Package 'RcppML'

October 12, 2022

Type Package

Title Rcpp Machine Learning Library

Version 0.3.7

Date 2021-09-21

Description Fast machine learning algorithms including matrix factorization and divisive clustering for large sparse and dense matrices.

License GPL (>= 2)

Imports Rcpp, Matrix, methods, stats

LinkingTo Rcpp, RcppEigen

VignetteBuilder knitr

RoxygenNote 7.1.1

Suggests knitr, rmarkdown, testthat (>= 3.0.0)

Config/testthat/edition 3

URL https://github.com/zdebruine/RcppML

BugReports https://github.com/zdebruine/RcppML/issues

NeedsCompilation yes

Author Zachary DeBruine [aut, cre] (<https://orcid.org/0000-0003-2234-4827>)

Maintainer Zachary DeBruine <zacharydebruine@gmail.com>

Repository CRAN

Date/Publication 2021-09-21 19:00:02 UTC

R topics documented:

`bipartition` *Bipartition a sample set*

Description

Spectral biparitioning by rank-2 matrix factorization

Usage

```
bipartition(
  A,
  tol = 1e-05,
  maxit = 100,
  nonneg = TRUE,
  samples = 1:ncol(A),
  seed = NULL,
  verbose = FALSE,
  calc_dist = FALSE,
  diag = TRUE
)
```

Arguments

`A`	matrix of features-by-samples in dense or sparse format (preferred classes are "matrix" or "Matrix::dgCMatrix", respectively). Prefer sparse storage when more than half of all values are zero.
`tol`	stopping criteria, the correlation distance between w across consecutive iterations, $1 - cor(w_i, w_{i-1})$
`maxit`	stopping criteria, maximum number of alternating updates of w and h
`nonneg`	enforce non-negativity
`samples`	samples to include in bipartition, numbered from 1 to `ncol(A)`. Default is `NULL` for all samples.
`seed`	random seed for model initialization
`verbose`	print model tolerances between iterations
`calc_dist`	calculate the relative cosine distance of samples within a cluster to either cluster centroid. If `TRUE`, centers for clusters will also be calculated.
`diag`	scale factors in w and h to sum to 1 by introducing a diagonal, d. This should generally never be set to `FALSE`. Diagonalization enables symmetry of models in factorization of symmetric matrices, convex L1 regularization, and consistent factor scalings.

Details

Spectral bipartitioning is a popular subroutine in divisive clustering. The sign of the difference between sample loadings in factors of a rank-2 matrix factorization gives a bipartition that is nearly identical to an SVD.

Rank-2 matrix factorization by alternating least squares is faster than rank-2-truncated SVD (i.e. *irlba*).

This function is a specialization of rank-2 `nmf` with support for factorization of only a subset of samples, and with additional calculations on the factorization model relevant to bipartitioning. See `nmf` for details regarding rank-2 factorization.

Value

A list giving the bipartition and useful statistics:

- v : vector giving difference between sample loadings between factors in a rank-2 factorization
- dist : relative cosine distance of samples within a cluster to centroids of assigned vs. not-assigned cluster
- size1 : number of samples in first cluster (positive loadings in 'v')
- size2 : number of samples in second cluster (negative loadings in 'v')
- samples1: indices of samples in first cluster
- samples2: indices of samples in second cluster
- center1 : mean feature loadings across samples in first cluster
- center2 : mean feature loadings across samples in second cluster

Author(s)

Zach DeBruine

References

Kuang, D, Park, H. (2013). "Fast rank-2 nonnegative matrix factorization for hierarchical document clustering." Proc. 19th ACM SIGKDD intl. conf. on Knowledge discovery and data mining.

