

# sparseWeightBasedPCA Package

Niek C. de Schipper n.c.deschipper@uvt.nl

2020-06-23

## Contents

<b>sparseWeightBasedPCA: A package for Regularized weight based Simultaneous Component Analysis (SCA) and Principal Component Analysis (PCA)</b>	<b>1</b>
Theoretical background . . . . .	1
Models of the sparseWeightBasedPCA package . . . . .	2
The implementation in R of the sparseWeightBasedPCA package . . . . .	3
Detailed examples of SCA and PCA with the sparseWeightBasedPCA package . . . . .	7

`library(sparseWeightBasedPCA)`

## sparseWeightBasedPCA: A package for Regularized weight based Simultaneous Component Analysis (SCA) and Principal Component Analysis (PCA)

### Theoretical background

#### Principal Component Analysis

Principal component analysis (PCA) is a widely used analysis technique for data reduction. It can give crucial insights in the underlying structure of the data when used as a latent variable model.

Given a data matrix  $\mathbf{X}$  that contains the scores for  $i = 1 \dots I$  observations on  $j = 1 \dots J$  variables; we follow the convention to present the  $J$  variable scores of observation  $i$  in row  $i$  and thus  $\mathbf{X}$  has size  $I \times J$ . PCA decomposes the data into  $Q$  components as follows,

$$\begin{aligned} \mathbf{X} &= \mathbf{XW}\mathbf{P}^T + \mathbf{E} \\ \text{subject to } \mathbf{P}^T\mathbf{P} &= \mathbf{I}, \end{aligned} \tag{1}$$

where  $\mathbf{W}$  is a  $J \times Q$  component weight matrix,  $\mathbf{P}$  is a  $J \times Q$  loading matrix and  $\mathbf{E}$  is a  $I \times J$  residual matrix. The component weight matrix  $\mathbf{W}$  will be the focus of this package, note that  $\mathbf{T} = \mathbf{XW}$  represent the component scores.

The advantage of inspecting the component weights instead of the loadings is that you can directly derive meaning to  $\mathbf{T}$ , this because you see precisely in what way items in  $\mathbf{X}$  are weighted together by  $\mathbf{W}$ .

#### Simultaneous Component Analysis

The decomposition in (1) can be extended to the case of multi-block data by taking  $\mathbf{X}_c = [\mathbf{X}_1 \dots \mathbf{X}_K]$ ; this is concatenating the  $K$  data blocks composed of different sets of variables of size  $J_k$  for the same units of observation. The decomposition of  $\mathbf{X}_c$  has the same block structured decomposition as in (1) with  $\mathbf{W}_c = [\mathbf{W}_1^T \dots \mathbf{W}_K^T]^T$  and  $\mathbf{P}_c = [\mathbf{P}_1^T \dots \mathbf{P}_K^T]^T$ . This multi-block formulation of PCA is known as simultaneous component analysis (SCA):

$$\begin{aligned} [\mathbf{X}_1 \dots \mathbf{X}_K] &= [\mathbf{X}_1 \dots \mathbf{X}_K][\mathbf{W}_1^T \dots \mathbf{W}_K^T]^T[\mathbf{P}_1^T \dots \mathbf{P}_K^T] + \mathbf{E} \\ \text{subject to } [\mathbf{P}_1^T \dots \mathbf{P}_K^T][\mathbf{P}_1^T \dots \mathbf{P}_K^T]^T &= \mathbf{I} \end{aligned} \tag{2}$$

When analyzing multi-block data with SCA identifying meaningful relations between data blocks is of prime interest. In order to gain insight in what multiple data blocks relate to each other, we can search for blockwise structures in the component weights that tell us whether a component is uniquely determined by variables from one single data block (distinctive component), or whether it is a component that is determined by variables from multiple data blocks (common component). In other words, a distinctive component is a linear combination of variables of a particular data block only, whereas a common component is a linear combination of variables of multiple data blocks. An example of common and distinctive components in the situation with two

data blocks is given below. The first two components are distinctive components, the third component is a common component,

$$\mathbf{T} = [\mathbf{X}_1 \quad \mathbf{X}_2] \begin{bmatrix} \mathbf{W}_1 \\ \mathbf{W}_2 \end{bmatrix} = [\mathbf{X}_1 \quad \mathbf{X}_2] \begin{bmatrix} 0 & w_{1,2} & w_{1,3} \\ 0 & w_{2,2} & w_{2,3} \\ 0 & w_{3,2} & w_{3,3} \\ w_{1,2,1} & 0 & w_{1,2,3} \\ w_{2,2,1} & 0 & w_{2,2,3} \\ w_{3,2,1} & 0 & w_{3,2,3} \end{bmatrix}.$$

The `sparseWeightBasedPCA` package will provide functions that perform PCA and SCA on the component weights. It will also provide function for selection of the hyper parameters of the model. We will now describe the core models of this package that will be estimated by the following functions.

