Package ‘CMLS’

June 10, 2018

Type Package

Title Constrained Multivariate Least Squares

Version 1.0-0

Date 2018-06-06

Author Nathaniel E. Helwig <helwig@umn.edu>

Maintainer Nathaniel E. Helwig <helwig@umn.edu>

Depends quadprog, parallel

Description Solves multivariate least squares (MLS) problems subject to constraints on the coefficients, e.g., non-negativity, orthogonality, equality, inequality, monotonicity, unimodality, smoothness, etc. Includes flexible functions for solving MLS problems subject to user-specified equality and/or inequality constraints, as well as a wrapper function that implements 24 common constraint options. Also does k-fold or generalized cross-validation to tune constraint options for MLS problems. See ten Berge (1993, ISBN:9789066950832) for an overview of MLS problems, and see Goldfarb and Idnani (1983) <doi:10.1007/BF02591962> for a discussion of the underlying quadratic programming algorithm.

License GPL (>= 2)

NeedsCompilation no

Repository CRAN

Date/Publication 2018-06-10 15:17:57 UTC

R topics documented:

CMLS-package .......................................................... 2
cmls ................................................................. 3
const ............................................................... 11
cv.cmls .......................................................... 12
mlsei ............................................................... 15
mlsun ............................................................. 18

Index 22
Description

Solves multivariate least squares (MLS) problems subject to constraints on the coefficients, e.g.,
negativity, orthogonality, equality, inequality, monotonicity, unimodality, smoothness, etc. In-
cludes flexible functions for solving MLS problems subject to user-specified equality and/or in-
nequality constraints, as well as a wrapper function that implements 24 common constraint options.
Also does k-fold or generalized cross-validation to tune constraint options for MLS problems. See
Berger (1993, ISBN:9789066950832) for an overview of MLS problems, and see Goldfarb and
Idnani (1983) <doi:10.1007/BF02591962> for a discussion of the underlying quadratic program-
ing algorithm.

Details

The DESCRIPTION file:

Package: CMLS
Type: Package
Title: Constrained Multivariate Least Squares
Version: 1.0-0
Date: 2018-06-06
Author: Nathaniel E. Helwig <helwig@umn.edu>
Maintainer: Nathaniel E. Helwig <helwig@umn.edu>
Depends: quadprog, parallel
Description: Solves multivariate least squares (MLS) problems subject to constraints on the coefficients, e.g., non-negativity
License: GPL (>=2)

Index of help topics:

- CMLS-package
- cmls
- const
- cv.cmls
- mlsei
- mlsun

The cmls function provides a user-friendly interface for solving the MLS problem with 24 com-
mon constraint options (the const function prints or returns the different constraint options). The
cv.cmls function does k-fold or generalized cross-validation to tune the constraint options of the
cmls function. The mlsei function solves the MLS problem subject to user-specified equality
and/or inequality (E/I) constraints on the coefficients. The \texttt{mlsun} function solves the MLS problem subject to unimodality constraints and user-specified E/I constraints on the coefficients.

Author(s)
Nathaniel E. Helwig <helwig@umn.edu>
Maintainer: Nathaniel E. Helwig <helwig@umn.edu>

References
Helwig, N. E. (in prep). Constrained multivariate least squares in R.

Examples

```r
# See examples for cmls, cv.cmls, mlsei, and mlsun
```

\section*{cmls
\textit{Solve a Constrained Multivariate Least Squares Problem}}

\section*{Description}
Finds the \( p \times m \) matrix \( B \) that minimizes the multivariate least squares problem

\[ \text{sum}((Y - X \times B)^2) \]

subject to the specified constraints on the rows of \( B \).

\section*{Usage}
\begin{verbatim}
cmls(X, Y, const = "uncons", struc = NULL,
df = 10, degree = 3, intercept = TRUE, 
backfit = FALSE, maxit = 1e3, eps = 1e-10, 
del = 1e-6, XTX = NULL, mode.range = NULL)
\end{verbatim}

\section*{Arguments}
\begin{itemize}
  \item \texttt{X} Matrix of dimension \( n \times p \).
  \item \texttt{Y} Matrix of dimension \( n \times m \).
  \item \texttt{const} Constraint code. See \texttt{const} for the 24 available options.
\end{itemize}
Structural constraints (defaults to unstructured). See Note.

