Package 'onlinePCA'

July 22, 2025

Type Package

Title Online Principal Component Analysis

Version 1.3.2

Date 2023-11-15

Author David Degras [aut, cre], Herve Cardot [ctb]

Maintainer David Degras <ddegrasv@gmail.com>

Description Online PCA for multivariate and functional data using perturbation methods, low-rank incremental methods, and stochastic optimization methods.

License GPL-3

Depends R (>= 3.0.2), RSpectra

Imports Rcpp (>= 0.11.4), splines, stats

LinkingTo Rcpp, RcppArmadillo

URL https://cran.r-project.org/package=onlinePCA

Repository CRAN

NeedsCompilation yes

Date/Publication 2023-11-15 04:50:02 UTC

Contents

`onlinePCA-package` *Online Principal Component Analysis*

Description

Online PCA algorithms using perturbation methods (`perturbationRpca`), secular equations (`secularRpca`), incremental PCA (`incRpca`, `incRpca.block`, `incRpca.rc`), and stochastic optimization (`bsoipca`, `ccipca`, `ghapca`, `sgapca`, `snlpca`). `impute` handles missing data with the regression approach of Brand (2002). `batchpca` performs fast batch (offline) PCA using iterative methods. `create.basis`, `coef2fd`, `fd2coef` respectively create B-spline basis sets for functional data (FD), convert FD to basis coefficients, and convert basis coefficients back to FD. `updateMean` and `updateCovariance` update the sample mean and sample covariance.

Author(s)

David Degras <ddegrasv@gmail.com>

References

Brand, M. (2002). Incremental singular value decomposition of uncertain data with missing values. *European Conference on Computer Vision (ECCV).*

Gu, M. and Eisenstat, S. C. (1994). A stable and efficient algorithm for the rank-one modification of the symmetric eigenproblem. *SIAM Journal of Matrix Analysis and Applications.*

Hegde et al. (2006) Perturbation-Based Eigenvector Updates for On-Line Principal Components Analysis and Canonical Correlation Analysis. *Journal of VLSI Signal Processing*.

Oja (1992). Principal components, Minor components, and linear neural networks. *Neural Networks.*

Sanger (1989). Optimal unsupervised learning in a single-layer linear feedforward neural network. *Neural Networks.*

Mitliagkas et al. (2013). Memory limited, streaming PCA. *Advances in Neural Information Processing Systems*.

Weng et al. (2003). Candid Covariance-free Incremental Principal Component Analysis. *IEEE Trans. Pattern Analysis and Machine Intelligence*.

batchpca *Batch PCA*

Description

This function performs the PCA of a data matrix or covariance matrix, returning the specified number of principal components (eigenvectors) and eigenvalues.

Usage

```
batchpca(x, q, center, type = c("data","covariance"), byrow = FALSE)
```

Arguments

x	data or covariance matrix
q	number of requested PCs
center	optional centering vector for x
type	type of the matrix x
byrow	Are observation vectors stored in rows (TRUE) or in columns (FALSE)?

Details

The PCA is efficiently computed using the functions svds or eigs_sym of package RSpectra, depending on the argument type. An Implicitly Restarted Arnoldi Method (IRAM) is used in the former case and an Implicitly Restarted Lanczos Method (IRLM) in the latter.
The arguments center and byrow are only in effect if type is "data". In this case a scaling factor $1/\sqrt{n}$ (not $1/\sqrt{n-1}$) is applied to x before computing its singular values and vectors, where n is the number of observation vectors stored in x.

Value

A list with components

values	the first q squared singular values of x if type="data"; the first Q eigenvalues if type="covariance".
vectors	the first q PC of x.

References

https://www.arpack.org

Examples

```
## Not run:
## Simulate data
n <- 1e4
d <- 500
q <- 10
x <- matrix(runif(n*d), n, d)
x <- x %*% diag(sqrt(12*(1:d)))
# The eigenvalues of cov(x) are approximately 1, 2, ..., d
# and the corresponding eigenvectors are approximately
# the canonical basis of R^p

## PCA computation (from fastest to slowest)
system.time(pca1 <- batchpca(scale(x,scale=FALSE), q, byrow=TRUE))
system.time(pca2 <- batchpca(cov(x), q, type="covariance"))
system.time(pca3 <- eigen(cov(x),TRUE))
system.time(pca4 <- svd(scale(x/sqrt(n-1),scale=FALSE), 0, q))
system.time(pca5 <- prcomp(x))

## End(Not run)
```

---

bsoipca *Block Stochastic Orthononal Iteration (BSOI)*

Description

The online PCA algorithm of Mitliagkas et al. (2013) is a block-wise stochastic variant of the classical power-method.

Usage

```
bsoipca(x, q, U, B, center, byrow = FALSE)
```

Arguments

x	data matrix.
q	number of PC to compute.
U	matrix of initial PCs in columns (optional).
B	size of block updates (optional).
center	centering vector (optional).
byrow	are the data vectors in x stored in rows (TRUE) or columns (FALSE)?