1. `scads` Regularized SCA with sparse component weights using constraints
2. `mmsca` Regularized SCA with sparse component weights using the group LASSO
3. `ccpca` PCA with sparse component weights using cardinality constraints

WRITE HERE WHAT OTHER PACKAGES DO

## Models of the `sparseWeightBasedPCA` package

### Regularized SCA with sparse component weights using constraints

Here we present an approach of performing regularized SCA, with ridge and LASSO regularization and block wise constraints on  $\mathbf{W}_c$  by solving,

$$\begin{aligned} L(\mathbf{W}_c, \mathbf{P}_c) = & \|\mathbf{X}_c - \mathbf{X}_c \mathbf{W}_c \mathbf{P}_c^T\|_2^2 + \lambda_L \|\mathbf{W}_c\|_1 + \lambda_R \|\mathbf{W}_c\|_2^2 \\ & \text{subject to } \mathbf{P}_c \mathbf{P}_c^T = \mathbf{I}, \text{ and } \lambda_L, \lambda_R \geq 0 \text{ and zero block constraints on } \mathbf{W}_c \end{aligned} \quad (3)$$

In order to get a minimum for (3) we alternate between the estimation of  $\mathbf{W}_c$  and  $\mathbf{P}_c$ . Given  $\mathbf{W}_c$  we can estimate  $\mathbf{P}_c$  by using procrustes rotation. Given  $\mathbf{P}_c$  we find estimates for  $\mathbf{W}_c$  by using a coordinate descent algorithm that works by soft-thresholding the weights. For the specifics we refer the reader to (REF: NIEK KATRIJN). This iterative procedure stops when an optimum has been found (i.e the loss function value is not decreasing anymore beyond pre-specified tolerance level). The optimization problem in (4) is non-convex and meaning there are local minima. In order to deal with that multiple random starts can be used with different initializations of  $\mathbf{W}$ , the start leading to the lowest evaluation of (4) is retained. Typically starting the algorithm with the solution of PCA (e.g. the first  $Q$  right singular vectors of  $\mathbf{X}$ ) will lead to smallest optimum.

The main advantage of analyzing multi-block data by using this procedure is that it is fast, and scalable to large data sets thanks to the coordinate descent implementation. The inclusion of the blockwise constraints on  $\mathbf{W}$  make sure common and distinctive components are found and the LASSO and ridge regularizers are optional and facilitate extra sparsity within the component weights. A disadvantage of the method is that the common and distinctive structures for  $\mathbf{W}$  need to be selected. This has to be done using model selection procedures and can be computationally demanding.

This procedure has been implemented in the `scads` function. This function will be discussed in detail in the next section and examples will be given outlining the analysis including model selection.

### Regularized SCA with sparse component weights using the group LASSO

Here we present a very flexible approach of performing regularized SCA using, ridge, LASSO, group LASSO and elitist LASSO regularization by solving:

$$\begin{aligned} L(\mathbf{W}_c, \mathbf{P}_c) = & \|\mathbf{X}_c - \mathbf{X}_c \mathbf{W}_c \mathbf{P}_c^T\|_2^2 + \lambda_L \|\mathbf{W}_c\|_1 + \lambda_R \|\mathbf{W}_c\|_2^2 \\ & + \sum_{q,k} (\lambda_G \sqrt{J_k} \|\mathbf{w}_q^{(k)}\|_2 + \lambda_E \|\mathbf{w}_q^{(k)}\|_{1,2}) \\ & \text{subject to } \mathbf{P}_c \mathbf{P}_c^T = \mathbf{I} \text{ and } \lambda_L, \lambda_R, \lambda_G, \lambda_E \geq 0 \end{aligned} \quad (4)$$

where  $\mathbf{W}_c = [(\mathbf{W}^{(1)})^T \dots (\mathbf{W}^{(K)})^T]^T$ , and  $\mathbf{w}_q^{(k)}$  denotes the  $q$ th column from the submatrix  $\mathbf{W}^{(k)}$ . In order to get a minimum for (4) we alternate between the estimation of  $\mathbf{W}_c$  and  $\mathbf{P}_c$ . Given  $\mathbf{W}_c$  we can estimate  $\mathbf{P}_c$  by using procrustes rotation. Given  $\mathbf{P}_c$  we can find estimates for  $\mathbf{W}_c$  by using the majorization minimization (MM) algorithm. For the specifics we refer the reader to (REF: NIEK KATRIJN). This iterative procedure stops when an optimum has been found (i.e the loss function value is not decreasing anymore beyond pre-specified tolerance level). The optimization problem in (4) is non-convex and meaning there are local minima. In order to deal with that multiple random starts can be used with different initializations of  $\mathbf{W}$ , the start leading to the lowest evaluation of (4) is retained. Typically starting the algorithm with the solution of PCA (e.g. the first  $Q$  right singular vectors of  $\mathbf{X}$ ) will lead to smallest optimum.

The main advantage of solving (4) is that it can automatically look for common and distinctive components by taking advantage of the properties of the group lasso. Because the group LASSO is specified on the colored segments (see below), it will either include these segments or put them zero, uncovering common and distinctive components. This is especially useful if the number of blocks and components is large, and an exhaustive approach of identifying common and distinctive is too computationally intensive.

$$\mathbf{T} = [\mathbf{X}_1 \quad \mathbf{X}_2] \begin{bmatrix} \mathbf{W}_1 \\ \mathbf{W}_2 \end{bmatrix} = [\mathbf{X}_1 \quad \mathbf{X}_2] \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \\ w_{1,2,1} & w_{1,2,2} & w_{1,2,3} \\ w_{2,2,1} & w_{2,2,2} & w_{2,2,3} \\ w_{3,2,1} & w_{3,2,2} & w_{3,2,3} \end{bmatrix}.$$

The inclusion the LASSO and ridge regularization are optional and facilitate extra sparsity within the colored segments. The elitist LASSO has a very special use case, the elitist LASSO will include all colored segments and will put weights within each segment to zero. The elitist lasso can be used to force components to be common. It is not advised to use the group LASSO and the elitist LASSO together as they have opposing goals. A disadvantage of using this procedure is that is potentially slow, this because its implemented using a MM-algorithm which tend to be slow in convergence.

This procedure has been implemented in the `mmsca` function. This function will be discussed in detail in the next section and examples will be given outlining the analysis including model selection.