Degrees of freedom for the spline basis (for smoothness constraints). See Note.

Polynomial degree for the spline basis (for smoothness constraints). See Note.

Logical indicating whether the spline basis should contain an intercept (for smoothness constraints). See Note.

Estimate $B$ via back-fitting (TRUE) or vectorization (FALSE). See Details.

Maximum number of iterations for back-fitting algorithm. Ignored if $\text{backfit} = \text{FALSE}$.

Convergence tolerance for back-fitting algorithm. Ignored if $\text{backfit} = \text{FALSE}$.

Stability tolerance for back-fitting algorithm. Ignored if $\text{backfit} = \text{FALSE}$.

Crossproduct matrix: $\text{xtx} = \text{crossprod}(X)$.

Mode search ranges (for unimodal constraints). See Note.

If $\text{backfit} = \text{FALSE}$ (default), a closed-form solution is used to estimate $B$ whenever possible. Otherwise a back-fitting algorithm is used, where the rows of $B$ are updated sequentially until convergence. The backfitting algorithm is determined to have converged when

$$\text{mean}((B.\text{new} - B.\text{old})^2) < \text{eps} \times (\text{mean}(B.\text{old}^2) + \text{del}),$$

where $B.\text{old}$ and $B.\text{new}$ denote the parameter estimates at iterations $t$ and $t + 1$ of the backfitting algorithm.

Returns the estimated matrix $B$ with attribute "df" (degrees of freedom), which gives the df for each row of $B$.

Structure constraints ($\text{struc}$) should be specified with a $p \times m$ matrix of logicals (TRUE/FALSE), such that FALSE elements indicate a weight should be constrained to be zero. Default uses unstructured weights, i.e., a $p \times m$ matrix of all TRUE values.

Inputs $\text{df}$, degree, and $\text{intercept}$ are only applicable when using one of the 12 constraints that involves a spline basis, i.e., "smooth", "smonon", "smoper", "smpeno", "ortsmo", "orsmpe", "mon-smo", "mosmno", "unismo", "unsmno", "unsmpo", "unsmpe", "unsmpn".

Input $\text{mode.range}$ is only applicable when using one of the 8 constraints that enforces unimodality: "unimod", "uninon", "uniper", "unpeno", "unismo", "unsmno", "unsmpo", "unsmpe", "unsmpn". Mode search ranges ($\text{mode.range}$) should be specified with a $2 \times p$ matrix of integers such that

$$1 \leq \text{mode.range}[1,j] \leq \text{mode.range}[2,j] \leq m \text{ for all } j = 1:p.$$ 

Default is $\text{mode.range} = \text{matrix}(c(1, m), 2, p)$.

Nathaniel E. Helwig <helwig@umn.edu>
References


Helwig, N. E. (in prep). Constrained multivariate least squares in R.


See Also

- const prints/returns the contraint options.
- cv.cmls performs k-fold cross-validation to tune the constraint options.
- mlsei and mlsun are used to implement several of the constraints.

Examples

`cmls` 5

### GENERATE DATA

```r
# make X
set.seed(2)
n <- 50
m <- 20
p <- 2
Xmat <- matrix(rnorm(n*p), nrow = n, ncol = p)

# make B (which satisfies all constraints except monotonicity)
x <- seq(0, 1, length.out = m)
Bmat <- rbind(sin(2*pi*x), sin(2*pi*x+pi)) / sqrt(4.75)
struc <- rbind(rep(c(TRUE, FALSE), each = m / 2),
               rep(c(FALSE, TRUE), each = m / 2))
Bmat <- Bmat * struc

# make noisy data
Ymat <- Xmat %*% Bmat + rnorm(n*m, sd = 0.25)
```