Details

The default value of B is $floor(n/nblock)$ with n the number of data vectors in x, d the number of variables, and $nblock = ceiling(log(d))$ the number of blocks.
If U is specified, q defaults to ncol(U); otherwise the initial PCs are computed from the first block of data and q must be specified explicitly.
Although the algorithm does not give eigenvalues, they can easily be estimated by computing the variance of the data along the PCs.

Value

A matrix with the q first eigenvectors/PCs in columns.

References

Mitliagkas et al. (2013). Memory limited, streaming PCA. *Advances in Neural Information Processing Systems*.

Examples

```
## Simulate Brownian Motion
n <- 100 # number of sample paths
d <- 50 # number of observation points
x <- matrix(rnorm(n*d,sd=1/sqrt(d)),n,d)
x <- t(apply(x,1,cumsum)) # dim(x) = c(100,50)

q <- 10 # number of PC to compute
B <- 20 # block size

## BSOI PCA
U <- bsoipca(x, q, B=B, byrow=TRUE) # PCs
lambda <- apply(x %*% U, 2, var) # eigenvalues
```

---

ccipca *Candid Covariance-Free Incremental PCA*

Description

Stochastic gradient ascent algorithm CCIPCA of Weng et al. (2003).

Usage

```
ccipca(lambda, U, x, n, q = length(lambda), l=2, center, tol = 1e-8, sort = TRUE)
```

Arguments

lambda	vector of eigenvalues.
U	matrix of eigenvectors (PC) stored in columns.
x	new data vector.
n	sample size before observing x.
q	number of eigenvectors to compute.
l	'amnesic' parameter.
center	optional centering vector for x.
tol	numerical tolerance.
sort	Should the new eigenpairs be sorted?

Details

The 'amnesic' parameter l determines the weight of past observations in the PCA update. If l=0, all observations have equal weight, which is appropriate for stationary processes. Otherwise, typical values of l range between 2 and 4. As l increases, more weight is placed on new observations and less on older ones. For meaningful results, the condition 0<=l<n should hold.
The CCIPCA algorithm iteratively updates the PCs while deflating x. If at some point the Euclidean norm of x becomes less than tol, the algorithm stops to prevent numerical overflow.
If sort is TRUE, the updated eigenpairs are sorted by decreasing eigenvalue. If FALSE, they are not sorted.

Value

A list with components

values	updated eigenvalues.
vectors	updated eigenvectors.

References

Weng et al. (2003). Candid Covariance-free Incremental Principal Component Analysis. *IEEE Trans. Pattern Analysis and Machine Intelligence*.

Examples

```
## Simulation of Brownian motion
n <- 100 # number of paths
d <- 50  # number of observation points
q <- 10  # number of PCs to compute
x <- matrix(rnorm(n*d,sd=1/sqrt(d)), n, d)
x <- t(apply(x,1,cumsum))

## Initial PCA
n0 <- 50
pca <- princomp(x[1:n0,])
xbar <- pca$center
```

```
pca <- list(values=pca$sdev^2, vectors=pca$loadings)

## Incremental PCA
for (i in n0:(n-1))
{ xbar <- updateMean(xbar, x[i+1,], i)
   pca <- ccipca(pca$values, pca$vectors, x[i+1,], i, q = q, center = xbar) }

# Uncentered PCA
nx1 <- sqrt(sum(x[1,]^2))
pca <- list(values=nx1^2, vectors=as.matrix(x[1,]/nx1))
for (i in n0:(n-1))
   pca <- ccipca(pca$values, pca$vectors, x[i+1,], i, q = q)
```

`coef2fd` *Recover functional data from their B-spline coefficients*

Description

This function computes functional data from their coefficients in a B-spline basis.

Usage

```
coef2fd(beta, basis, byrow = TRUE)
```

Arguments

`beta`	B-spline coefficients
`basis`	object created by `create.basis`
`byrow`	are the coefficients of each functional observation stored in rows (TRUE) or in columns (FALSE)?

Value

A matrix of functional data stored in the same format (row or columns) as the coefficients `beta`.

Note

In view of (online or offline) functional PCA, the coefficients `beta` are left- or right- multiplied by `M^{-1/2}` (depending on their row/column format)) before applying the B-spline matrix `B`, with `M` the Gram matrix associated to `B`.

See Also

`create.basis`, `fd2coef`

Examples

```
n <- 100 # number of curves
d <- 500 # number of observation points
grid <- (1:d)/d # observation points
p <- 50 # number of B-spline basis functions

# Simulate Brownian motion
x <- matrix(rnorm(n*d,sd=1/sqrt(d)),n,d)
x <- t(apply(x,1,cumsum))

# Create B-spline basis
mybasis <- create.basis(grid, p, 1e-4)

# Compute smooth basis coefficients
beta <- fd2coef(x, mybasis)

# Recover smooth functional data
x.smooth <- coef2fd(beta, mybasis)

# Standard PCA and Functional PCA
pca <- prcomp(x)
fpca <- prcomp(beta)
```

---

create.basis *Create a smooth B-spline basis*

Description

This function creates a smooth B-spline basis and provides tools to find the coefficients of functional data in the basis and to recover functional data from basis coefficients.