### PCA with sparse component weights using cardinality constraints

Here we present an approach of solving PCA by applying cardinality constraints to the component weights by solving:

$$L(\mathbf{W}, \mathbf{P}) = \|\mathbf{X} - \mathbf{XWP}^T\|_2^2 \quad (5)$$

subject to  $\mathbf{W}$  including  $K$  zeros.

In order to get a minimum for (5) we need to alternate between the estimation of  $\mathbf{W}$  and  $\mathbf{P}$ . Given  $\mathbf{W}$  we can estimate  $\mathbf{P}$  by using procruste rotation (REF: Ten\_Berge\_2005, Zou\_2006),  $\mathbf{P} = \mathbf{UV}^T$ , where  $\mathbf{U}$  and  $\mathbf{V}$  are the left and right singular vectors of  $\mathbf{X}^T \mathbf{XW}$ . Given  $\mathbf{P}$  we can find estimates for  $\mathbf{W}$  given the cardinality constraints using the cardinality constraint regression algorithm for detail see (REF: NIEK KATRIJN). The optimization problem in (5) is non-convex and meaning there are local minima. In order to deal with that multiple random starts can be used with different initializations of  $\mathbf{W}$ , the start leading to the lowest evaluation of (5) is retained. Typically starting the algorithm with the solution of PCA (e.g. the first  $Q$  right singular vectors of  $\mathbf{X}$ ) will lead to smallest optimum.

The main advantage of solving (5) is that this model tries to directly tackle the problem of finding the underlying subset of weights, in contrast to the usage of a penalty that shrinks the weights and also induces sparsity such as the LASSO. This approach can lead to better discovery of the underlying weights compared to LASSO (REF: NIEK KATRIJN). Another advantage is that you directly impose cardinality constraints on  $\mathbf{W}$ . This gives the user total control over the amount of sparsity. This can be desirable if there is already an idea about the level of sparsity in the final model. A disadvantage of using this procedure is that is potentially slow, this because the CCREG algorithm uses a MM-algorithm which tend to be slow in convergence. Another potential downside could be the absence of regularizers, they tend to shrink the variance of the estimators leading to more efficiency. In noisy situations other procedures might outperform this procedure.

This model has been implemented in the `ccpca` function. This function will be discussed in detail in the next section.

### The implementation in R of the `sparseWeightBasedPCA` package

The `sparseWeightBasedPCA` provides functions for the aforementioned models, also model selection procedures are provided in order to tune the hyper parameters. The main functions: `scads`, `mmsca` and `ccpca` are implemented in C++ using the packages Rcpp (REF) and RcppArmadillo (REF). The model selection procedures are implemented in R. The functions of the package will be discussed after which detailed examples will be given.

#### Implementation of the main functions of the `sparseWeightBasedPCA` package

The main functions: `scads`, `mmsca` and `ccpca` are all implemented in a similar manor. At the core they perform SCA/PCA with variations depending on the goals of the user. An typical minimal use case `scads` given below (`mmsca` and `ccpca` are implemented very similarly) first generate some fake data:

```
J <- 30
I <- 100
X <- matrix(rnorm(I*J), I, J)
```

With this data `scads` can be run with supplied values for the minimal required arguments. This minimal example performs PCA, with no constraints and, ridge and LASSO regularization.

```
ncomp <- 3
constraints <- matrix(1, J, ncomp) # No constraints
res <- scads(X,
  ncomp = ncomp,
  ridge = 10e-8,
  lasso = rep(1, ncomp),
  constraints = constraints,
  Wstart = matrix(1, J, ncomp),
  itr = 10e5)
```

All the main functions: `scads`, `mmsca` and `ccpca` output a list with the following named elements:

- `W` A matrix containing the component weights
- `P` A matrix containing the loadings
- `loss` A numeric variable containing the minimum loss function value of all the `nStarts` starts
- `converged` A boolean containing `TRUE` if converged `FALSE` if not converged.

Details examples of data analysis will follow in the next section or see the documentation `?scads` et cetera. The main functions `scads`, `mmsca` and `ccpca` have a slightly different set of arguments. See the following list for the specifics.

### Overview of the `scads` arguments

- `X` A data matrix of class `matrix`
- `ncomp` The number of components to estimate (an integer)
- `ridge` A numeric value containing the ridge parameter for ridge regularization on the component weight matrix `W`
- `lasso` A vector containing a ridge parameter for each column of `W` separately, to set the same lasso penalty for the component weights `W`, specify: `lasso = rep(value, ncomp)`
- `constraints` A matrix of the same dimensions as the component weights matrix `W` (`ncol(X)` x `ncomp`). A zero entry corresponds in `constraints` corresponds to an element in the same location in `W` that needs to be constraint to zero. A non-zero entry corresponds to an element in the same location in `W` that needs to be estimated.
- `itr` The maximum number of iterations (an integer)
- `Wstart` A matrix of `ncomp` columns and `nrow(X)` rows with starting values for the component weight matrix `W`, if `Wstart` only contains zeros, a warm start is used: the first `ncomp` right singular vectors of `X`
- `tol` The convergence is determined by comparing the loss function value after each iteration, if the difference is smaller than `tol`, the analysis is converged. The default value is `10e-8`.
- `nStarts` The number of random starts the analysis should perform. The first start will be performed with the values given by `Wstart`. The consecutive starts will be `Wstart` plus a matrix with random uniform values times the current start number (the first start has index zero).
- `printLoss` A boolean: `TRUE` will print the loss function value each 10th iteration.

