(numentities * sizeof(int *));
for (i = 0; i < numentities; ++i) {
    probleminfo->rowind[i] = (int *) malloc(numattributes * sizeof(int));
}

probleminfo->vind = (int **) malloc(numattributes * sizeof(int *));
for (j = 0; j < numattributes; ++j) {
    probleminfo->vind[j] = (int *) malloc(q * sizeof(int)) ;
}
return 0;
} /* end allocateMemory */
R Code

PCA-L1

R/InterfacePcaL1.R

pcal1 <- function (X, projDim = 1, center=FALSE, score = FALSE, dispersionExp = FALSE)
{
  if(class(X) != "matrix")
  {
    if(class(X) == "data.frame")
      X <- as.matrix(X)
    else
      X <- matrix(X, ncol = 1)
  }

  if(center){
    mymedian <- apply(X, 2, median)
    myMedMat <- matrix(rep(mymedian, nrow(X)), ncol = ncol(X), byrow=TRUE)
    X <- X−myMedMat
  }

  A <- t(X)
  X <- X[apply(abs(X),1,sum) > 0,] # get rid of origin points for algorithm
  X <- t(X)
  pcLen <- projDim * (nrow(X))
  sol <- .C("PcaL1_R", as.double(X), as.integer(dim(X)),
            as.integer(projDim), PCs = double(pcLen), PACKAGE="PCA3")
  PCS <- sol[["PCs"]]
  dim(PCS) <- c(nrow(X), projDim)
  final <- list(PCs =PCS)
  SCORE = t(A) %>% PCS
  if(score) {
    final["Score"] <- list(Score = SCORE)
  }
if(dispersionExp){
  PropVar = double(projDim)
  TotalVar <- sum(apply(t(A), 2, abs))
  Scorevar <- (apply(abs(SCORE), 2, sum))
  PropVar <- Scorevar/TotalVar
  final["PropL1Dispersion"] <- list(PropL1Dispersion = PropVar)
}
final

L1-PCA*

R/Inter_L1PcaStar.R

l1pcaStar <- function (X, projDim = 1, center=FALSE, getScore = FALSE, getProjPoints =FALSE, dispersionExp = FALSE)
{
  if(class(X) != "matrix")
  {
    if (class(X) == "data.frame")
      X <- as.matrix(X)
    else
      X <- matrix(X, ncol = 1)
  }
  if(center){
    mymedian <- apply(X, 2, median)
    myMedMat <- matrix(rep(mymedian, nrow(X)), ncol = ncol(X), byrow=TRUE)
    X <- X-mymedMat
  }
  X<-t(X)
  pcLen <- nrow(X)*nrow(X)
  scoreLength <- projDim * ncol(X)
  projLength <- nrow(X) * ncol(X)
  res <- .C("l1pcaStar", as.double(X), as.integer(dim(X)), as.integer(projDim), as.integer(getScore), as.integer(getProjPoints), pcs = double(pcLen), Scores = double(scoreLength), ProjPoints = double(projLength))
  final <- new.env()
  #$final$PCs <- matrix(res[["pcs"]], ncol=projDim, byrow=TRUE)
  tempPC <- matrix(res[["pcs"]], ncol=nrow(X), byrow=TRUE)
\)

\text{tempPC} \leftarrow \text{tempPC}[,1:\text{projDim}]

\text{final}\$\text{PCs} \leftarrow \text{tempPC}

#dim(\text{final}\$\text{PCs}) \leftarrow \text{c}(\text{dim}(X)[1], \text{dim}(X)[1])

\text{MatScores} \leftarrow \text{matrix}(\text{res}["\text{Scores"}], \text{ncol}=\text{projDim}, \text{byrow} =\text{TRUE})

\text{if(\text{getScore})}

\{ \text{final}\$\text{Scores} \leftarrow \text{MatScores} \}

\text{if(\text{getProjPoints})}

\{ \text{final}\$\text{ProjPoints} \leftarrow \text{res}["\text{ProjPoints"}]