Usage

```
create.basis(x, p, sp = 1e-09, degree = 3, nderiv = 2)
```

Arguments

x	vector of observation times
p	number of basis functions
sp	smoothing parameter
degree	degree of the B splines
nderiv	order of the derivative to penalize for smoothing

Details

The knots of the B-spline basis are taken as regular quantiles of x. The function output is intended for use with functions coef2fd and fd2coef.

Value

A list with fields

B	matrix of B-splines evaluated at x (each column represents a basis function)
S	matrix that maps functional data to their (smoothed) coefficients of their projection in the basis set. For the purpose of PCA, the coefficients are premultiplied by M^{1/2}, where M is the Gram matrix associated with B
invsqrtM	matrix M^{-1/2} used to recover functional # data from their coefficients in the basis set

See Also

coef2fd, fd2coef

Examples

```
n <- 100 # number of curves
d <- 500 # number of observation points
grid <- (1:d)/d # observation points
p <- 50 # number of B-spline basis functions

# Simulate Brownian motion
x <- matrix(rnorm(n*d,sd=1/sqrt(d)),n,d)
x <- t(apply(x,1,cumsum))

# Create B-spline basis
mybasis <- create.basis(grid, p, 1e-4)

# Compute smooth basis coefficients
beta <- fd2coef(x, mybasis)

# Recover smooth functional data
x.smooth <- coef2fd(beta, mybasis)

# Standard PCA and Functional PCA
pca <- prcomp(x)
fpca <- prcomp(beta)
```

---

fd2coef *Compute the coefficients of functional data in a B-spline basis*

Description

This function computes the coefficients of functional data in a B-spline basis.

Usage

```
fd2coef(x, basis, byrow = TRUE)
```

Arguments

x	matrix of functional data.
basis	object created by create.basis
.	
byrow	are the functional data stored in rows (TRUE) or in columns (FALSE)?

Value

A matrix of B-spline coefficients stored in the same format (row or columns) as functional data.

Note

In view of (online or offline) functional PCA, the coefficients are smoothed and premultiplied by M^{1/2}, with M the Gram matrix associated to the B-spline matrix.

See Also

create.basis, coef2fd

Examples

```
n <- 100 # number of curves
d <- 500 # number of observation points
grid <- (1:d)/d # observation points
p <- 50 # number of B-spline basis functions

# Simulate Brownian motion
x <- matrix(rnorm(n*d,sd=1/sqrt(d)),n,d)
x <- t(apply(x,1,cumsum))

# Create B-spline basis
mybasis <- create.basis(grid, p, 1e-4)

# Compute smooth basis coefficients
beta <- fd2coef(x, mybasis)

# Recover smooth functional data
x.smooth <- coef2fd(beta, mybasis)

# Standard PCA and Functional PCA
pca <- prcomp(x)
fpca <- prcomp(beta)
```

ghapca *Generalized Hebbian Algorithm for PCA*

Description

Online PCA with the GHA of Sanger (1989).

Usage

```
ghapca(lambda, U, x, gamma, q = length(lambda), center, sort = TRUE)
```

Arguments

lambda	optional vector of eigenvalues.
U	matrix of eigenvectors (PC) stored in columns.
x	new data vector.
gamma	vector of gain parameters.
q	number of eigenvectors to compute.
center	optional centering vector for x.
sort	Should the new eigenpairs be sorted?

Details

The vector gamma determines the weight placed on the new data in updating each eigenvector (the first coefficient of gamma corresponds to the first eigenvector, etc). It can be specified as a single positive number or as a vector of length ncol(U). Larger values of gamma place more weight on x and less on U. A common choice for (the components of) gamma is of the form c/n, with n the sample size and c a suitable positive constant.
If sort is TRUE and lambda is not missing, the updated eigenpairs are sorted by decreasing eigenvalue. Otherwise, they are not sorted.

Value

A list with components

values	updated eigenvalues or NULL.
vectors	updated eigenvectors.

References

Sanger (1989). Optimal unsupervised learning in a single-layer linear feedforward neural network. *Neural Networks.*

See Also

sgapca, snlpca

Examples

```
## Initialization
n <- 1e4  # sample size
n0 <- 5e3 # initial sample size
d <- 10   # number of variables
q <- d # number of PC
x <- matrix(runif(n*d), n, d)
x <- x %*% diag(sqrt(12*(1:d)))
# The eigenvalues of X are close to 1, 2, ..., d
# and the corresponding eigenvectors are close to
# the canonical basis of R^d

## GHA PCA
pca <- princomp(x[1:n0,])
xbar <- pca$center
pca <- list(values=pca$sdev[1:q]^2, vectors=pca$loadings[,1:q])
for (i in (n0+1):n) {
  xbar <- updateMean(xbar, x[i,], i-1)
  pca <- ghapca(pca$values, pca$vectors, x[i,], 2/i, q, xbar)
}
```

---

impute *BLUP Imputation of Missing Values*

Description

Missing values of a vector are imputed by best linear unbiased prediction (BLUP) assuming a multivariate normal distribution.

