Package 'NMF'

August 22, 2024

Type Package

Title Algorithms and Framework for Nonnegative Matrix Factorization (NMF)

Version 0.28

Date 2024-08-19

Maintainer Nicolas Sauwen <nicolas.sauwen@openanalytics.eu>

Description Provides a framework to perform Non-negative Matrix Factorization (NMF). The package implements a set of already published algorithms and seeding methods, and provides a framework to test, develop and plug new/custom algorithms. Most of the built-in algorithms have been optimized in C++, and the main interface function provides an easy way of performing parallel computations on multicore machines.

License GPL (>= 2)

URL http://renozao.github.io/NMF/

LazyLoad yes

VignetteBuilder knitr

Depends R (>= 3.0.0), methods, utils, registry, rngtools (>= 1.2.3), cluster

Imports graphics, stats, stringr (>= 1.0.0), digest, grid, grDevices, gridBase, colorspace, RColorBrewer, foreach, doParallel, ggplot2, reshape2, Biobase, codetools, BiocManager

Suggests fastICA, doMPI, bigmemory (>= 4.2), synchronicity(>= 1.3.2), corpcor, xtable, devtools, knitr, RUnit

Collate 'colorcode.R' 'options.R' 'grid.R' 'atracks.R' 'aheatmap.R' 'algorithmic.R' 'nmf-package.R' 'rmatrix.R' 'utils.R' 'versions.R' 'NMF-class.R' 'transforms.R' 'Bioc-layer.R' 'NMFstd-class.R' 'NMFOffset-class.R' 'heatmaps.R' 'NMFns-class.R' 'nmfModel.R' 'fixed-terms.R' 'NMFfit-class.R' 'NMFSet-class.R' 'NMFStrategy-class.R' 'registry.R' 'NMFSeed-class.R' 'NMFStrategyFunction-class.R' 'NMFStrategyIterative-class.R' 'NMFplots.R' 'registry-algorithms.R' 'algorithms-base.R' 'algorithms-lnmf.R'

'algorithms-lsnmf.R' 'algorithms-pe-nmf.R' 'algorithms-siNMF.R'
'algorithms-snmf.R' 'data.R' 'extractFeatures.R' 'parallel.R'
'registry-seed.R' 'nmf.R' 'rnmf.R' 'run.R' 'seed-base.R'
'seed-ica.R' 'seed-nndsvd.R' 'setNMFClass.R' 'simulation.R'
'tests.R' 'vignetteFunctions.R'

NeedsCompilation yes

Repository CRAN

Date/Publication 2024-08-22 16:20:01 UTC

RoxygenNote 7.3.1

Author Renaud Gaujoux [aut],
Cathal Seoighe [aut],
Nicolas Sauwen [cre]

Contents

`NMF-package` *Algorithms and framework for Nonnegative Matrix Factorization (NMF).*

Description

This package provides a framework to perform Non-negative Matrix Factorization (NMF). It implements a set of already published algorithms and seeding methods, and provides a framework to test, develop and plug new/custom algorithms. Most of the built-in algorithms have been optimized in C++, and the main interface function provides an easy way of performing parallel computations on multicore machines.

Details

nmf Run a given NMF algorithm

Author(s)

Renaud Gaujoux <renaud@cbio.uct.ac.za>

References

https://cran.r-project.org/

See Also

nmf

Examples

```
# generate a synthetic dataset with known classes
n <- 50; counts <- c(5, 5, 8);
V <- syntheticNMF(n, counts)

# perform a 3-rank NMF using the default algorithm
res <- nmf(V, 3)

basismap(res)
coefmap(res)
```

.fcnnls *Internal Routine for Fast Combinatorial Nonnegative Least-Squares*

Description

This is the workhorse function for the higher-level function fcnnls, which implements the fast nonnegative least-square algorithm for multiple right-hand-sides from *Van Benthem et al. (2004)* to solve the following problem:

$$\min \|Y - XK\|_F$$
$$\text{s.t. } K >= 0$$

where Y and X are two real matrices of dimension $n \times p$ and $n \times r$ respectively, and $\|.\|_F$ is the Frobenius norm.