See Also

`nmf`, `dclust`

Examples

```
## Not run:
library(Matrix)
data(iris)
A <- as(as.matrix(iris[,1:4]), "dgCMatrix")
bipartition(A, calc_dist = TRUE)

## End(Not run)
```

`dclust` *Divisive clustering*

Description

Recursive bipartitioning by rank-2 matrix factorization with an efficient modularity-approximate stopping criteria

Usage

```
dclust(
  A,
  min_samples,
  min_dist = 0,
  verbose = TRUE,
  tol = 1e-05,
  maxit = 100,
  nonneg = TRUE,
  seed = NULL
)
```

Arguments

`A`	matrix of features-by-samples in sparse format (preferred class is "Matrix::dgCMatrix")
`min_samples`	stopping criteria giving the minimum number of samples permitted in a cluster
`min_dist`	stopping criteria giving the minimum cosine distance of samples within a cluster to the center of their assigned vs. unassigned cluster. If `0`, neither this distance nor cluster centroids will be calculated.
`verbose`	print number of divisions in each generation
`tol`	in rank-2 NMF, the correlation distance $(1 - R^2)$ between w across consecutive iterations at which to stop factorization
`maxit`	stopping criteria, maximum number of alternating updates of w and h
`nonneg`	in rank-2 NMF, enforce non-negativity
`seed`	random seed for rank-2 NMF model initialization

Details

Divisive clustering is a sensitive and fast method for sample classification. Samples are recursively partitioned into two groups until a stopping criteria is satisfied and prevents successful partitioning.

See `nmf` and `bipartition` for technical considerations and optimizations relevant to bipartitioning.

Stopping criteria. Two stopping criteria are used to prevent indefinite division of clusters and tune the clustering resolution to a desirable range:

- `min_samples`: Minimum number of samples permitted in a cluster

- `min_dist`: Minimum cosine distance of samples to their cluster center relative to their unassigned cluster center (an approximation of Newman-Girvan modularity)

Newman-Girvan modularity (Q) is an interpretable and widely used measure of modularity for a bipartition. However, it requires the calculation of distance between all within-cluster and between-cluster sample pairs. This is computationally intensive, especially for large sample sets.

`dclust` uses a measure which linearly approximates Newman-Girvan modularity, and simply requires the calculation of distance between all samples in a cluster and both cluster centers (the assigned and unassigned center), which is orders of magnitude faster to compute. Cosine distance is used instead of Euclidean distance since it handles outliers and sparsity well.

A bipartition is rejected if either of the two clusters contains fewer than `min_samples` or if the mean relative cosine distance of the bipartition is less than `min_dist`.

A bipartition will only be attempted if there are more than `2 * min_samples` samples in the cluster, meaning that `dist` may not be calculated for some clusters.

Reproducibility. Because rank-2 NMF is approximate and requires random initialization, results may vary slightly across restarts. Therefore, specify a `seed` to guarantee absolute reproducibility.

Other than setting the seed, reproducibility may be improved by setting `tol` to a smaller number to increase the exactness of each bipartition.

Value

A list of lists corresponding to individual clusters:

- id : character sequence of "0" and "1" giving position of clusters along splitting hierarchy
- samples : indices of samples in the cluster
- center : mean feature expression of all samples in the cluster
- dist : if applicable, relative cosine distance of samples in cluster to assigned/unassigned cluster center.
- leaf : is cluster a leaf node

Author(s)

Zach DeBruine

References

Schwartz, G. et al. "TooManyCells identifies and visualizes relationships of single-cell clades". Nature Methods (2020).

Newman, MEJ. "Modularity and community structure in networks". PNAS (2006)

Kuang, D, Park, H. (2013). "Fast rank-2 nonnegative matrix factorization for hierarchical document clustering." Proc. 19th ACM SIGKDD intl. conf. on Knowledge discovery and data mining.

See Also

`bipartition`, `nmf`

Examples

```
## Not run:
library(Matrix)
data(USArrests)
A <- as(as.matrix(t(USArrests)), "dgCMatrix")
clusters <- dclust(A, min_samples = 2, min_dist = 0.001)
str(clusters)

## End(Not run)
```

`getRcppMLthreads` *Get the number of threads RcppML should use*

Description

Get the number of threads that will be used by RcppML functions supporting parallelization with OpenMP. Use `setRcppMLthreads` to set the number of threads to be used.

Usage

```
getRcppMLthreads()
```

Value

integer giving number of threads to be used by RcppML functions. `0` corresponds to all available threads, as determined by OpenMP.