### Overview of the `mmsca` arguments

- `X` A data matrix of class `matrix`
- `ncomp` The number of components to estimate (an integer)
- `ridge` A vector containing a ridge parameter for each column of `W` separately, to set the same ridge penalty for the component weights `W`, specify: `ridge = rep(value, ncomp)`, value is a non-negative double
- `lasso` A vector containing a ridge parameter for each column of `W` separately, to set the same lasso penalty for the component weights `W`, specify: `lasso = rep(value, ncomp)`, value is a non-negative double
- `grouplasso` A vector containing a grouplasso parameter for each column of `W` separately, to set the same grouplasso penalty for the component weights `W`, specify: `grouplasso = rep(value, ncomp)`, value is a non-negative double
- `elitistlasso` A vector containing a elitistlasso parameter for each column of `W` separately, to set the same elitistlasso penalty for the component weights `W`, specify: `elitistlasso = rep(value, ncomp)`, value is a non-negative double
- `groups` A vector specifying which columns of `X` belong to what block. Example: `c(10, 100, 1000)`. The first 10 variables belong to the first block, the 100 variables after that belong to the second block etc.
- `constraints` A matrix of the same dimensions as the component weights matrix `W` (`ncol(X)` x `ncomp`). A zero entry corresponds in `constraints` corresponds to an element in the same location in `W` that needs to be constraint to zero. A non-zero entry corresponds to an element in the same location in `W` that needs to be estimated.
- `itr` The maximum number of iterations (a positive integer)
- `Wstart` A matrix of `ncomp` columns and `nrow(X)` rows with starting values for the component weight matrix `W`, if `Wstart` only contains zeros, a warm start is used: the first `ncomp` right singular vectors of `X`
- `tol` The convergence is determined by comparing the loss function value after each iteration, if the difference is smaller than `tol`, the analysis is converged. Default value is `10e-8`

- **nStarts** The number of random starts the analysis should perform. The first start will be performed with the values given by **Wstart**. The consecutive starts will be **Wstart** plus a matrix with random uniform values times the current start number (the first start has index zero).
- **printLoss** A boolean: **TRUE** will print the loss function value each 10th iteration.
- **coordDes** A boolean with the default **FALSE**. If **coordDes** is **FALSE** the estimation of the majorizing function to estimate the component weights **W** conditional on the loadings **P** will be found using matrix inverses which can be slow. If set to true the majorizing function will be optimized (or partially optimized) using coordinate descent, in many cases coordinate descent will be faster
- **coordDesItr** An integer specifying the maximum number of iterations for the coordinate descent algorithm, the default is set to 1. You do not have to run this algorithm until convergence before alternating back to the estimation of the loadings. The tolerance for this algorithm is hard coded and set to  $10^{-8}$ .

## Overview of the **ccpca** arguments

- **X** A data matrix of class **matrix**
- **ncomp** The number of components to estimate (an integer)
- **nzeros** A vector of length **ncomp** containing the number of desired zeros in the columns of the component weight matrix **W**
- **itr** The maximum number of iterations (an integer)
- **Wstart** A matrix of **ncomp** columns and **nrow(X)** rows with starting values for the component weight matrix **W**, if **Wstart** only contains zeros, a warm start is used: the first **ncomp** right singular vectors of **X**
- **nStarts** The number of random starts the analysis should perform. The first start will be performed with the values given by **Wstart**. The consecutive starts will be **Wstart** plus a matrix with random uniform values times the current start number (the first start has index zero). The default value is 1.
- **tol** The convergence is determined by comparing the loss function value after each iteration, if the difference is smaller than **tol** the analysis is converged. The default value is  $10e-8$
- **printLoss** A boolean: **TRUE** will print the loss function value each 10th iteration.

## Implementation of the model selection functions of the **sparseWeightBasedPCA** package

The to execute the **scads**, **mmsca** and **ccpca** arguments for the hyper parameters need to selected, for example the number of components or values for the **lasso** arguments. The **sparseWeightBasedPCA** package provides three procedures of selecting the hyper-parameters of the model: Cross-validation using the EigenVector method (REF), the Index of Sparseness (IS) and the Bayesian Information Criterion (BIC), for details see (REF). The procedures are implemented with in the functions: **CVforPCAwithSparseWeights**, **ISforPCAwithSparseWeights** and **BICforPCAwithSparseWeights**. They work as follows: first specify the arguments that the model selection function needs, then a pointer to the function that does the analysis should be given in the followed by its arguments. An example is given here:

```
J <- 30
I <- 100
X <- matrix(rnorm(I*J), I, J)
```

With this data, **CVforPCAwithSparseWeights** using **scads** works as follows,

```
ncomp <- 3
res <- CVforPCAwithSparseWeights(X = X,
                                nrFolds = 10,
                                FUN = scads, # Pointer to scads(), followed by the arguments
                                ncomp = ncomp,
                                ridge = 0,
                                lasso = rep(0.1, ncomp),
                                constraints = matrix(1, J, ncomp),
                                Wstart = matrix(0, J, ncomp),
                                itr = 10e5,
                                printLoss = FALSE)
```

This function returns a list with the following elements:

- **MSPE** The mean squared prediction error given the tuning parameters
- **MSPEstdError** The standard error of the MSPE
- **nNonZeroCoef** The number of non-zero coefficients in the model

Model selection of the hyper-parameters can be based on the model that results in the lowest mean squared prediction error, or a model can be picked with the least number of non-zero coefficients still within one standard error of the model with the lowest mean squared prediction error. The other functions work similar check see **?BICforPCAwithSparseWeights** and **?ISforPCAwithSparseWeights** for more detailed information.