The algorithm is very fast compared to other approaches, as it is optimised for handling multiple right-hand sides.

Usage

```
.fcnnls(x, y, verbose = FALSE, pseudo = FALSE, eps = 0)
```

Arguments

x	the coefficient matrix
y	the target matrix to be approximated by XK.
verbose	logical that indicates if log messages should be shown.
pseudo	By default (pseudo=FALSE) the algorithm uses Gaussian elimination to solve the successive internal linear problems, using the solve function. If pseudo=TRUE the algorithm uses Moore-Penrose generalized pseudoinverse from the corpcor package instead of solve.
eps	threshold for considering entries as nonnegative. This is an experimental parameter, and it is recommended to leave it at 0.

Value

A list with the following elements:

coef	the fitted coefficient matrix.
Pset	the set of passive constraints, as a logical matrix of the same size as K that indicates which element is positive.

References

Van Benthem M and Keenan MR (2004). "Fast algorithm for the solution of large-scale non-negativity-constrained least squares problems." _Journal of Chemometrics_, *18*(10), pp. 441-450. ISSN 0886-9383, <URL: http://dx.doi.org/10.1002/cem.889>, <URL: http://doi.wiley.com/10.1002/cem.889>.

---

advanced-NMF *Advanced Usage of the Package NMF*

Description

The functions documented here provide advanced functionalities useful when developing within the framework implemented in the NMF package.

which.best returns the index of the best fit in a list of NMF fit, according to some quantitative measure. The index of the fit with the lowest measure is returned.

Usage

```
which.best(object, FUN = deviance, ...)
```

Arguments

object	an NMF model fitted by multiple runs.
FUN	the function that computes the quantitative measure.
...	extra arguments passed to FUN.

`aggregate.measure` *Utility function to aggregate numerical quality measures from* `NMFfitXn` *objects.*

Description

Given a numerical vector, this function computes an aggregated value using one of the following methods: best or mean

Usage

```
## S3 method for class 'measure'
aggregate(x, method = c("best", "mean"), decreasing = FALSE, ...)
```

Arguments

`x`	a numerical vector
`method`	the method to aggregate values. This argument can take two values : - mean: the mean of the measures - best: the best measure according to the specified sorting order (decreasing or not)
`decreasing`	logical that specified the sorting order
`...`	extra arguments to allow extension

`aheatmap` *Annotated Heatmaps*

Description

The function `aheatmap` plots high-quality heatmaps, with a detailed legend and unlimited annotation tracks for both columns and rows. The annotations are coloured differently according to their type (factor or numeric covariate). Although it uses grid graphics, the generated plot is compatible with base layouts such as the ones defined with `'mfrow'` or `layout`, enabling the easy drawing of multiple heatmaps on a single a plot – at last!.

Usage

```
aheatmap(x, color = "-RdYlBu2:100", breaks = NA,
  border_color = NA, cellwidth = NA, cellheight = NA,
  scale = "none", Rowv = TRUE, Colv = TRUE,
  revC = identical(Colv, "Rowv") || is_NA(Rowv) || (is.integer(Rowv) &&
      length(Rowv) > 1) || is(Rowv, "silhouette"),
  distfun = "euclidean", hclustfun = "complete",
  reorderfun = function(d, w) reorder(d, w),
  treeheight = 50, legend = TRUE, annCol = NA,
```

```
annRow = NA, annColors = NA, annLegend = TRUE,
labRow = NULL, labCol = NULL, subsetRow = NULL,
subsetCol = NULL, txt = NULL, fontsize = 10,
cexRow = min(0.2 + 1/log10(nr), 1.2),
cexCol = min(0.2 + 1/log10(nc), 1.2), filename = NA,
width = NA, height = NA, main = NULL, sub = NULL,
info = NULL, verbose = getOption("verbose"),
gp = gpar())
```