Author(s)

Zach DeBruine

See Also

`setRcppMLthreads`

Examples

```
## Not run:
# set serial configuration
setRcppMLthreads(1)
getRcppMLthreads()

# restore default parallel configuration,
# letting OpenMP decide how many threads to use
setRcppMLthreads(0)
getRcppMLthreads()

## End(Not run)
```

`mse` *Mean Squared Error loss of a factor model*

Description

MSE of factor models `w` and `h` given sparse matrix A

Usage

```
mse(A, w, d = NULL, h, mask_zeros = FALSE)
```

Arguments

`A`	matrix of features-by-samples in dense or sparse format (preferred classes are "matrix" or "Matrix::dgCMatrix", respectively). Prefer sparse storage when more than half of all values are zero.
`w`	dense matrix of class `matrix` with factors (columns) by features (rows)
`d`	diagonal scaling vector of rank length
`h`	dense matrix of class `matrix` with samples (columns) by factors (rows)
`mask_zeros`	handle zeros as missing values, available only when `A` is sparse

Details

Mean squared error of a matrix factorization of the form $A = wdh$ is given by

$$\frac{\sum_{i,j} (A - wdh)^2}{ij}$$

where i and j are the number of rows and columns in A.

Thus, this function simply calculates the cross-product of wh or wdh (if d is specified), subtracts that from A, squares the result, and calculates the mean of all values.

If no diagonal scaling vector is present in the model, input `d = rep(1, k)` where `k` is the rank of the model.

Parallelization. Calculation of mean squared error is performed in parallel across columns in `A` using the number of threads set by `setRcppMLthreads`. By default, all available threads are used, see `getRcppMLthreads`.

Value

mean squared error of the factorization model

Author(s)

Zach DeBruine

Examples

```
## Not run:
library(Matrix)
A <- Matrix::rsparsematrix(1000, 1000, 0.1)
model <- nmf(A, k = 10, tol = 0.01)
c_mse <- mse(A, model$w, model$d, model$h)
R_mse <- mean((A - model$w %*% Diagonal(x = model$d) %*% model$h)^2)
all.equal(c_mse, R_mse)

## End(Not run)
```

`nmf` *Non-negative matrix factorization*

Description

Sparse matrix factorization of the form $A = wdh$ by alternating least squares with optional non-negativity constraints.

Usage

```
nmf(
  A,
  k,
  tol = 1e-04,
  maxit = 100,
  verbose = TRUE,
  L1 = c(0, 0),
  seed = NULL,
  mask_zeros = FALSE,
  diag = TRUE,
  nonneg = TRUE
)
```

Arguments

`A`	matrix of features-by-samples in dense or sparse format (preferred classes are "matrix" or "Matrix::dgCMatrix", respectively). Prefer sparse storage when more than half of all values are zero.
`k`	rank
`tol`	stopping criteria, the correlation distance between w across consecutive iterations, $1 - cor(w_i, w_{i-1})$
`maxit`	stopping criteria, maximum number of alternating updates of w and h
`verbose`	print model tolerances between iterations
`L1`	L1/LASSO penalties between 0 and 1, array of length two for `c(w, h)`
`seed`	random seed for model initialization

mask_zeros	handle zeros as missing values, available only when A is sparse
diag	scale factors in w and h to sum to 1 by introducing a diagonal, d. This should generally never be set to FALSE. Diagonalization enables symmetry of models in factorization of symmetric matrices, convex L1 regularization, and consistent factor scalings.
nonneg	enforce non-negativity

Details

This fast non-negative matrix factorization (NMF) implementation decomposes a matrix A into lower-rank non-negative matrices w and h, with factors scaled to sum to 1 via multiplication by a diagonal, d:

$$A = wdh$$

The scaling diagonal enables symmetric factorization, convex L1 regularization, and consistent factor scalings regardless of random initialization.

The factorization model is randomly initialized, and w and h are updated alternately using least squares. Given A and w, h is updated according to the equation:

$$w^T wh = wA_j$$

This equation is in the form $ax = b$ where $a = w^T w$, $x = h$, and $b = wA_j$ for all columns j in A.