Usage

```
impute(lambda, U, x, center, tol = 1e-07)
```

Arguments

`lambda`	vector of eigenvalues of length `q`.
`U`	matrix of eigenvectors (principal components) of dimension `p * q`.
`x`	vector of observations of length `p` with missing entries.
`center`	centering vector for `x`. Default is zero.
`tol`	tolerance in the calculation of the pseudoinverse.

Details

The vector `x` is assumed to arise from a multivariate normal distribution with mean vector `center` and covariance matrix $Udiag(lambda)U^T$.

Value

The imputed vector x.

References

Brand, M. (2002). Incremental singular value decomposition of uncertain data with missing values. *European Conference on Computer Vision (ECCV).*

Examples

```
set.seed(10)
lambda <- c(1,2,5)
U <- qr.Q(qr(matrix(rnorm(30),10,3)))
x <- U %*% diag(sqrt(lambda)) %*% rnorm(3) + rnorm(10, sd =.05)
x.na <- x
x.na[c(1,3,7)] <- NA
x.imputed <- impute(lambda,U,x.na)
cbind(x,x.imputed)
```

---

incRpca *Incremental PCA*

Description

Online PCA using the incremental SVD method of Brand (2002) and Arora et al. (2012).

Usage

```
incRpca(lambda, U, x, n, f = 1/n, q = length(lambda), center, tol = 1e-7)
```

Arguments

lambda	vector of eigenvalues.
U	matrix of eigenvectors (principal components) stored in columns.
x	new data vector.
n	sample size before observing x.
f	forgetting factor: a number in (0,1).
q	number of eigenvectors to compute.
center	optional centering vector for x.
tol	numerical tolerance.

Details

If the Euclidean distance between `x` and `U` is more than `tol`, the number of eigenpairs increases to `length(lambda)+1` before eventual truncation at order `q`. Otherwise, the eigenvectors remain unchanged and only the eigenvalues are updated.
The forgetting factor `f` can be interpreted as the inverse of the number of observation vectors effectively used in the PCA: the "memory" of the PCA algorithm goes back `1/f` observations in the past. For larger values of `f`, the PCA update gives more relative weight to the new data `x` and less to the current PCA (`lambda,U`). For nonstationary processes, `f` should be closer to 1.
Only one of the arguments `n` and `f` needs being specified. If it is `n`, then `f` is set to `1/n` by default (usual PCA of sample covariance matrix where all data points have equal weight). If `f` is specified, its value overrides any eventual specification of `n`.

Value

A list with components

`values`	updated eigenvalues in decreasing order.
`vectors`	updated eigenvectors.

References

Arora et al. (2012). Stochastic Optimization for PCA and PLS. *50th Annual Conference on Communication, Control, and Computing (Allerton).*
Brand, M. (2002). Incremental singular value decomposition of uncertain data with missing values. *European Conference on Computer Vision (ECCV).*

Examples

```
## Simulate Brownian motion
n <- 100 # number of sample paths
d <- 50  # number of observation points
q <- 10  # number of PCs to compute
n0 <- 50 # number of sample paths used for initialization
x <- matrix(rnorm(n*d,sd=1/sqrt(d)), n, d)
x <- t(apply(x,1,cumsum))
dim(x) # (100,50)


## Incremental PCA (IPCA, centered)
pca <- prcomp(x[1:n0,]) # initialization
xbar <- pca$center
pca <- list(values=pca$sdev[1:q]^2, vectors=pca$rotation[,1:q])
for (i in (n0+1):n)
{
  xbar <- updateMean(xbar, x[i,], i-1)
  pca <- incRpca(pca$values, pca$vectors, x[i,], i-1, q = q,
center = xbar)
}

## Incremental PCA (IPCA, uncentered)
pca <- prcomp(x[1:n0,],center=FALSE) # initialization
```

```
pca <- list(values = pca$sdev[1:q]^2, vectors = pca$rotation[,1:q])
for (i in (n0+1):n)
  pca <- incRpca(pca$values, pca$vectors, x[i,], i-1, q = q)
```

`incRpca.block` *Incremental PCA with Block Update*

Description

Sequential Karhunen-Loeve (SKL) algorithm of Levy and Lindenbaum (2000). The PCA can be updated with respect to a data matrix (not just a data vector).

Usage

```
incRpca.block(x, B, lambda, U, n0 = 0, f, q = length(lambda), center, byrow = FALSE)
```

Arguments

`x`	data matrix
`B`	block size
`lambda`	initial eigenvalues (optional)
`U`	initial eigenvectors/PCs (optional)
`n0`	initial sample size (optional)
`f`	vector of forgetting factors
`q`	number of requested PCs
`center`	centering vector for `x` (optional)
`byrow`	Are the data vectors in `x` stored in rows (TRUE) or columns (FALSE)?