### UNCONSTRAINED

```r
# unconstrained
Bhat <- cmls(X = Xmat, Y = Ymat, const = "uncons")
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# unconstrained and structured
Bhat <- cmls(X = Xmat, Y = Ymat, const = "uncons", struc = struc)
mean((Bhat - Bmat)^2)
attr(Bhat, "df")
```
# non-negative
Bhat <- cmls(X = Xmat, Y = Ymat, const = "nonneg")
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# non-negative and structured
Bhat <- cmls(X = Xmat, Y = Ymat, const = "nonneg", struc = struc)
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# periodic
Bhat <- cmls(X = Xmat, Y = Ymat, const = "period")
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# periodic and structured
Bhat <- cmls(X = Xmat, Y = Ymat, const = "period", struc = struc)
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# periodic and non-negative
Bhat <- cmls(X = Xmat, Y = Ymat, const = "pernon")
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# periodic and non-negative and structured
Bhat <- cmls(X = Xmat, Y = Ymat, const = "pernon", struc = struc)
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# smooth
Bhat <- cmls(X = Xmat, Y = Ymat, const = "smooth")
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# smooth and structured
Bhat <- cmls(X = Xmat, Y = Ymat, const = "smooth", struc = struc)
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# smooth and periodic
Bhat <- cmls(X = Xmat, Y = Ymat, const = "smoper")
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# smooth and periodic and structured
Bhat <- cmls(X = Xmat, Y = Ymat, const = "smoper", struc = struc)
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# smooth and non-negative
Bhat <- cmls(X = Xmat, Y = Ymat, const = "smonon")
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# smooth and non-negative and structured
Bhat <- cmls(X = Xmat, Y = Ymat, const = "smonon", struc = struc)
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# smooth and periodic and non-negative
Bhat <- cmls(X = Xmat, Y = Ymat, const = "smpeno")
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# smooth and periodic and non-negative and structured
Bhat <- cmls(X = Xmat, Y = Ymat, const = "smpeno", struc = struc)
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

######### ORTHOGONALITY #########

# orthogonal
Bhat <- cmls(X = Xmat, Y = Ymat, const = "orthog")
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# orthogonal and structured
Bhat <- cmls(X = Xmat, Y = Ymat, const = "orthog", struc = struc)
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# orthogonal and non-negative
Bhat <- cmls(X = Xmat, Y = Ymat, const = "ortnon")
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# orthogonal and non-negative and structured
Bhat <- cmls(X = Xmat, Y = Ymat, const = "ortnon", struc = struc)
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# orthogonal and smooth
Bhat <- cmls(X = Xmat, Y = Ymat, const = "ortsmo")
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# unimodal and periodic and non-negative and structured
Bhat <- cmls(X = Xmat, Y = Ymat, const = "unpeno", struc = struc)
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# UNIMODALITY AND SMOOTHNESS

# unimodal and smooth
Bhat <- cmls(X = Xmat, Y = Ymat, const = "unismo")
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# unimodal and smooth and structured
Bhat <- cmls(X = Xmat, Y = Ymat, const = "unismo", struc = struc)
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# unimodal and smooth and non-negative
Bhat <- cmls(X = Xmat, Y = Ymat, const = "unsmno")
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# unimodal and smooth and non-negative and structured
Bhat <- cmls(X = Xmat, Y = Ymat, const = "unsmno", struc = struc)
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# unimodal and smooth and periodic
Bhat <- cmls(X = Xmat, Y = Ymat, const = "unsmpe")
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# unimodal and smooth and periodic and structured
Bhat <- cmls(X = Xmat, Y = Ymat, const = "unsmpe", struc = struc)
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# unimodal and smooth and periodic and non-negative
Bhat <- cmls(X = Xmat, Y = Ymat, const = "unsmpn")
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# unimodal and smooth and periodic and non-negative and structured
Bhat <- cmls(X = Xmat, Y = Ymat, const = "unsmpn", struc = struc)
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# MONOTONICITY