The corresponding update for w is

$$hh^T w = hA_j^T$$

Stopping criteria. Alternating least squares projections (see `project` subroutine) are repeated until a stopping criteria is satisfied, which is either a maximum number of iterations or a tolerance based on the correlation distance between models ($1 - cor(w_i, w_{i-1})$) across consecutive iterations. Use the `tol` parameter to control the stopping criteria for alternating updates:

- `tol` = `1e-2` is appropriate for approximate mean squared error determination and coarse cross-validation, useful for rank determination.
- `tol` = `1e-3` to `1e-4` are suitable for rapid expermentation, cross-validation, and preliminary analysis.
- `tol` = `1e-5` and smaller for publication-quality runs
- `tol` = `1e-10` and smaller for robust factorizations at or near machine-precision

Parallelization. Least squares projections in factorizations of rank-3 and greater are parallelized using the number of threads set by `setRcppMLthreads`. By default, all available threads are used, see `getRcppMLthreads`. The overhead of parallization is too great to benefit rank-1 and rank-2 factorization.

Specializations. There are specializations for symmetric matrices, and for rank-1 and rank-2 factorization.

L1 regularization. L1 penalization increases the sparsity of factors, but does not change the information content of the model or the relative contributions of the leading coefficients in each factor to the model. L1 regularization only slightly increases the error of a model. L1 penalization should be used to aid interpretability. Penalty values should range from 0 to 1, where 1 gives complete

sparsity. In this implementation of NMF, a scaling diagonal ensures that the L1 penalty is equally applied across all factors regardless of random initialization and the distribution of the model. Many other implementations of matrix factorization claim to apply L1, but the magnitude of the penalty is at the mercy of the random distribution and more significantly affects factors with lower overall contribution to the model. L1 regularization of rank-1 and rank-2 factorizations has no effect.

Rank-2 factorization. When $k = 2$, a very fast optimized algorithm is used. Two-variable least squares solutions to the problem $ax = b$ are found by direct substitution:

$$x_1 = \frac{a_{22}b_1 - a_{12}b_2}{a_{11}a_{22} - a_{12}^2}$$

$$x_2 = \frac{a_{11}b_2 - a_{12}b_1}{a_{11}a_{22} - a_{12}^2}$$

In the above equations, the denominator is constant and thus needs to be calculated only once. Additionally, if non-negativity constraints are to be imposed, if $x_1 < 0$ then $x_1 = 0$ and $x_2 = \frac{b_1}{a_{11}}$. Similarly, if $x_2 < 0$ then $x_2 = 0$ and $x_1 = \frac{b_2}{a_{22}}$.

Rank-2 NMF is useful for bipartitioning, and is a subroutine in `bipartition`, where the sign of the difference between sample loadings in both factors gives the partitioning.

Rank-1 factorization. Rank-1 factorization by alternating least squares gives vectors equivalent to the first singular vectors in an SVD. It is a normalization of the data to a middle point, and may be useful for ordering samples based on the most significant axis of variation (i.e. pseudotime trajectories). Diagonal scaling guarantees consistent linear scaling of the factor across random restarts.

Random seed and reproducibility. Results of a rank-1 and rank-2 factorizations should be reproducible regardless of random seed. For higher-rank models, results across random restarts should, in theory, be comparable at very high tolerances (i.e. machine precision for *double*, corresponding to about `tol = 1e-10`). However, to guarantee reproducibility without such low tolerances, set the `seed` argument. Note that `set.seed()` will not work. Only random initialization is supported, as other methods incur unnecessary overhead and sometimes trap updates into local minima.

Rank determination. This function does not attempt to provide a method for rank determination. Like any clustering algorithm or dimensional reduction, finding the optimal rank can be subjective. An easy way to estimate rank uses the "elbow method", where the inflection point on a plot of Mean Squared Error loss (MSE) vs. rank gives a good idea of the rank at which most of the signal has been captured in the model. Unfortunately, this inflection point is not often as obvious for NMF as it is for SVD or PCA.

k-fold cross-validation is a better method. Missing value of imputation has previously been proposed, but is arguably no less subjective than test-training splits and requires computationally slower factorization updates using missing values, which are not supported here.