Arguments

x	numeric matrix of the values to be plotted. An *ExpressionSet* object can also be passed, in which case the expression values are plotted (exprs(x)).
color	colour specification for the heatmap. Default to palette '-RdYlBu2:100', i.e. reversed palette 'RdYlBu2' (a slight modification of RColorBrewer's palette 'RdYlBu') with 100 colors. Possible values are: • a character/integer vector of length greater than 1 that is directly used and assumed to contain valid R color specifications. • a single color/integer (between 0 and 8)/other numeric value that gives the dominant colors. Numeric values are converted into a pallete by rev(sequential_hcl(2, h = x, l = c(50, 95))). Other values are concatenated with the grey colour '#F1F1F1'. • one of RColorBrewer's palette name (see display.brewer.all) , or one of 'RdYlBu2', 'rainbow', 'heat', 'topo', 'terrain', 'cm'. When the coluor palette is specified with a single value, and is negative or preceded a minus ('-'), the reversed palette is used. The number of breaks can also be specified after a colon (':'). For example, the default colour palette is specified as '-RdYlBu2:100'.
breaks	a sequence of numbers that covers the range of values in x and is one element longer than color vector. Used for mapping values to colors. Useful, if needed to map certain values to certain colors. If value is NA then the breaks are calculated automatically. If breaks is a single value, then the colour palette is centered on this value.
border_color	color of cell borders on heatmap, use NA if no border should be drawn.
cellwidth	individual cell width in points. If left as NA, then the values depend on the size of plotting window.
cellheight	individual cell height in points. If left as NA, then the values depend on the size of plotting window.
scale	character indicating how the values should scaled in either the row direction or the column direction. Note that the scaling is performed after row/column clustering, so that it has no effect on the row/column ordering. Possible values are: • "row": center and standardize each row separately to row Z-scores • "column": center and standardize each column separately to column Z-scores

- "r1": scale each row to sum up to one
- "c1": scale each column to sum up to one
- "none": no scaling

Rowv clustering specification(s) for the rows. It allows to specify the distance/clustering/ordering/display parameters to be used for the *rows only*. Possible values are:

- TRUE or NULL (to be consistent with heatmap): compute a dendrogram from hierarchical clustering using the distance and clustering methods distfun and hclustfun.
- NA: disable any ordering. In this case, and if not otherwise specified with argument revC=FALSE, the heatmap shows the input matrix with the rows in their original order, with the first row on top to the last row at the bottom. Note that this differ from the behaviour or heatmap, but seemed to be a more sensible choice when vizualizing a matrix without reordering.
- an integer vector of length the number of rows of the input matrix (nrow(x)), that specifies the row order. As in the case Rowv=NA, the ordered matrix is shown first row on top, last row at the bottom.
- a character vector or a list specifying values to use instead of arguments distfun, hclustfun and reorderfun when clustering the rows (see the respective argument descriptions for a list of accepted values). If Rowv has no names, then the first element is used for distfun, the second (if present) is used for hclustfun, and the third (if present) is used for reorderfun.
- a numeric vector of weights, of length the number of rows of the input matrix, used to reorder the internally computed dendrogram d by reorderfun(d, Rowv).
- FALSE: the dendrogram *is* computed using methods distfun, hclustfun, and reorderfun but is not shown.
- a single integer that specifies how many subtrees (i.e. clusters) from the computed dendrogram should have their root faded out. This can be used to better highlight the different clusters.
- a single double that specifies how much space is used by the computed dendrogram. That is that this value is used in place of treeheight.

Colv clustering specification(s) for the columns. It accepts the same values as argument Rowv (modulo the expected length for vector specifications), and allow specifying the distance/clustering/ordering/display parameters to be used for the *columns only*. Colv may also be set to "Rowv", in which case the dendrogram or ordering specifications applied to the rows are also applied to the columns. Note that this is allowed only for square input matrices, and that the row ordering is in this case by default reversed (revC=TRUE) to obtain the diagonal in the standard way (from top-left to bottom-right). See argument Rowv for other possible values.

revC a logical that specify if the *row order* defined by Rowv should be reversed. This is mainly used to get the rows displayed from top to bottom, which is not the case by default. Its default value is computed at runtime, to suit common situations where natural ordering is a more sensible choice: no or fix ordering of the rows (Rowv=NA or an integer vector of indexes – of length > 1), and when a symmetric ordering is requested – so that the diagonal is shown as expected. An argument

in favor of the "odd" default display (bottom to top) is that the row dendrogram is plotted from bottom to top, and reversing its reorder may take a not too long but non negligeable time.