\text{dim(\text{final}\$\text{ProjPoints})} \leftarrow \text{dim}(t(X)) \}

\text{if(\text{dispersionExp})}

\{ \text{PropDisp} = \text{double}(\text{projDim})

\text{TVar} \leftarrow \text{sum}(\text{apply}(t(X), 2, \text{abs}))

\text{print(TVar)}

\text{ScoreVar} \leftarrow (\text{apply}(\text{abs}(\text{MatScores}), 2, \text{sum}))

\text{print(ScoreVar)}

\text{PropDisp} \leftarrow \text{ScoreVar}/\text{TVar}

\text{final}\$\text{PropL1Dispersion} \leftarrow \text{PropDisp} \}

\text{as.\text{list}(final)}

\}\text{L1-PCA}

R/InterL1Pca.R

\text{l1pca} \leftarrow \text{function}(X, \text{projdim} =1, \text{center}=\text{TRUE}, \text{tolerance} = 0.0001, \text{iterations} = 10, \text{getScore} =\text{TRUE},

\text{getProjPoints} = \text{TRUE}, \text{dispersionExp} = \text{TRUE})

\{ \text{if(\text{class}(X) != "matrix")}

\{ \text{if (\text{class}(X) == "data.frame")}

\text{X} \leftarrow \text{as.\text{matrix}(X)}

\text{else}

\text{X} \leftarrow \text{matrix}(X, \text{ncol} = 1) \}

\text{if(\text{center})}
mymedian <- apply(X, 2, median)
myMedMat <- matrix(rep(mymedian, nrow(X)), ncol = ncol(X), byrow=TRUE)
X <- X-myMedMat
}

mypca <- prcomp(X, retx=TRUE)
InitV <- mypca$rotation[,1:projdim]
# print(InitV)
X <- t(X)
pcLength <- projdim * nrow(X)
scoreLength <- projdim * ncol(X)
dataLength <- nrow(X) * ncol(X)
sol <- .C("l1pca", as.double(X), as.integer(dim(X)),
as.integer(projdim), as.double(tolerance), as.integer(iterations), as.double(InitV), as.integer(getScore), pcs = double(pcLength), Scores = double(scoreLength), PACKAGE="pcaL1")
solution <- new.env()
solution$PC <- matrix(sol[['pcs']], ncol=projdim, byrow =TRUE)
#dim(solution$PC) <- c(dim(X)[1], projdim)
ScoreMatrix <- matrix(sol[['Scores']], ncol=projdim, byrow =FALSE)
#dim(solution$Scores) = c(dim(X)[2], projdim)

if(getScore){
solution$Scores <- ScoreMatrix
}

if(getProjPoints){
  ProjectedPoints <- t((solution$PC) %*% t(solution$PC) %*% X)
solution$ProjectedPoints <- ProjectedPoints
}

if(dispersionExp){
  PropDisp <- double(projdim)
  TotalVariance <- sum(apply(t(X), 2, abs))
  # print(TotalVariance)
  ScoreVariance <- (apply(abs(ScoreMatrix), 2, sum))
  # print(ScoreVariance)
  PropDisp <- ScoreVariance/TotalVariance
  solution$PropL1Dispersion <- PropDisp
}
so <- sort(solution$PropL1Dispersion, decreasing=TRUE,
$\text{index.return = TRUE)}$

solution$\text{Scores} \leftarrow \text{solution}\text{Scores [ }, \text{so}\text{ix]}$

solution$\text{PC} \leftarrow \text{solution}\text{PC [ }, \text{so}\text{ix]}$

solution$\text{PropL1Dispersion} \leftarrow \text{sort (solution}\text{PropL1Dispersion, decreasing = TRUE)}$

as.list(solution)