Symmetric factorization. Special optimization for symmetric matrices is automatically applied. Specifically, alternating updates of `w` and `h` require transposition of `A`, but `A == t(A)` when `A` is symmetric, thus no up-front transposition is performed.

Zero-masking. When zeros in a data structure can be regarded as "missing", `mask_zeros = TRUE` may be set. However, this requires a slower algorithm, and tolerances will fluctuate more dramatically.

Publication reference. For theoretical and practical considerations, please see our manuscript: "DeBruine ZJ, Melcher K, Triche TJ (2021) High-performance non-negative matrix factorization for large single cell data." on BioRXiv.

Value

A list giving the factorization model:

- w : feature factor matrix
- d : scaling diagonal vector
- h : sample factor matrix
- tol : tolerance between models at final update
- iter : number of alternating updates run

Author(s)

Zach DeBruine

References

DeBruine, ZJ, Melcher, K, and Triche, TJ. (2021). "High-performance non-negative matrix factorization for large single-cell data." BioRXiv.

Lin, X, and Boutros, PC (2020). "Optimization and expansion of non-negative matrix factorization." BMC Bioinformatics.

Lee, D, and Seung, HS (1999). "Learning the parts of objects by non-negative matrix factorization." Nature.

Franc, VC, Hlavac, VC, Navara, M. (2005). "Sequential Coordinate-Wise Algorithm for the Non-negative Least Squares Problem". Proc. Int'l Conf. Computer Analysis of Images and Patterns. Lecture Notes in Computer Science.

See Also

nnls, project, mse

Examples

```
## Not run:
library(Matrix)
# basic NMF
model <- nmf(rsparsematrix(1000, 100, 0.1), k = 10)

# compare rank-2 NMF to second left vector in an SVD
data(iris)
A <- as(as.matrix(iris[,1:4]), "dgCMatrix")
nmf_model <- nmf(A, 2, tol = 1e-5)
bipartitioning_vector <- apply(nmf_model$w, 1, diff)
second_left_svd_vector <- base::svd(A, 2)$u[,2]
abs(cor(bipartitioning_vector, second_left_svd_vector))

# compare rank-1 NMF with first singular vector in an SVD
abs(cor(nmf(A, 1)$w[,1], base::svd(A, 2)$u[,1]))

# symmetric NMF
```

```
A <- crossprod(rsparsematrix(100, 100, 0.02))
model <- nmf(A, 10, tol = 1e-5, maxit = 1000)
plot(model$w, t(model$h))
# see package vignette for more examples

## End(Not run)
```

---

nnls *Non-negative least squares*

Description

Solves the equation a %*% x = b for x subject to $x > 0$.

Usage

```
nnls(a, b, cd_maxit = 100L, cd_tol = 1e-08, fast_nnls = FALSE, L1 = 0)
```

Arguments

a	symmetric positive definite matrix giving coefficients of the linear system
b	matrix giving the right-hand side(s) of the linear system
cd_maxit	maximum number of coordinate descent iterations
cd_tol	stopping criteria, difference in x across consecutive solutions over the sum of x
fast_nnls	initialize coordinate descent with a FAST NNLS approximation
L1	L1/LASSO penalty to be subtracted from b

Details

This is a very fast implementation of non-negative least squares (NNLS), suitable for very small or very large systems.

Algorithm. Sequential coordinate descent (CD) is at the core of this implementation, and requires an initialization of x. There are two supported methods for initialization of x:

1. **Zero-filled initialization** when fast_nnls = FALSE and cd_maxit > 0. This is generally very efficient for well-conditioned and small systems.
2. **Approximation with FAST** when fast_nnls = TRUE. Forward active set tuning (FAST), described below, finds an approximate active set using unconstrained least squares solutions found by Cholesky decomposition and substitution. To use only FAST approximation, set cd_maxit = 0.

a must be symmetric positive definite if FAST NNLS is used, but this is not checked.