`distfun` default distance measure used in clustering rows and columns. Possible values are:

- all the distance methods supported by `dist` (e.g. "euclidean" or "maximum").
- all correlation methods supported by `cor`, such as `"pearson"` or `"spearman"`. The pairwise distances between rows/columns are then computed as `d <- dist(1 - cor(..., method = distfun))`.
  One may as well use the string "correlation" which is an alias for "pearson".
- an object of class `dist` such as returned by `dist` or `as.dist`.

`hclustfun` default clustering method used to cluster rows and columns. Possible values are:

- a method name (a character string) supported by `hclust` (e.g. `'average'`).
- an object of class `hclust` such as returned by `hclust`
- a dendrogram

`reorderfun` default dendrogram reordering function, used to reorder the dendrogram, when either `Rowv` or `Colv` is a numeric weight vector, or provides or computes a dendrogram. It must take 2 parameters: a dendrogram, and a weight vector.

`subsetRow` Specification of subsetting the rows before drawing the heatmap. Possible values are:

- an integer vector of length > 1 specifying the indexes of the rows to keep;
- a character vector of length > 1 specyfing the names of the rows to keep. These are the original rownames, not the names specified in `labRow`.
- a logical vector of length > 1, whose elements are recycled if the vector has not as many elements as rows in `x`.

Note that in the case `Rowv` is a dendrogram or hclust object, it is first converted into an ordering vector, and cannot be displayed – and a warning is thrown.

`subsetCol` Specification of subsetting the columns before drawing the heatmap. It accepts the similar values as `subsetRow`. See details above.

`txt` character matrix of the same size as `x`, that contains text to display in each cell. `NA` values are allowed and are not displayed. See demo for an example.

`treeheight` how much space (in points) should be used to display dendrograms. If specified as a single value, it is used for both dendrograms. A length-2 vector specifies separate values for the row and column dendrogram respectively. Default value: 50 points.

`legend` boolean value that determines if a colour ramp for the heatmap's colour palette should be drawn or not. Default is `TRUE`.

`annCol` specifications of column annotation tracks displayed as coloured rows on top of the heatmaps. The annotation tracks are drawn from bottom to top. A single annotation track can be specified as a single vector; multiple tracks are specified as a list, a data frame, or an *ExpressionSet* object, in which case the phenotypic data is used (`pData(eset)`). Character or integer vectors are converted and displayed as factors. Unnamed tracks are internally renamed into `Xi`, with i

	being incremented for each unamed track, across both column and row annotation tracks. For each track, if no corresponding colour is specified in argument `annColors`, a palette or a ramp is automatically computed and named after the track's name.
`annRow`	specifications of row annotation tracks displayed as coloured columns on the left of the heatmaps. The annotation tracks are drawn from left to right. The same conversion, renaming and colouring rules as for argument `annCol` apply.
`annColors`	list for specifying annotation track colors manually. It is possible to define the colors for only some of the annotations. Check examples for details.
`annLegend`	boolean value specifying if the legend for the annotation tracks should be drawn or not. Default is `TRUE`.
`labRow`	labels for the rows.
`labCol`	labels for the columns. See description for argument `labRow` for a list of the possible values.
`fontsize`	base fontsize for the plot
`cexRow`	fontsize for the rownames, specified as a fraction of argument `fontsize`.
`cexCol`	fontsize for the colnames, specified as a fraction of argument `fontsize`.
`main`	Main title as a character string or a grob.
`sub`	Subtitle as a character string or a grob.
`info`	(experimental) Extra information as a character vector or a grob. If `info=TRUE`, information about the clustering methods is displayed at the bottom of the plot.
`filename`	file path ending where to save the picture. Currently following formats are supported: png, pdf, tiff, bmp, jpeg. Even if the plot does not fit into the plotting window, the file size is calculated so that the plot would fit there, unless specified otherwise.
`width`	manual option for determining the output file width in
`height`	manual option for determining the output file height in inches.
`verbose`	if `TRUE` then verbose messages are displayed and the borders of some viewports are highlighted. It is entended for debugging purposes.
`gp`	graphical parameters for the text used in plot. Parameters passed to `grid.text`, see `gpar`.