# unimodal and periodic and non-negative and structured
Bhat <- cmls(X = Xmat, Y = Ymat, const = "unpeno", struc = struc)
mean((Bhat - Bmat)^2)
attr(Bhat, "df")
```r
# make B
x <- 1:m
Bmat <- rbind((1 / (1 + exp(-(x - quantile(x, 0.5))))),
               (1 / (1 + exp(-(x - quantile(x, 0.8)))))
struc <- rbind(rep(c(FALSE, TRUE), c(1 * m, 3 * m) / 4),
               rep(c(FALSE, TRUE), c(m, m) / 2))
Bmat <- Bmat * struc

# make noisy data
set.seed(1)
Ymat <- Xmat %*% Bmat + rnorm(m*n, sd = 0.25)

# monotonic increasing
Bhat <- cmls(X = Xmat, Y = Ymat, const = "moninc")
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# monotonic increasing and structured
Bhat <- cmls(X = Xmat, Y = Ymat, const = "moninc", struc = struc)
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# monotonic increasing and non-negative
Bhat <- cmls(X = Xmat, Y = Ymat, const = "monnon")
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# monotonic increasing and non-negative and structured
Bhat <- cmls(X = Xmat, Y = Ymat, const = "monnon", struc = struc)
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# monotonic increasing and smooth
Bhat <- cmls(X = Xmat, Y = Ymat, const = "monsmo")
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# monotonic increasing and smooth and structured
Bhat <- cmls(X = Xmat, Y = Ymat, const = "monsmo", struc = struc)
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# monotonic increasing and smooth and non-negative
Bhat <- cmls(X = Xmat, Y = Ymat, const = "mosmno")
mean((Bhat - Bmat)^2)
attr(Bhat, "df")

# monotonic increasing and smooth and non-negative and structured
Bhat <- cmls(X = Xmat, Y = Ymat, const = "mosmno", struc = struc)
mean((Bhat - Bmat)^2)
attr(Bhat, "df")
```
Description
Prints or returns six letter constraint codes for `cmls`, along with corresponding descriptions.

Usage
`const(x, print = TRUE)`

Arguments
- `x` Vector of six letter constraint codes. If missing, prints/returns all 24 options.
- `print` Should constraint information be printed (`print = TRUE`) or returned as a data frame (`print = FALSE`).

Value
Prints (or returns) constraint codes and descriptions.

Author(s)
Nathaniel E. Helwig <helwig@umn.edu>

References
Helwig, N. E. (in prep). Constrained multivariate least squares in R.

See Also
Constraints are used in the `cmls` function.

Examples
```r
# print some constraints
c(const(c("uncons", "smpeno")))

# return some constraints
c(const(c("uncons", "smpeno"), print = FALSE))

# print all constraints
c(const())

# return all constraints
c(const(print = FALSE))
```
cv.cmls

Cross-Validation for cmls

Description

Does k-fold or generalized cross-validation to tune the constraint options for cmls. Tunes the model with respect to any combination of the arguments const, df, degree, and/or intercept.

Usage

cv.cmls(X, Y, nfolds = 2, foldid = NULL, parameters = NULL,
  const = "uncons", df = 10, degree = 3, intercept = TRUE,
  mse = TRUE, parallel = FALSE, cl = NULL, verbose = TRUE, ...)

Arguments

X          Matrix of dimension \(n \times p\).
Y          Matrix of dimension \(n \times m\).
nfolds     Number of folds for k-fold cross-validation. Ignored if foldid argument is provided. Set nfolds=1 for generalized cross-validation (GCV).
foldid     Factor or integer vector of length \(n\) giving the fold identification for each observation.
parameters Parameters for tuning. Data frame with columns const, df, degree, and intercept. See Details.
const      Parameters for tuning. Character vector specifying constraints for tuning. See Details.
df         Parameters for tuning. Integer vector specifying degrees of freedom for tuning. See Details.
degree     Parameters for tuning. Integer vector specifying polynomial degrees for tuning. See Details.
intercept  Parameters for tuning. Logical vector specifying intercepts for tuning. See Details.
mse        If TRUE (default), the mean squared error is used as the CV loss function. Otherwise the mean absolute error is used.
parallel   Logical indicating if parsapply should be used. See Examples.
cl          Cluster created by makeCluster. Only used when parallel = TRUE.
verbose     If TRUE, tuning progress is printed via txtProgressBar. Ignored if parallel = TRUE.
...         Additional arguments to the cmls function, e.g., struc, backfit, etc.
Details

The parameters for tuning can be supplied via one of two options:

(A) Using the parameters argument. In this case, the argument parameters must be a data frame with columns const, df, degree, and intercept, where each row gives a combination of parameters for the CV tuning.

(B) Using the const, df, degree, and intercept arguments. In this case, the expand.grid function is used to create the parameters data frame, which contains all combinations of the arguments const, df, degree, and intercept. Duplicates are removed before the CV tuning.

Value

best.parameters
Best combination of parameters, i.e., the combination that minimizes the cvloss.

top5.parameters
Top five combinations of parameters, i.e., the combinations that give the five smallest values of the cvloss.

full.parameters
Full set of parameters. Data frame with cvloss (GCV, MSE, or MAE) for each combination of parameters.