Details

This incremental PCA algorithm utilizes QR factorization and RSVD. It generalizes the algorithm `incRpca` from vector to matrix/block updates. However, `incRpca` should be preferred for vector updates as it is faster.

If `lambda` and `U` are specified, they are taken as the initial PCA. Otherwise, the PCA is initialized by the SVD of the first block of data in `x` (`B` data vectors). The number `n0` is the sample size before observing `x` (default value = 0). The number `B` is the size of blocks to be used in the PCA updates. Ideally, `B` should be a divisor of the number of data vectors in `x` (otherwise the last smaller block is discarded). If `U` is provided, then `B` and `q` default to the number of columns (=PC) of `U`.

The argument `f` determines the relative weight of current PCA and new data in each block update. Its length should be equal to the number of blocks in `x`, say $nblock$. If `n0` and `B` are provided, then `f` defaults to `B/(n0+(0:(nblock-1)*B))`, i.e., the case where all data points have equal weights. The values in `f` should be in (0,1), with higher values giving more weight to new data and less to the current PCA.

Value

A list with components

values	first q eigenvalues.
vectors	first q eigenvectors/PCs.

References

Levy, A. and Lindenbaum, M. (2000). Sequential Karhunen-Loeve basis extraction and its application to images. *IEEE Transactions on Image Processing*.

See Also

incRpca, incRpca.rc

Examples

```
## Simulate Brownian Motion
n <- 100 # number of sample paths
d <- 50 # number of observation points
x <- matrix(rnorm(n*d,sd=1/sqrt(d)),n,d)
x <- t(apply(x,1,cumsum)) # dim(x) = c(100,50)
q <- 10 # number of PC to compute
B <- 20 # block size
n0 <- B # initial sample size (if relevant)

## PCA without initial values
res1 <- incRpca.block(t(x), B, q=q) # data vectors in columns
res2 <- incRpca.block(x, B, q=q, byrow=TRUE) # data vectors in rows
all.equal(res1,res2) # TRUE

## PCA with initial values
svd0 <- svd(x[1:n0,], 0, n0) # use first block for initialization
lambda <- svd0$d[1:n0]^2/n0 # initial eigenvalues
U <- svd0$v # initial PC
res3 <- incRpca.block(x[-(1:n0),], B, lambda, U, n0, q=q, byrow=TRUE)
# run PCA with this initialization on rest of the data
all.equal(res1,res3) # compare with previous PCA: TRUE

## Compare with function incRpca
res4 <- list(values=lambda, vectors=U)
for (i in (n0+1):n)
res4 <- incRpca(res4$values, res4$vectors, x[i,], i-1, q=q)
B <- 1 # vector update
res5 <- incRpca.block(x[-(1:n0),], B, lambda, U, n0, q=q, byrow=TRUE)
ind <- which(sign(res5$vectors[1,]) != sign(res4$vectors[1,]))
res5$vectors[,ind] <- - res5$vectors[,ind] # align PCs (flip orientation as needed)
all.equal(res4,res5) # TRUE
```

incRpca.rc *Incremental PCA With Reduced Complexity*

Description

The incremental PCA is computed without rotating the updated projection space (Brand, 2002; Arora et al., 2012). Specifically, PCs are specified through a matrix of orthogonal vectors `Ut` that spans the PC space and a rotation matrix `Us` such that the PC matrix is `UtUs`. Given a new data vector, the PCA is updated by adding one column to `Ut` and recalculating the low-dimensional rotation matrix `Us`. This reduces complexity and helps preserving orthogonality. Eigenvalues are updated as the usual incremental PCA algorithm.

Usage

```
incRpca.rc(lambda, Ut, Us, x, n, f = 1/n, center, tol = 1e-07)
```

Arguments

`lambda`	vector of eigenvalues.
`Ut`	matrix of orthogonal vectors stored in columns.
`Us`	rotation matrix.
`x`	new data vector.
`n`	sample size before observing `x`.
`f`	forgetting factor: a number in (0,1).
`center`	optional centering vector for `x`.
`tol`	numerical tolerance for eigenvalues.

Details

For large datasets, this algorithm is considerably faster than its counterpart `incRpca`, reducing the time complexity of each update from $O(qd^2)$ to $O(qd + q^3)$ flops with `d` the length of `x`. A consequence of not rotating the PC basis at each update is that the dimension of the PCA decomposition increases whenever a new observation vector is not entirely contained in the PC space. To keep the number of PCs and eigenvalues from getting too large, it is necessary to multiply the matrices U_t and U_s at regular time intervals so as to recover the individual PCs and retain only the largest ones.

Value

A list with components

`values`	updated eigenvalues in decreasing order.
`Ut`	updated projection space.
`Us`	updated rotation matrix.