## Additional tuning functions for `mmsca`

This package provides more elaborate model selection functions for `mmsca`, because of its flexibility it can be overwhelming to use the basic tuning function provided. Two additional functions are provided `mmscaModelSelection` and `mmscaHyperCubeSelection`. `mmscaModelSelection` uses a fixed grid of all combinations of the hyper-parameters to pick the best combination from, whereas `mmscaHyperCubeSelection` uses an adaptive grid that zooms in on a good combination of hyper-parameters until it converges on a certain combination. Note that `mmscaHyperCubeSelection` is experimental, it could potentially speed up the process of tuning enormously but it has not been scrutinized using a simulation study. A basic example of both is given here:

To perform model selection with `mmsca` using an exhaustive grid of all combinations of the tuning parameters the following can be done:

```
J <- 30
I <- 100
X <- matrix(rnorm(I*J), I, J)

out <- mmscaModelSelection(X,
  ridgeSeq = seq(0, 1, by = 0.1), # Sequences of the hyper parameter
  lassoSeq = 0:100,
  grouplassoSeq = 0, # No group lasso and no elitist lasso
  elitistlassoSeq = 0,
  ncompSeq = 1:3,
  tuningMethod = "CV", # Indicate the tuning method
  nrFolds = 10,
  groups = ncol(X), # Arguments for mmsca()
  itr = 100000,
  nStart = 1,
  coordDes = FALSE,
  coordDesItr = 100,
  printProgress = TRUE)
```

This function returns a list with two elements:

- **results** A list with `ncomp` elements each containing the following elements
  - "BIC, IS or MSPE" The index chosen in `tuningMethod` for all combinations of ridge, lasso, grouplasso and elitistlasso
  - "bestBIC, bestIS, bestMSPE or bestMSPE1stdErrorRule" The best index according to the chosen tuning method
  - "nNonZeroCoef" The number of non zero weights in the best model
  - "ridge" The value of the ridge penalty corresponding to the best model
  - "lasso" The value of the lasso penalty corresponding to the best model
  - "grouplasso" The value of the group lasso penalty corresponding to the best model
  - "elististlasso" The value of the elitist lasso penalty corresponding to the best model
  - "ncomp" The number of component that was used for these items
  - "ridge1stdErrorRule" In case `tuningMethod == "CV"`, the value of the ridge penalty according to the 1 standard error rule: the most sparse model within one standard error of the model with the lowest MSPE
  - "lasso1stdErrorRule" In case `tuningMethod == "CV"`, the value of the lasso penalty according to the 1 standard error rule: the most sparse model within one standard error of the model with the lowest MSPE
  - "grouplasso1stdErrorRule" In case `tuningMethod == "CV"`, the value of the group lasso penalty according to the 1 standard error rule: the most sparse model within one standard error of the model with the lowest MSPE
  - "elististlasso1stdErrorRule" In case `tuningMethod == "CV"`, the value of the elitist lasso penalty according to the 1 standard error rule: the most sparse model within one standard error of the model with the lowest MSPE
  - "ridge1stdErrorRule" In case `tuningMethod == "CV"`, the value of the ridge according to the 1 standard error rule: the most sparse model within one standard error of the model with the lowest MSPE
- **bestNcomp** The number of component with the best value for the chosen tuning index

For more details ?`mmscaModelSelection`. This procedure can be slow because the number of combinations can be great it takes a lot of time to evaluate them all in order to pick the best one. To that end we also provide an alternative way of tuning using `mmscaHyperCubeSelection`. This function tunes a grid of the tuning parameters determine by the min and max of their corresponding sequences and a step size the provided by `stepsize` argument. It picks out the best combination, and zooms in on that combination, by making a new smaller grid around the previous best combination. This process continues until the average range of the sequences is less than 0.05. The new sequences are determined by taking the minimum value to be: best value - range, and maximum value by: best value + range, and a pre-specified step size in `stepsize`. In order for this procedure to work well, the grid needs to include an optimal combination of tuning parameters, and it needs a reasonable step size (at least 3, 5 is better, 2 is too small). This approach assumes that a local optimum of tuning parameters is good enough to get interpretable results. Note that this function is experimental and has not been scrutinized using a simulation study. This procedure can be



performed as follows,

```
out <- mmscaHyperCubeSelection(X,
  ncomp = 3, # This function works with a fixed number of components
  ridgeSeq = 0:3,
  lassoSeq = 0:10,
  grouplassoSeq = 0,
  elitistlassoSeq = 0,
  stepsize = 5, # Step size of the sequences
  logscale = FALSE, # Sequences can be on the log-scale
  method = "CV1stdError", # Tuning method
  groups = ncol(X), # Arguments for mmsca()
  nStart = 1,
  itr = 100000,
  printProgress = TRUE,
  coordDes = FALSE,
  coordDesItr = 1,
  tol = 10e-5,
  nrFolds = 10)
```

This function returns a list with the following elements:

- **ridge** A vector with `ncomp` elements all equal to the chosen ridge value
- **lasso** A vector with `ncomp` elements all equal to the chosen lasso value
- **grouplasso** A vector with `ncomp` elements all equal to the chosen group lasso value
- **elitistlasso** A vector with `ncomp` elements all equal to the chosen elitist lasso value

For more details see `?mmscaHyperCubeSelection`.