Author(s)

Nathaniel E. Helwig <helwig@umn.edu>

References

Helwig, N. E. (in prep). Constrained multivariate least squares in R.

See Also

See the cmls and const functions for further details on the available constraint options.

Examples

# make X
set.seed(1)
n <- 50
m <- 20
p <- 2
Xmat <- matrix(rnorm(n*p), nrow = n, ncol = p)

# make B (which satisfies all constraints except monotonicity)
x <- seq(0, 1, length.out = m)
Bmat <- rbind(sin(2*pi*x), sin(2*pi*x*pi)) / sqrt(4.75)
struc <- rbind(rep(c(TRUE, FALSE), each = m / 2),
               rep(c(FALSE, TRUE), each = m / 2))
Bmat <- Bmat * struc
# make noisy data
Ymat <- Xmat %*% Bmat + rnorm(n*m, sd = 0.5)

# 5-fold CV: tune df (5,...,15) for const = "smooth"
kcv <- cv.cmls(X = Xmat, Y = Ymat, nfolds = 5,
               const = "smooth", df = 5:15)
kcv$best.parameters
ekcv$top5.parameters
plot(kcv$full.parameters$df, kcv$full.parameters$cvloss, t = "b")

# sample foldid for 5-fold CV
set.seed(2)
foldid <- sample(rep(1:5, length.out = n))

# 5-fold CV: tune df (5,...,15) w/ all 20 relevant constraints (no struc)
# using sequential computation (default)
myconst <- as.character(const(print = FALSE)$label[-c(13:16)])
system.time(
  kcv <- cv.cmls(X = Xmat, Y = Ymat, foldid = foldid,
                 const = myconst, df = 5:15)
)
kcv$best.parameters
kcv$top5.parameters

# 5-fold CV: tune df (5,...,15) w/ all 20 relevant constraints (no struc)
# using parallel package for parallel computations
myconst <- as.character(const(print = FALSE)$label[-c(13:16)])
system.time(
  cl <- makeCluster(detectCores())
  kcv <- cv.cmls(X = Xmat, Y = Ymat, foldid = foldid,
                 const = myconst, df = 5:15,
                 parallel = TRUE, cl = cl)
  stopCluster(cl)
)
kcv$best.parameters
kcv$top5.parameters

# 5-fold CV: tune df (5,...,15) w/ all 20 relevant constraints (w/ struc)
# using sequential computation (default)
myconst <- as.character(const(print = FALSE)$label[-c(13:16)])
system.time(
  kcv <- cv.cmls(X = Xmat, Y = Ymat, foldid = foldid,
                 const = myconst, df = 5:15, struc = struc)
)
kcv$best.parameters
kcv$top5.parameters
mlsei

Multivariate Least Squares with Equality/Inequality Constraints

Description

Finds the $q \times p$ matrix $B$ that minimizes the multivariate least squares problem

$$\text{sum}( (Y - X \times B)^2 )$$

subject to $t(A) \times B[j] \geq b$ for all $j = 1:p$. Unique basis functions and constraints are allowed for each column of $B$.

Usage

mlsei(X, Y, Z, A, meq, meq = FALSE, maxit = 1000, eps = 1e-10, del = 1e-6, X = NULL, Z = NULL, simplify = TRUE, catchError = FALSE)

Arguments

X Matrix of dimension $n \times p$.
Y Matrix of dimension $n \times m$.
Z Matrix of dimension $m \times q$. Can also input a list (see Note). If missing, then $Z = \text{diag}(m)$ so that $q = m$.
A Constraint matrix of dimension $q \times r$. Can also input a list (see Note). If missing, no constraints are imposed.
Constraint vector of dimension $r \times 1$. Can also input a list (see Note). If missing, then $b = \text{rep}(\emptyset, r)$.

The first $meq$ columns of $A$ are equality constraints, and the remaining $r - meq$ are inequality constraints. Can also input a vector (see Note). If missing, then $meq = 0$.

Estimate $B$ via back-fitting (TRUE) or vectorization (FALSE). See Details.

Maximum number of iterations for back-fitting algorithm. Ignored if $backfit = \text{FALSE}$.

Convergence tolerance for back-fitting algorithm. Ignored if $backfit = \text{FALSE}$.