References

Arora et al. (2012). Stochastic Optimization for PCA and PLS. *50th Annual Conference on Communication, Control, and Computing (Allerton).*
Brand, M. (2002). Incremental singular value decomposition of uncertain data with missing values. *European Conference on Computer Vision (ECCV).*

See Also

incRpca, incRpca.block

Examples

```
## Not run:
# Data generation
n <- 400 # number of units
d <- 10000 # number of variables
n0 <- 200 # initial sample
q <- 20 # required number of PCs
x <- matrix(rnorm(n*d,sd=1/sqrt(d)),n,d) # data matrix
x <- t(apply(x,1,cumsum)) # standard Brownian motion

# Initial PCA
# Initial PCA
xbar0 <- colMeans(x[1:n0,])
pca0 <- batchpca(x0c, q, center=xbar0, byrow=TRUE)

# Incremental PCA with rotation
xbar <- xbar0
pca1 <- pca0
system.time({
for (i in n0:(n-1)) {
xbar <- updateMean(xbar, x[i+1,], i)
pca1 <- incRpca(pca1$values, pca1$vectors, x[i+1,], i, center=xbar)
}
})

# Incremental PCA without rotation
xbar <- xbar0
pca2 <- list(values=pca0$values, Ut=pca0$vectors, Us=diag(q))

system.time({
for (i in n0:(n-1)) {
xbar <- updateMean(xbar, x[i+1,], i)
pca2 <- incRpca.rc(pca2$values, pca2$Ut, pca2$Us, x[i+1,],
i, center = xbar)
# Rotate the PC basis and reduce its size to q every k observations
if (i %% q == 0 || i == n-1)
{ pca2$values <- pca2$values[1:q]
  pca2$Ut <- pca2$Ut %*% pca2$Us[,1:q]
  pca2$Us <- diag(q)
}
}
```

```
})

# Check that the results are identical
# Relative differences in eigenvalues
range(pca1$values/pca2$values-1)
# Cosines of angles between eigenvectors
abs(colSums(pca1$vectors * pca2$Ut))

## End(Not run)
```

`perturbationRpca` *Recursive PCA using a rank 1 perturbation method*

Description

This function recursively updates the PCA with respect to a single new data vector, using the (fast) perturbation method of Hegde et al. (2006).

Usage

```
perturbationRpca(lambda, U, x, n, f = 1/n, center, sort = TRUE)
```

Arguments

`lambda`	vector of eigenvalues.
`U`	matrix of eigenvectors (PC) stored in columns.
`x`	new data vector.
`n`	sample size before observing `x`.
`f`	forgetting factor: a number between 0 and 1.
`center`	optional centering vector for `x`.
`sort`	Should the eigenpairs be sorted?

Details

The forgetting factor `f` can be interpreted as the inverse of the number of observation vectors effectively used in the PCA: the "memory" of the PCA algorithm goes back `1/f` observations in the past. For larger values of `f`, the PCA update gives more relative weight to the new data `x` and less to the current PCA (`lambda,U`). For nonstationary processes, `f` should be closer to 1.
Only one of the arguments `n` and `f` needs being specified. If it is `n`, then `f` is set to `1/n` by default (usual PCA of sample covariance matrix where all data points have equal weight). If `f` is specified, its value overrides any eventual specification of `n`.
If `sort` is TRUE, the updated eigenpairs are sorted by decreasing eigenvalue. Otherwise, they are not sorted.

Value

A list with components

values	updated eigenvalues.
vectors	updated eigenvectors.

Note

This perturbation method is based on large sample approximations. It tends to be highly inaccurate for small/medium sized samples and should not be used in this case.

References

Hegde et al. (2006) Perturbation-Based Eigenvector Updates for On-Line Principal Components Analysis and Canonical Correlation Analysis. *Journal of VLSI Signal Processing*.

See Also

secularRpca

Examples

```
n <- 1e3
n0 <- 5e2
d <- 10
x <- matrix(runif(n*d), n, d)
 x <- x %*% diag(sqrt(12*(1:d)))
# The eigenvalues of cov(x) are approximately equal to 1, 2, ..., d
# and the corresponding eigenvectors are approximately equal to
# the canonical basis of R^d

## Perturbation-based recursive PCA
# Initialization: use factor 1/n0 (princomp) rather
# than factor 1/(n0-1) (prcomp) in calculations
pca <- princomp(x[1:n0,], center=FALSE)
xbar <- pca$center
pca <- list(values=pca$sdev^2, vectors=pca$loadings)

for (i in (n0+1):n) {
xbar <- updateMean(xbar, x[i,], i-1)
pca <- perturbationRpca(pca$values, pca$vectors, x[i,],
i-1, center=xbar) }
```

secularRpca *Recursive PCA Using Secular Equations*

Description

The PCA is recursively updated after observation of a new vector (rank one modification of the covariance matrix). Eigenvalues are computed as roots of a secular equation. Eigenvectors (principal components) are deduced by explicit calculation (Bunch et al., 1978) or approximated with the method of Gu and Eisenstat (1994).