}
Documentation: man folder

pca1

pca1.Rd

\name{PcaL1Kwak}
\alias{PcaL1Kwak}
\title{Implementation of Algorithm PCA-L1}
\description{
Performs the principal component analysis using
algorithm PCA-L1, proposed by Kwak.
}
\usage{
PcaL1Kwak(X, projDim=1, center=FALSE, score=TRUE,
dispersionExp=FALSE)
}
\arguments{
\item{X}{\data, must be in \code{matrix} or \table form}
\item{projDim}{number of projections to project \data into, must be an \code{integer}, \code{default} is 1.}
\item{center}{whether to center the \data using Median, \code{default} is \code{FALSE}}
\item{score}{whether to \code{return} a \code{Scores} \code{matrix}, \code{default} is \code{FALSE}}
\item{dispersionExp}{whether to \code{return} L1 dispersion explained for each component, \code{default} is \code{FALSE}}
}
\examples{
## for data matrix `mat", projects data into 1 dimension.
myPca1 <- PcaL1Kwak(mat)

## projects centered data into 2 dimensions.
myPca1 <- PcaL1Kwak(mat, projDim=2, center=TRUE)
}
\keyword{PCA}
l1pcastar

\name{L1PcaStar}
\alias{L1PcaStar}
\title{Implementation of L1−PCA\cite{Brooks2010}}
\description{Performs the principal component analysis
using algorithm L1−PCA\cite{Brooks2010} proposed by
Brooks et. al.}
\usage{
L1PcaStar(X, projDim=1, center=FALSE, getScore=FALSE,
getProjPoints=FALSE, dispersionExp=FALSE)
}
\arguments{
\item{X}{data, must be in \code{matrix} or \code{table} form}
\item{projDim}{number of projections to project data
into, must be an \code{integer}, default is 1.}
\item{center}{whether to center the data using Median,
default is \code{FALSE}}
\item{getScore}{whether to \code{return} a Scores matrix,
default is \code{FALSE}}
\item{getProjPoints}{whether to calculate the projected
points (reconstruction), default is \code{FALSE}}
\item{dispersionExp}{whether to \code{return} L1 dispersion
explained for each component, default is \code{FALSE}}
}
\examples{
## for data matrix ‘‘mat”, projects data into 1 dimension.
myl1pcastar <- L1PcaStar(mat)

## projects centered data into 2 dimensions.
myl1pcastar <- L1PcaStar(mat, projDim=2, center=TRUE)
}
\references{
Brooks J.P, Dula J., and Boone E (2010) \emph{A Pure L1
−norm Principal Component Analysis}
}

l1pca

\name{L1Pca}
\alias{L1Pca}
\title{Implementation of L1−PCA}
\textbf{description}{Perform the principal component analysis using algorithm L1-PCA proposed by Ke and Kanade}

\textbf{usage}{
L1Pca(X, projdim=1, center=FALSE, tolerance=0.0001, iterations=10, getScore=FALSE, getProjPoints=FALSE, dispersionExp=FALSE)
}

\textbf{arguments}{
\item \texttt{\{X\}}{data, must be in \texttt{matrix} or \texttt{table} form}
\item \texttt{\{projDim\}}{number of projections to project data into, must be an integer, default is 1.}
\item \texttt{\{center\}}{whether to center the data using Median, default is FALSE}
\item \texttt{\{tolerance\}}{sets the convergence tolerance for the algorithm, default is 0.0001}
\item \texttt{\{iterations\}}{sets the number of iterations to run before returning the result, default is 10}
\item \texttt{\{getScore\}}{whether to return a Scores matrix, default is FALSE}
\item \texttt{\{getProjPoints\}}{whether to calculate the projected points (reconstruction), default is FALSE}
\item \texttt{\{dispersionExp\}}{whether to return L1 dispersion explained for each component, default is FALSE}
}

\textbf{examples}{
## for data matrix "mat", projects data into 1 dimension.
myL1pca <- L1Pca(mat)

## projects centered data into 2 dimensions.
myL1pca <- L1Pca(mat, projDim=2, center=TRUE, tolerance =0.00001, iterations=20)
}

\textbf{references}{
Kanade T. and Ke Q. (2005) \textbf{Robust L1 Norm Factorization in the Presence of Outliers and Missing Data by Alternative Convex Programming}
}