Stability tolerance for back-fitting algorithm. Ignored if $backfit = \text{FALSE}$.

Crossproduct matrix: $xtX = \text{crossprod}(X)$.

Crossproduct matrix: $ztZ = \text{crossprod}(Z)$.

If $Z$ is a list, should $B$ be returned as a matrix (if possible)? See Note.

If $catcherror = \text{FALSE}$, an error induced by $\text{solve.QP}$ will be returned. Otherwise $\text{tryCatch}$ will be used in attempt to catch the error.

If $backfit = \text{FALSE}$ (default), a closed-form solution is used to estimate $B$ whenever possible. Otherwise a back-fitting algorithm is used, where the columns of $B$ are updated sequentially until convergence. The backfitting algorithm is determined to have converged when

$$\text{mean}((B_{\text{new}} - B_{\text{old}})^2) < \text{eps} \times (\text{mean}(B_{\text{old}}^2) + \text{del}),$$

where $B_{\text{old}}$ and $B_{\text{new}}$ denote the parameter estimates at iterations $t$ and $t + 1$ of the backfitting algorithm.

If $Z$ is a list with $q_j = q$ for all $j = 1, \ldots, p$, then...

$B$ is returned as a $q \times p$ matrix when $\text{simplify} = \text{TRUE}$

$B$ is returned as a list of length $p$ when $\text{simplify} = \text{FALSE}$

If $Z$ is a list with $q_j \neq q$ for some $j$, then $B$ is returned as a list of length $p$.

Otherwise $B$ is returned as a $q \times p$ matrix.

The $Z$ input can also be a list of length $p$ where $Z[[j]]$ contains a $m \times q_j$ matrix. If $q_j = q$ for all $j = 1, \ldots, p$ and $\text{simplify} = \text{TRUE}$, the output $B$ will be a matrix. Otherwise $B$ will be a list of length $p$ where $B[[j]]$ contains a $q_j \times 1$ vector.

The $A$ and $b$ inputs can also be lists of length $p$ where $t(A[[j]]) \%\% B[.,j] \geq b[[j]]$ for all $j = 1, \ldots, p$. If $A$ and $b$ are lists of length $p$, the $meq$ input should be a vector of length $p$ indicating the number of equality constraints for each element of $A$.

Nathaniel E. Helwig <helwig@umn.edu>
References
Helwig, N. E. (in prep). Constrained multivariate least squares in R.

See Also
cmls calls this function for several of the constraints.

Examples

```
# make X
set.seed(2)
n <- 50
m <- 20
p <- 2
Xmat <- matrix(rnorm(n*p), nrow = n, ncol = p)

# make B (which satisfies all constraints except monotonicity)
x <- seq(0, 1, length.out = m)
Bmat <- rbind(sin(2*pi*x), sin(2*pi*x+pi)) / sqrt(4.75)
struc <- rbind(rep(c(TRUE, FALSE), each = m/2),
               rep(c(FALSE, TRUE), each = m/2))
Bmat <- Bmat * struc

# make noisy data
set.seed(1)
Ymat <- Xmat %*% Bmat + rnorm(n*m, sd = 0.25)

# unconstrained
Bhat.cmls <- cmls(X = Xmat, Y = Ymat, const = "uncons")
Bhat.mlsei <- t(mlsei(X = Xmat, Y = Ymat))
mean((Bhat.cmls - Bhat.mlsei)^2)

# unconstrained and structured (note: cmls is more efficient)
Bhat.cmls <- cmls(X = Xmat, Y = Ymat, const = "uncons", struc = struc)
Amat <- vector("list", p)
meq <- rep(0, p)
for(j in 1:p){
  meq[j] <- sum(!struc[j])
  if(meq[j] > 0){
```
A <- matrix(0, nrow = m, ncol = meq[j])
A[!struc[j,],] <- diag(meq[j])
Amat[[j]] <- A
}

else {
  Amat[[j]] <- matrix(0, nrow = m, ncol = 1)
}

Bhat.mlsei <- t(mlsei(X = Xmat, Y = Ymat, A = Amat, meq = meq))
mean((Bhat.cmls - Bhat.mlsei)^2)