Usage

```
secularRpca(lambda, U, x, n, f = 1/n, center, tol = 1e-10, reortho = FALSE)
```

Arguments

lambda	vector of eigenvalues.
U	matrix of eigenvectors (PCs) stored in columns.
x	new data vector.
n	sample size before observing x.
f	forgetting factor: a number in (0,1).
center	centering vector for x (optional).
tol	tolerance for the computation of eigenvalues.
reortho	if FALSE, eigenvectors are explicitly computed using the method of Bunch et al. (1978). If TRUE, they are approximated with the method of Gu and Eisenstat (1994).

Details

The method of secular equations provides accurate eigenvalues in all but pathological cases. On the other hand, the perturbation method implemented by perturbationRpca typically runs much faster but is only accurate for a large sample size n.

The default eigendecomposition method is that of Bunch et al. (1978). This algorithm consists in three stages: initial deflation, nonlinear solution of secular equations, and calculation of eigenvectors. The calculation of eigenvectors (PCs) is accurate for the first few eigenvectors but loss of accuracy and orthogonality may occur for the next ones. In contrast the method of Gu and Eisenstat (1994) is robust against small errors in the computation of eigenvalues. It provides eigenvectors that may be less accurate than the default method but for which strict orthogonality is guaranteed.

The forgetting factor f can be interpreted as the inverse of the number of observation vectors effectively used in the PCA: the "memory" of the PCA algorithm goes back 1/f observations in the past. For larger values of f, the PCA update gives more relative weight to the new data x and less to the current PCA (lambda,U). For nonstationary processes, f should be closer to 1.

Only one of the arguments n and f needs being specified. If it is n, then f is set to 1/n by default (usual PCA of sample covariance matrix where all data points have equal weight). If f is specified, its value overrides any eventual specification of n.

Value

A list with components

values	updated eigenvalues in decreasing order.
vectors	updated eigenvectors (PCs).

References

Bunch, J.R., Nielsen, C.P., and Sorensen, D.C. (1978). Rank-one modification of the symmetric eigenproblem. *Numerische Mathematik.*
Gu, M. and Eisenstat, S.C. (1994). A stable and efficient algorithm for the rank-one modification of the symmetric eigenproblem. *SIAM Journal of Matrix Analysis and Applications.*

See Also

perturbationRpca

Examples

```
# Initial data set
n <- 100
d <- 50
x <- matrix(runif(n*d),n,d)
xbar <- colMeans(x)
pca0 <- eigen(cov(x))

# New observation
newx <- runif(d)

# Recursive PCA with secular equations
xbar <- updateMean(xbar, newx, n)
pca <- secularRpca(pca0$values, pca0$vectors, newx, n, center = xbar)
```

---

sgapca *Stochastic Gradient Ascent PCA*

Description

Online PCA with the SGA algorithm of Oja (1992).

Usage

```
sgapca(lambda, U, x, gamma, q = length(lambda), center,
type = c("exact", "nn"), sort = TRUE)
```

Arguments

lambda	optional vector of eigenvalues.
U	matrix of eigenvectors (PC) stored in columns.
x	new data vector.
gamma	vector of gain parameters.
q	number of eigenvectors to compute.
center	optional centering vector for x.
type	algorithm implementation: "exact" or "nn" (neural network).
sort	Should the new eigenpairs be sorted?

Details

The gain vector gamma determines the weight placed on the new data in updating each principal component. The first coefficient of gamma corresponds to the first principal component, etc.. It can be specified as a single positive number (which is recycled by the function) or as a vector of length ncol(U). For larger values of gamma, more weight is placed on x and less on U. A common choice for (the components of) gamma is of the form c/n, with n the sample size and c a suitable positive constant.
The Stochastic Gradient Ascent PCA can be implemented exactly or through a neural network. The latter is less accurate but faster.
If sort is TRUE and lambda is not missing, the updated eigenpairs are sorted by decreasing eigenvalue. Otherwise, they are not sorted.

Value

A list with components

values	updated eigenvalues or NULL.
vectors	updated principal components.

References

Oja (1992). Principal components, Minor components, and linear neural networks. *Neural Networks.*

See Also

ghapca, snlpca

Examples

```
## Initialization
n <- 1e4  # sample size
n0 <- 5e3 # initial sample size
d <- 10   # number of variables
q <- d # number of PC to compute
x <- matrix(runif(n*d), n, d)
```

```
x <- x %*% diag(sqrt(12*(1:d)))
# The eigenvalues of x are close to 1, 2, ..., d
# and the corresponding eigenvectors are close to
# the canonical basis of R^d

## SGA PCA
xbar <- colMeans(x[1:n0,])
pca <- batchpca(x[1:n0,], q, center=xbar, byrow=TRUE)
for (i in (n0+1):n) {
  xbar <- updateMean(xbar, x[i,], i-1)
  pca <- sgapca(pca$values, pca$vectors, x[i,], 2/i, q, xbar)
}
pca
```

---

snlpca *Subspace Network Learning PCA*

Description

Online PCA with the SNL algorithm of Oja (1992).