## Detailed examples of SCA and PCA with the `sparseWeightBasedPCA` package

In this section we will give some detailed examples where we analyze multi- and single block data with SCA and PCA including the model selection process. We give an example of `scads`, `mmsca` and `ccpca` and the model selection functions. We will demonstrate these procedures using simulated data, for that purpose the package includes data generating functions that simulate data according to the following structure,

$$\mathbf{X} = \mathbf{X}\mathbf{W}\mathbf{P}^T, \quad (6)$$

where  $\mathbf{W}$  is  $J \times J$ ,  $\mathbf{W}^T\mathbf{W} = \mathbf{I}$  and  $\mathbf{W} = \mathbf{P}$ .  $\mathbf{W}$  is manipulated such that it contains a given level of sparsity in the first  $Q$  columns. The covariance matrix of  $\mathbf{X}$  can be constructed by taking  $\mathbf{\Sigma} = \mathbf{W}\mathbf{\Lambda}\mathbf{W}^T$ , the eigenvalues in  $\mathbf{\Lambda}$  can be manipulated to control the variance of the signal components versus the variance of the noise components. Using the covariance matrix data can be sampled from the multivariate normal distribution using `mvrnorm` from the package `MASS` (REF). Functions for data generation are provided by: `sparsify` to put sparsity in  $\mathbf{W}$ , `makeVariance` to manipulate the eigenvalues in  $\mathbf{\Lambda}$ , and `makeDat` to simulate the data. Check the package documentation for more details.

### Example of SCA with `scads`

Here we will demonstrate data analysis using `scads`. In this example we will have 2 data blocks each with 15 variables and 3 components. First we create a common and distinctive structure for the component weights to generate data from, we will use a structure with 2 distinctive components and 1 common component.

```
set.seed(1)
ncomp <- 3
J <- 30
comdis <- matrix(1, J, ncomp) # Component weight structure
comdis[1:15, 1] <- 0 # The first component is distinctive for the first block
comdis[16:30, 2] <- 0 # The second component is distinctive for the second block
comdis <- sparsify(comdis, 0.2) #set 20 percent of the 1's to zero
comdis
```

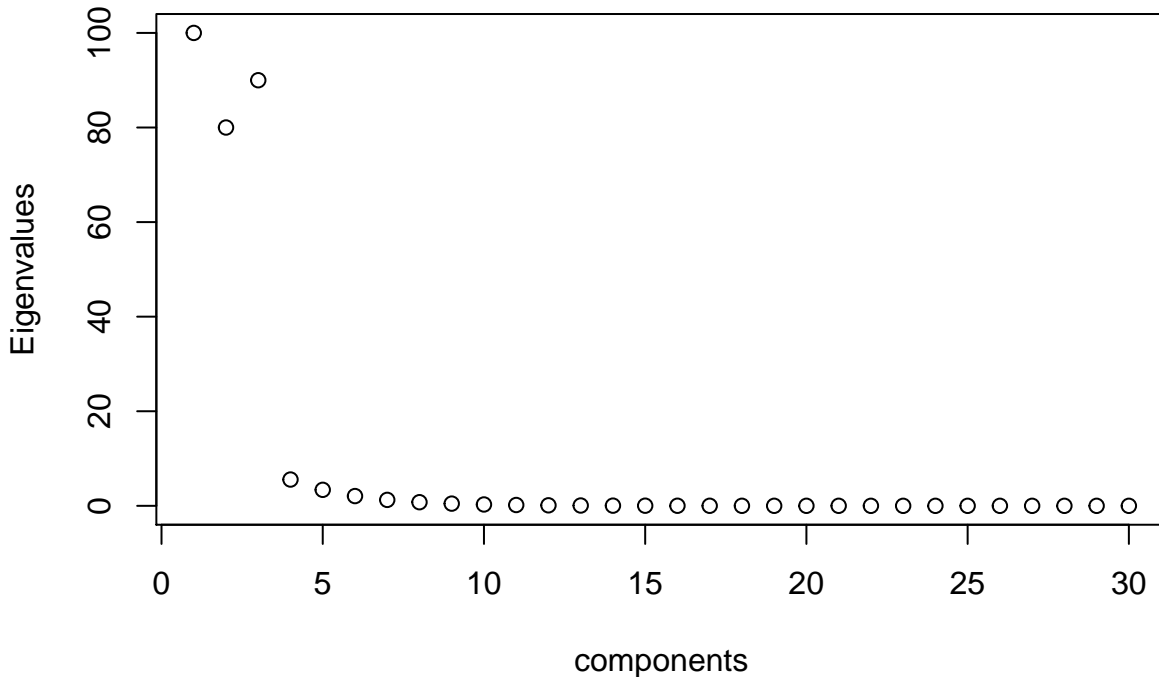
```
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    1
## [3,]    0    1    1
## [4,]    0    1    1
## [5,]    0    1    1
```

```
## [6,] 0 1 1
## [7,] 0 1 1
## [8,] 0 1 1
## [9,] 0 1 1
## [10,] 0 1 1
## [11,] 0 1 0
## [12,] 0 1 1
## [13,] 0 0 1
## [14,] 0 1 0
## [15,] 0 1 1
## [16,] 1 0 1
## [17,] 1 0 1
## [18,] 1 0 0
## [19,] 0 0 0
## [20,] 1 0 1
## [21,] 1 0 1
## [22,] 0 0 1
## [23,] 1 0 0
## [24,] 0 0 1
## [25,] 1 0 1
## [26,] 1 0 1
## [27,] 1 0 1
## [28,] 1 0 1
## [29,] 1 0 1
## [30,] 1 0 1
```

Now given this component weight structure we can simulate data as follows,

```
variances <- makeVariance(varianceOfComps = c(100, 80, 90),
                          J = J, error = 0.05) #create realistic eigenvalues
plot(variances, xlab = "components",
     ylab = "Eigenvalues", main = "Scree plot of the Eigenvalues X")
```