### NON-NEGATIVITY ###

# non-negative
Bhat.cmls <- cmls(X = Xmat, Y = Ymat, const = "nonneg")
Bhat.mlsei <- t(mlsei(X = Xmat, Y = Ymat, A = diag(m)))
mean((Bhat.cmls - Bhat.mlsei)^2)

# non-negative and structured (note: cmls is more efficient)
Bhat.cmls <- cmls(X = Xmat, Y = Ymat, const = "nonneg", struc = struc)

eye <- diag(m)
meq <- rep(0, p)
for(j in 1:p){
  meq[j] <- sum(!struc[j,])
  Amat[[j]] <- eye[,sort(struc[j,], index.return = TRUE)$ix]
}

Bhat.mlsei <- t(mlsei(X = Xmat, Y = Ymat, A = Amat, meq = meq))
mean((Bhat.cmls - Bhat.mlsei)^2)

# see internals of cmls.R for further examples

---

**mlsun**

*Multivariate Least Squares with Unimodality (and E/I) Constraints*

**Description**

Finds the \( q \times p \) matrix \( B \) that minimizes the multivariate least squares problem

\[
\sum((Y - X) %*% t(Z %*% B))^2
\]

subject to \( Z %*% B[j,] \) is unimodal and \( t(A) %*% B[j,] \geq b \) for all \( j = 1:p \). Unique basis functions and constraints are allowed for each column of \( B \).

**Usage**

```r
mlsun(X, Y, Z, A, b, meq,
      mode.range = NULL, maxit = 1000,
```
eps = 1e-10, del = 1e-6,
XtX = NULL, ZtZ = NULL,
simplify = TRUE, catchError = FALSE)

Arguments

X Matrix of dimension n x p.
Y Matrix of dimension n x m.
Z Matrix of dimension m x q. Can also input a list (see Note). If missing, then Z = diag(m) so that q = m.
A Constraint matrix of dimension q x r. Can also input a list (see Note). If missing, no equality/inequality (E/I) constraints are imposed.
b Constraint vector of dimension r x 1. Can also input a list (see Note). If missing, then b = rep(0, r).
meq The first meq columns of A are equality constraints, and the remaining r - meq are inequality constraints. Can also input a vector (see Note). If missing, then meq = 0.
mode.range Mode search ranges, which should be a 2 x p matrix of integers such that 1 <= mode.range[1,j] <= mode.range[2,j] <= m for all j = 1:p. Default is mode.range = matrix(c(1, m), 2, p).
maxit Maximum number of iterations for back-fitting algorithm. Ignored if backfit = FALSE.
eps Convergence tolerance for back-fitting algorithm. Ignored if backfit = FALSE.
del Stability tolerance for back-fitting algorithm. Ignored if backfit = FALSE.
XtX Crossproduct matrix: XtX = crossprod(X).
ZtZ Crossproduct matrix: ZtZ = crossprod(Z).
simplify If Z is a list, should B be returned as a matrix (if possible)? See Note.
catchError If catchError = FALSE, an error induced by solve.QP will be returned. Otherwise tryCatch will be used in attempt to catch the error.

Details

A back-fitting algorithm is used to estimate B, where the columns of B are updated sequentially until convergence (outer loop). For each column of B, (the inner loop of) the algorithm searches for the j-th mode across the search range specified by the j-th column of mode.range. The backfitting algorithm is determined to have converged when

\[ \text{mean}((B.\text{new} - B.\text{old})^2) < \text{eps} * (\text{mean}(B.\text{old}^2) + \text{del}), \]

where B.\text{old} and B.\text{new} denote the parameter estimates at outer iterations t and t + 1 of the backfitting algorithm.

Value

If Z is a list with q_j = q for all j = 1, ..., p, then...

B is returned as a q x p matrix when simplify = TRUE
B is returned as a list of length p when simplify = FALSE
If $Z$ is a list with $q_j \neq q$ for some $j$, then $B$ is returned as a list of length $p$.
Otherwise $B$ is returned as a $q \times p$ matrix.

**Note**

The $Z$ input can also be a list of length $p$ where $Z[[j]]$ contains a $m \times q_j$ matrix. If $q_j = q$ for all $j = 1, \ldots, p$ and `simplify = TRUE`, the output $B$ will be a matrix. Otherwise $B$ will be a list of length $p$ where $B[[j]]$ contains a $q_j \times 1$ vector.