See our BioRXiv manuscript (references) for benchmarking against Lawson-Hanson NNLS and for a more technical introduction to these methods.

Coordinate Descent NNLS. Least squares by **sequential coordinate descent** is used to ensure the solution returned is exact. This algorithm was introduced by Franc et al. (2005), and our

implementation is a vectorized and optimized rendition of that found in the NNLM R package by Xihui Lin (2020).

FAST NNLS. Forward active set tuning (FAST) is an exact or near-exact NNLS approximation initialized by an unconstrained least squares solution. Negative values in this unconstrained solution are set to zero (the "active set"), and all other values are added to a "feasible set". An unconstrained least squares solution is then solved for the "feasible set", any negative values in the resulting solution are set to zero, and the process is repeated until the feasible set solution is strictly positive.

The FAST algorithm has a definite convergence guarantee because the feasible set will either converge or become smaller with each iteration. The result is generally exact or nearly exact for small well-conditioned systems (< 50 variables) within 2 iterations and thus sets up coordinate descent for very rapid convergence. The FAST method is similar to the first phase of the so-called "TNT-NN" algorithm (Myre et al., 2017), but the latter half of that method relies heavily on heuristics to refine the approximate active set, which we avoid by using coordinate descent instead.

Value

vector or matrix giving solution for `x`

Author(s)

Zach DeBruine

References

DeBruine, ZJ, Melcher, K, and Triche, TJ. (2021). "High-performance non-negative matrix factorization for large single-cell data." BioRXiv.

Franc, VC, Hlavac, VC, and Navara, M. (2005). "Sequential Coordinate-Wise Algorithm for the Non-negative Least Squares Problem. Proc. Int'l Conf. Computer Analysis of Images and Patterns."

Lin, X, and Boutros, PC (2020). "Optimization and expansion of non-negative matrix factorization." BMC Bioinformatics.

Myre, JM, Frahm, E, Lilja DJ, and Saar, MO. (2017) "TNT-NN: A Fast Active Set Method for Solving Large Non-Negative Least Squares Problems". Proc. Computer Science.

See Also

`nmf`, `project`

Examples

```
## Not run:
# compare solution to base::solve for a random system
X <- matrix(runif(100), 10, 10)
a <- crossprod(X)
b <- crossprod(X, runif(10))
unconstrained_soln <- solve(a, b)
nonneg_soln <- nnls(a, b)
unconstrained_err <- mean((a %*% unconstrained_soln - b)^2)
nonnegative_err <- mean((a %*% nonneg_soln - b)^2)
unconstrained_err
```

```
nonnegative_err
all.equal(solve(a, b), nnls(a, b))

# example adapted from multiway::fnnls example 1
X <- matrix(1:100,50,2)
y <- matrix(101:150,50,1)
beta <- solve(crossprod(X)) %*% crossprod(X, y)
beta
beta <- nnls(crossprod(X), crossprod(X, y))
beta

## End(Not run)
```

`project` *Project a linear factor model*

Description

Solves the equation $A = wh$ for either h or w given either w or h and A

Usage

```
project(A, w = NULL, h = NULL, nonneg = TRUE, L1 = 0, mask_zeros = FALSE)
```

Arguments

`A`	matrix of features-by-samples in dense or sparse format (preferred classes are "matrix" or "Matrix::dgCMatrix", respectively). Prefer sparse storage when more than half of all values are zero.
`w`	dense matrix of factors x features giving the linear model to be projected (if `h = NULL`)
`h`	dense matrix of factors x samples giving the linear model to be projected (if `w = NULL`)
`nonneg`	enforce non-negativity
`L1`	L1/LASSO penalty to be applied. No scaling is performed. See details.
`mask_zeros`	handle zeros as missing values, available only when `A` is sparse

Details

For the classical alternating least squares matrix factorization update problem $A = wh$, the updates (or projection) of h is given by the equation:

$$w^T wh = wA_j$$

which is in the form $ax = b$ where $a = w^T w$ $x = h$ and $b = wA_j$ for all columns j in A.