Usage

```
snlpca(lambda, U, x, gamma, q = length(lambda), center,
type = c("exact", "nn"), sort = TRUE)
```

Arguments

lambda	optional vector of eigenvalues.
U	matrix of eigenvectors (PC) stored in columns.
x	new data vector.
gamma	vector of learning rates.
q	number of eigenvectors to compute.
center	optional centering vector for x.
type	algorithm implementation: "exact" or "nn" (neural network).
sort	Should the new eigenpairs be sorted?

Details

The vector gamma determines the weight placed on the new data in updating each PC. For larger values of gamma, more weight is placed on x and less on U. A common choice is of the form c/n, with n the sample size and c a suitable positive constant. Argument gamma can be specified as a single positive number (common to all PCs) or as a vector of length q.
If sort is TRUE and lambda is not missing, the updated eigenpairs are sorted by decreasing eigenvalue. Otherwise, they are not sorted.

Value

A list with components

values	updated eigenvalues or NULL.
vectors	updated (rotated) eigenvectors.

Note

The Subspace Network Learning PCA can be implemented exactly or through a neural network. The latter is less accurate but much faster. Unlike the GHA and SGA algorithms, the SNL algorithm does not consistently estimate principal components. It provides only the linear space spanned by the PCs.

References

Oja (1992). Principal components, Minor components, and linear neural networks. *Neural Networks.*

See Also

ghapca, sgapca

Examples

```
## Initialization
n <- 1e4  # sample size
n0 <- 5e3 # initial sample size
d <- 10   # number of variables
q <- d # number of PC to compute
x <- matrix(runif(n*d), n, d)
x <- x %*% diag(sqrt(12*(1:d)))
# The eigenvalues of x are close to 1, 2, ..., d
# and the corresponding eigenvectors are close to
# the canonical basis of R^d

## SNL PCA
xbar <- colMeans(x[1:n0,])
pca <- batchpca(x[1:n0,], q, center=xbar, byrow=TRUE)
for (i in (n0+1):n) {
  xbar <- updateMean(xbar, x[i,], i-1)
  pca <- snlpca(pca$values, pca$vectors, x[i,], 1/i, q, xbar)
}
```

`updateCovariance` *Update the Sample Covariance Matrix*

Description

This function recursively updates a covariance matrix without entirely recomputing it when new observations arrive.

Usage

```
updateCovariance(C, x, n, xbar, f, byrow = TRUE)
```

Arguments

`C`	covariance matrix.
`x`	vector/matrix of new data.
`n`	sample size before observing `x`.
`xbar`	mean vector before observing `x`.
`f`	forgetting factor: a number beween 0 and 1.
`byrow`	Are the observation vectors in `x` stored in rows?

Details

The forgetting factor `f` determines the balance between past and present observations in the PCA update: the closer it is to 1 (resp. to 0), the more weight is placed on current (resp. past) observations. At least one of the arguments `n` and `f` must be specified. If `f` is specified, its value overrides the argument `n`. The default `f=1/n` corresponds to a stationnary observation process.
The argument `byrow` should be set to TRUE (default value) if the data vectors in `x` are stored in rows and to FALSE if they are stored in columns. The function automatically handles the case where `x` is a single vector.

Value

The updated covariance matrix.

See Also

`updateMean`

Examples

```
n <- 1e4
n0 <- 5e3
d <- 10
x <- matrix(runif(n*d), n, d)

## Direct computation of the covariance
```

```
C <- cov(x)

## Recursive computation of the covariance
xbar0 <- colMeans(x[1:n0,])
C0 <- cov(x[1:n0,])
Crec <- updateCovariance(C0, x[(n0+1):n,], n0, xbar0)

## Check equality
all.equal(C, Crec)
```

---

updateMean *Update the Sample Mean Vector*

Description

Recursive update of the sample mean vector.

Usage

```
updateMean(xbar, x, n, f, byrow = TRUE)
```

Arguments

xbar	current mean vector.
x	vector or matrix of new data.
n	sample size before observing x.
f	forgetting factor: a number in (0,1).
byrow	Are the observation vectors in x stored in rows (TRUE) or in columns (FALSE)?

Details

The forgetting factor f determines the balance between past and present observations in the PCA update: the closer it is to 1 (resp. to 0), the more weight is placed on current (resp. past) observations. At least one of the arguments n and f must be specified. If f is specified, its value overrides the argument n. For a given argument n, the default value off is$k/(n+k)$, with k the number of new vector observations. This corresponds to a stationnary observation process.

Value

The updated mean vector.

See Also

updateCovariance

Examples

```
n <- 1e4
n0 <- 5e3
d <- 10
x <- matrix(runif(n*d), n, d)

## Direct computation
xbar1 <- colMeans(x)

## Recursive computation
xbar2 <- colMeans(x[1:n0,])
xbar2 <- updateMean(xbar2, x[(n0+1):n,], n0)

## Check equality
all.equal(xbar1, xbar2)
```

Index