## Scree plot of the Eigenvalues X



```
dat <- makeDat(n = 100, comdis = comdis, variances = variances)
X <- dat$X
round(dat$P[, 1:ncomp], 3) # The data generating component weight structure
```

```
##      [,1]  [,2]  [,3]
```



```
## [1,] 0.000 0.000 0.000
## [2,] 0.000 0.000 0.072
## [3,] 0.000 -0.082 -0.004
## [4,] 0.000 0.517 0.149
## [5,] 0.000 0.165 0.121
## [6,] 0.000 -0.133 -0.226
## [7,] 0.000 -0.245 -0.103
## [8,] 0.000 -0.343 0.098
## [9,] 0.000 -0.313 0.233
## [10,] 0.000 -0.460 -0.086
## [11,] 0.000 0.340 0.000
## [12,] 0.000 0.245 -0.197
## [13,] 0.000 0.000 -0.418
## [14,] 0.000 0.078 0.000
## [15,] 0.000 -0.111 0.334
## [16,] 0.054 0.000 0.208
## [17,] 0.326 0.000 -0.162
## [18,] -0.023 0.000 0.000
## [19,] 0.000 0.000 0.000
## [20,] 0.441 0.000 0.158
## [21,] -0.280 0.000 0.004
## [22,] 0.000 0.000 0.365
## [23,] 0.019 0.000 0.000
## [24,] 0.000 0.000 0.378
## [25,] -0.220 0.000 -0.188
## [26,] -0.176 0.000 0.055
## [27,] -0.264 0.000 -0.177
## [28,] 0.295 0.000 0.085
## [29,] 0.468 0.000 -0.105
## [30,] 0.403 0.000 -0.203
```

In the scree plot you can see the eigenvalues of the data generating covariance matrix and `round(dat$P[, 1:ncomp], 3)` prints the data generating component weights for the signal components. Given this generate data set we can perform `scads` by first looking for the common and distinctive structure by trying all structures out and finding the best structure according to model selection, in this case we will use cross-validation. To generate all common and distinctive structures we implemented a function called `allCommonDistinctive` for more details see the documentation.

```
# Generate all possible common and distinctive structures
allstructures <- allCommonDistinctive(vars = c(15, 15), ncomp = 3, allPermutations = TRUE, filterZeroSegments =

#Use cross-validation to look for the data generating structure
index <- rep(NA, length(allstructures))
for (i in 1:length(allstructures)) {
  index[i] <- CVforPCAwithSparseWeights(X = X, nrFolds = 10, FUN = scads,
                                         ncomp, ridge = 0, lasso = rep(0, ncomp),
                                         constraints = allstructures[[i]], Wstart = matrix(0, J, ncomp),
                                         itr = 100000, nStarts = 1, printLoss = FALSE, tol = 10^-5)$MSPE
}

winningStructure <- allstructures[[which.min(index)]]
allstructures[[which.min(index)]] # print the best common and distinctive structure
```

```
##      [,1] [,2] [,3]
## [1,]    0    1    1
## [2,]    0    1    1
## [3,]    0    1    1
## [4,]    0    1    1
## [5,]    0    1    1
## [6,]    0    1    1
## [7,]    0    1    1
## [8,]    0    1    1
## [9,]    0    1    1
```

```
## [10,] 0 1 1
## [11,] 0 1 1
## [12,] 0 1 1
## [13,] 0 1 1
## [14,] 0 1 1
## [15,] 1 1 0
## [16,] 1 1 0
## [17,] 1 1 0
## [18,] 1 1 0
## [19,] 1 1 0
## [20,] 1 1 0
## [21,] 1 1 0
## [22,] 1 1 0
## [23,] 1 1 0
## [24,] 1 1 0
## [25,] 1 1 0
## [26,] 1 1 0
## [27,] 1 1 0
## [28,] 1 1 0
## [29,] 1 1 0
## [30,] 1 1 0
```

Given this common and distinctive structure we can tune the `lasso` parameter to get some sparsity inside the weights, for this we will use Cross validation with the one standard error rule. This entails cross validating the model a bunch of times for different values of the `lasso` parameter, from which we will select the model with the most weights at zero still within one standard error of the model with the least mean squared prediction error (MSPE). If there are not models within one standard error of the best model you should decrease the step size of the lasso sequence.