Given A, `project` can solve for either w or h given the other:

- When given A and w, h is found using a highly efficient parallelization scheme.
- When given A and h, w is found without transposition of A (as would be the case in traditional block-pivoting matrix factorization) by accumulating the right-hand sides of linear systems in-place in A, then solving the systems. Note that w may also be found by inputting the transpose of A and h in place of w, (i.e. `A = t(A), w = h, h = NULL`). However, for most applications, the cost of a single projection in-place is less than transposition of A. However, for matrix factorization, it is desirable to transpose A if possible because many projections are needed.

Parallelization. Least squares projections in factorizations of rank-3 and greater are parallelized using the number of threads set by `setRcppMLthreads`. By default, all available threads are used, see `getRcppMLthreads`. The overhead of parallization is too great for rank-1 and -2 factorization.

L1 Regularization. Any L1 penalty is subtracted from b and should generally be scaled to `max(b)`, where $b = WA_j$ for all columns j in A. An easy way to properly scale an L1 penalty is to normalize all columns in w to sum to 1. No scaling is applied in this function. Such scaling guarantees that `L1 = 1` gives a completely sparse solution.

Specializations. There are specializations for symmetric input matrices, and for rank-1 and rank-2 projections. See documentation for `nmf` for theoretical details and guidance.

Publication reference. For theoretical and practical considerations, please see our manuscript: "DeBruine ZJ, Melcher K, Triche TJ (2021) High-performance non-negative matrix factorization for large single cell data." on BioRXiv.

Value

matrix h or w

Author(s)

Zach DeBruine

References

DeBruine, ZJ, Melcher, K, and Triche, TJ. (2021). "High-performance non-negative matrix factorization for large single-cell data." BioRXiv.

See Also

`nnls`, `nmf`

Examples

```
## Not run:
library(Matrix)
w <- matrix(runif(1000 * 10), 1000, 10)
h_true <- matrix(runif(10 * 100), 10, 100)
# A is the crossproduct of "w" and "h" with 10% signal dropout
A <- (w %*% h_true) * (rsparsematrix(1000, 100, 0.9) > 0)
h <- project(A, w)
cor(as.vector(h_true), as.vector(h))
```

```
# alternating projections refine solution (like NMF)
mse_bad <- mse(A, w, rep(1, ncol(w)), h) # mse before alternating updates
h <- project(A, w = w)
w <- project(A, h = h)
h <- project(A, w)
w <- project(A, h = h)
h <- project(A, w)
w <- t(project(A, h = h))
mse_better <- mse(A, w, rep(1, ncol(w)), h) # mse after alternating updates
mse_better < mse_bad

# two ways to solve for "w" that give the same solution
w <- project(A, h = h)
w2 <- project(t(A), w = t(h))
all.equal(w, w2)

## End(Not run)
```

RcppML *RcppML: Rcpp Machine Learning Library*

Description

High-performance non-negative matrix factorization and linear model projection for sparse matrices, and fast non-negative least squares implementations

Author(s)

Zach DeBruine

---

setRcppMLthreads *Set the number of threads RcppML should use*

Description

The number of threads is 0 by default (corresponding to all available threads), but can be set manually using this function. If you clear environment variables or affect the "RcppMLthreads" environment variable specifically, you will need to set your number of preferred threads again.

Usage

```
setRcppMLthreads(threads)
```

Arguments

threads	number of threads to be used in RcppML functions that are parallelized with OpenMP.

Details

The number of threads set affects OpenMP parallelization only for functions in the RcppML package. It does not affect other R packages that use OpenMP. Parallelization is used for projection of linear factor models with rank > 2, calculation of mean squared error for linear factor models, and for divisive clustering.

Author(s)

Zach DeBruine

See Also

getRcppMLthreads

Examples

```
## Not run:
# set serial configuration
setRcppMLthreads(1)
getRcppMLthreads()

# restore default parallel configuration,
# letting OpenMP decide how many threads to use
setRcppMLthreads(0)
getRcppMLthreads()

## End(Not run)
```

Index