The $A$ and $b$ inputs can also be lists of length $p$ where $t(A[[j]]) \%\% b[,j] \geq b[[j]]$ for all $j = 1, \ldots, p$. If $A$ and $b$ are lists of length $p$, the `meq` input should be a vector of length $p$ indicating the number of equality constraints for each element of $A$.

**Author(s)**

Nathaniel E. Helwig <helwig@umn.edu>

**References**


Helwig, N. E. (in prep). Constrained multivariate least squares in R.


**See Also**

cmls calls this function for the unimodality constraints.

**Examples**

### GENERATE DATA

```r
# make X
set.seed(2)
n <- 50
m <- 20
p <- 2
Xmat <- matrix(rnorm(n*p), nrow = n, ncol = p)

# make B (which satisfies all constraints except monotonicity)
x <- seq(0, 1, length.out = m)
Bmat <- rbind(sin(2*pi*x), sin(2*pi*x*pi)) / sqrt(4.75)
struc <- rbind(rep(c(TRUE, FALSE), each = m / 2),
               rep(c(FALSE, TRUE), each = m / 2))
Bmat <- Bmat * struc

# make noisy data
```
```r
set.seed(1)
Ymat <- Xmat %*% Bmat + rnorm(n*m, sd = 0.25)

########### UNIMODALITY ###########

# unimodal
Bhat.cmls <- cmls(X = Xmat, Y = Ymat, const = "unimod")
Bhat.ml.sun <- t(mlsun(X = Xmat, Y = Ymat))
mean((Bhat.cmls - Bhat.ml.sun)^2)

# unimodal and structured
Bhat.cmls <- cmls(X = Xmat, Y = Ymat, const = "unimod", struc = struc)
Amat <- vector("list", p)
meq <- rep(0, p)
for(j in 1:p){
  meq[j] <- sum(!struc[j,])
  if(meq[j] > 0){
    A <- matrix(0, nrow = m, ncol = meq[j])
    A[!struc[j,]] <- diag(meq[j])
    Amat[[j]] <- A
  } else {
    Amat[[j]] <- matrix(0, nrow = m, ncol = 1)
  }
}
Bhat.ml.sun <- t(mlsun(X = Xmat, Y = Ymat, A = Amat, meq = meq))
mean((Bhat.cmls - Bhat.ml.sun)^2)

# unimodal and non-negative
Bhat.cmls <- cmls(X = Xmat, Y = Ymat, const = "uninon")
Bhat.ml.sun <- t(mlsun(X = Xmat, Y = Ymat, A = diag(m)))
mean((Bhat.cmls - Bhat.ml.sun)^2)

# unimodal and non-negative and structured
Bhat.cmls <- cmls(X = Xmat, Y = Ymat, const = "uninon", struc = struc)
eye <- diag(m)
meq <- rep(0, p)
for(j in 1:p){
  meq[j] <- sum(!struc[j,])
  Amat[[j]] <- eye[,sort(struc[j,], index.return = TRUE)$ix]
}
Bhat.ml.sun <- t(mlsun(X = Xmat, Y = Ymat, A = Amat, meq = meq))
mean((Bhat.cmls - Bhat.ml.sun)^2)

# see internals of cmls.R for further examples
```
Index

*Topic **models**
  cmls, 3
  cv.cmls, 12
  mlsei, 15
  mlsun, 18

*Topic **multivariate**
  cmls, 3
  cv.cmls, 12
  mlsei, 15
  mlsun, 18

*Topic **optimize**
  cmls, 3
  cv.cmls, 12
  mlsei, 15
  mlsun, 18

*Topic **package**
  CMLS-package, 2

*Topic **regression**
  cmls, 3
  cv.cmls, 12
  mlsei, 15
  mlsun, 18

*Topic **smooth**
  cmls, 3
  cv.cmls, 12
  mlsei, 15
  mlsun, 18

CMLS (CMLS-package), 2
  cmls, 2, 3, 11–13, 17, 20
  CMLS-package, 2
  const, 2, 3, 5, 11, 13
  cv.cmls, 2, 5, 12
  expand.grid, 13

makeCluster, 12
  mlsei, 2, 5, 15
  mlsun, 3, 5, 18
  parSapply, 12

solve.QP, 16, 19
tryCatch, 16, 19
txtProgressBar, 12