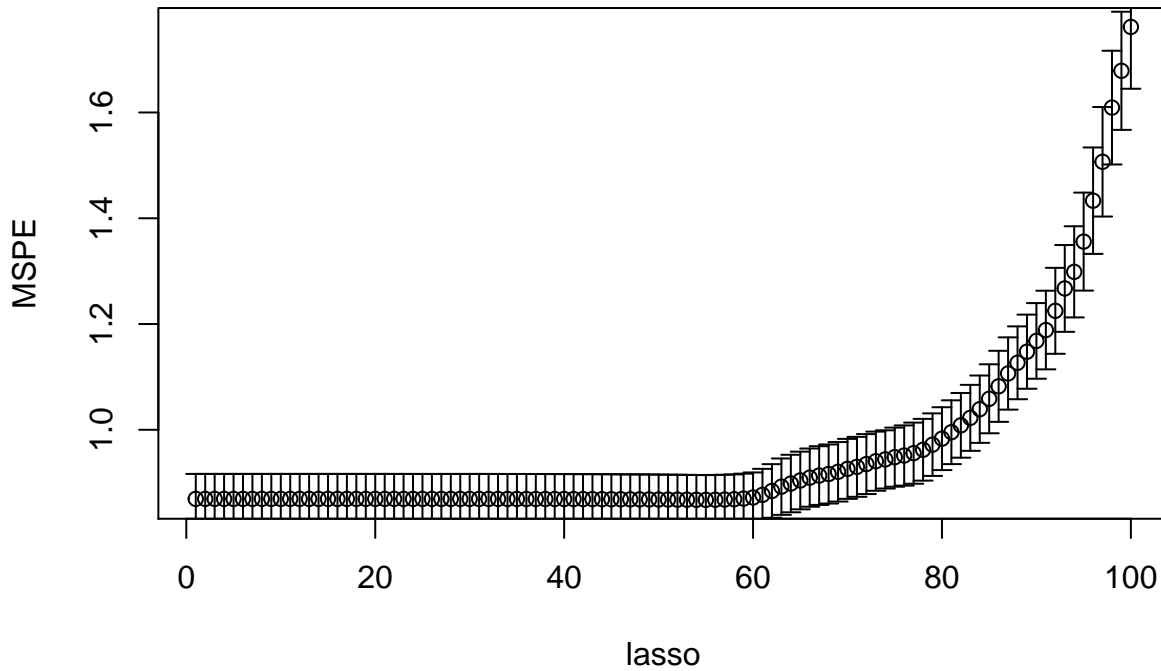
```
lasso <- exp(seq(log(0.0000001), log(1), length.out = 100)) # Generate candidate lasso values
MSPE <- rep(NA, length(lasso))
MSPEstdError <- rep(NA, length(lasso))
nNonZeroCoef <- rep(NA, length(lasso))

for (i in 1:length(lasso)) {
  res <- CVforPCAwithSparseWeights(X = X, nrFolds = 10, FUN = scads,
    ncomp, ridge = 0, lasso = rep(lasso[i], ncomp),
    constraints = winningStructure, Wstart = matrix(0, J, ncomp),
    itr = 100000, nStarts = 1, printLoss = FALSE, tol = 10^-5)
  MSPE[i] <- res$MSPE # Store MSPE for each lasso value
  MSPEstdError[i] <- res$MSPEstdError # Store the standard error of the MSPE
  nNonZeroCoef[i] <- res$nNonZeroCoef # Store the number of non-zero weights
}

x <- 1:length(lasso)
plot(x, MSPE, xlab = "lasso", ylab = "MSPE",
  main = "MSPE with one standard error for different lasso values")

# Add error bars to the plot
arrows(x, MSPE - MSPEstdError, x, MSPE + MSPEstdError, length = 0.05, angle = 90, code = 3)
```

## MSPE with one standard error for different lasso values



```
# Select all models within one standard error of the best model
eligibleModels <- MSPE < MSPE[which.min(MSPE)] + MSPEstdError[which.min(MSPE)]

# Selected from those models the models with the lowest number of non-zero weights
best <- which.min(nNonZeroCoef[eligibleModels])

# Do the analysis with the "winning" structure and best lasso
results <- scads(X = X, ncomp = ncomp,
  ridge = 0, lasso = rep(lasso[best], ncomp),
  constraints = allstructures[[which.min(index)]],
  Wstart = matrix(0, J, ncomp),
  itr = 100000, nStarts = 1, printLoss = FALSE, tol = 10^-5)

# Compare results from the analysis with the data generating model
compare <- cbind(dat$P[, 1:ncomp], results$W)
colnames(compare) <- c(paste("True W_", 1:3, sep = ""), paste("Est W_", 1:3, sep = ""))
rownames(compare) <- paste("Var", 1:30)
round(compare, 3)
```

##	True W_1	True W_2	True W_3	Est W_1	Est W_2	Est W_3
## Var 1	0.000	0.000	0.000	0.000	0.000	0.005
## Var 2	0.000	0.000	0.072	0.000	0.000	0.000
## Var 3	0.000	-0.082	-0.004	0.000	0.000	0.000
## Var 4	0.000	0.517	0.149	0.000	-0.021	-0.529
## Var 5	0.000	0.165	0.121	0.000	-0.138	-0.057
## Var 6	0.000	-0.133	-0.226	0.000	0.351	0.032
## Var 7	0.000	-0.245	-0.103	0.000	0.046	0.211
## Var 8	0.000	-0.343	0.098	0.000	-0.315	0.452
## Var 9	0.000	-0.313	0.233	0.000	-0.194	0.382
## Var 10	0.000	-0.460	-0.086	0.000	0.261	0.498
## Var 11	0.000	0.340	0.000	0.000	0.000	-0.328
## Var 12	0.000	0.245	-0.197	0.000	0.000	-0.178
## Var 13	0.000	0.000	-0.418	0.000	0.604	0.002
## Var 14	0.000	0.078	0.000	0.000	0.000	0.000
## Var 15	0.000	-0.111	0.334	0.000	-0.276	0.000
## Var 16	0.054	0.000	0.208	0.000	-0.279	0.000
## Var 17	0.326	0.000	-0.162	0.357	0.000	0.000

## Var 18	-0.023	0.000	0.000	0.000	0.000	0.000
## Var 19	0.000	0.000	0.000	0.000	0.000	0.000
## Var 20	0.441	0.000	0.158	0.446	-0.190	0.000
## Var 21	-0.280	0.000	0.004	-0.234	-0.042	0.000
## Var 22	0.000	0.000	0.365	-0.157	-0.536	0.000
## Var 23	0.019	0.000	0.000	0.000	0.000	0.000
## Var 24	0.000	0.000	0.378	-0.189	-0.232	0.000
## Var 25	-0.220	0.000	-0.188	-0.237	0.191	0.000
## Var 26	-0.176	0.000	0.055	-0.354	0.000	0.000
## Var 27	-0.264	0.000	-0.177	-0.164	0.137	0.000
## Var 28	0.295	0.000	0.085	0.177	-0.050	0.000
## Var 29	0.468	0.000	-0.105	0.479	0.000	0.000
## Var 30	0.403	0.000	-0.203	0.423	-0.005	0.000

This concludes the analysis of multi-block data with **scads**. This procedure offers lots of flexibility to the user at the cost of a little more complexity (i.e. the user has to know for-loops and some **R** basics).

#### Example of SCA with **mmsca**