Details

The development of this function started as a fork of the function `pheatmap` from the **pheatmap** package, and provides several enhancements such as:

- argument names match those used in the base function `heatmap`;
- unlimited number of annotation for **both** columns and rows, with simplified and more flexible interface;
- easy specification of clustering methods and colors;
- return clustering data, as well as grid grob object.

Please read the associated vignette for more information and sample code.

PDF graphic devices

if plotting on a PDF graphic device – started with pdf, one may get generate a first blank page, due to internals of standard functions from the **grid** package that are called by aheatmap. The **NMF** package ships a custom patch that fixes this issue. However, in order to comply with CRAN policies, the patch is **not** applied by default and the user must explicitly be enabled it. This can be achieved on runtime by either setting the NMF specific option 'grid.patch' via `nmf.options(grid.patch=TRUE)`, or on load time if the environment variable 'R_PACKAGE_NMF_GRID_PATCH' is defined and its value is something that is not equivalent to FALSE (i.e. not '', 'false' nor 0).

Author(s)

Original version of pheatmap: Raivo Kolde

Enhancement into aheatmap: Renaud Gaujoux

Examples

```
## See the demo 'aheatmap' for more examples:
## Not run:
demo('aheatmap')

## End(Not run)

# Generate random data
n <- 50; p <- 20
x <- abs(rmatrix(n, p, rnorm, mean=4, sd=1))
x[1:10, seq(1, 10, 2)] <- x[1:10, seq(1, 10, 2)] + 3
x[11:20, seq(2, 10, 2)] <- x[11:20, seq(2, 10, 2)] + 2
rownames(x) <- paste("ROW", 1:n)
colnames(x) <- paste("COL", 1:p)

## Default heatmap
aheatmap(x)

## Distance methods
aheatmap(x, Rowv = "correlation")
aheatmap(x, Rowv = "man") # partially matched to 'manhattan'
aheatmap(x, Rowv = "man", Colv="binary")

# Generate column annotations
annotation = data.frame(Var1 = factor(1:p %% 2 == 0, labels = c("Class1", "Class2")), Var2 = 1:10)
aheatmap(x, annCol = annotation)
```

`algorithm,NMFList-method`

Returns the method names used to compute the NMF fits in the list. It returns NULL *if the list is empty.*

Description

Returns the method names used to compute the NMF fits in the list. It returns NULL if the list is empty.

Usage

```
  ## S4 method for signature 'NMFList'
algorithm(object, string = FALSE,
    unique = TRUE)
```

Arguments

string	a logical that indicate whether the names should be collapsed into a comma-separated string.
unique	a logical that indicates whether the result should contain the set of method names, removing duplicated names. This argument is forced to TRUE when string=TRUE.
object	an object computed using some algorithm, or that describes an algorithm itself.

---

algorithmic-NMF *Generic Interface for Algorithms*

Description

The functions documented here are S4 generics that define an general interface for – optimisation – algorithms.

This interface builds upon the broad definition of an algorithm as a workhorse function to which is associated auxiliary objects such as an underlying model or an objective function that measures the adequation of the model with observed data. It aims at complementing the interface provided by the stats package.

Usage

```
algorithm(object, ...)

algorithm(object, ...)<-value

seeding(object, ...)

seeding(object, ...)<-value

niter(object, ...)

niter(object, ...)<-value

nrun(object, ...)
```

```
objective(object, ...)

objective(object, ...)<-value

runtime(object, ...)

runtime.all(object, ...)

seqtime(object, ...)

modelname(object, ...)

run(object, y, x, ...)

logs(object, ...)

compare(object, ...)
```

Arguments

object	an object computed using some algorithm, or that describes an algorithm itself.
value	replacement value
...	extra arguments to allow extension
y	data object, e.g. a target matrix
x	a model object used as a starting point by the algorithm, e.g. a non-empty NMF model.

Details

`algorithm` and `algorithm<-` get/set an object that describes the algorithm used to compute another object, or with which it is associated. It may be a simple character string that gives the algorithm's names, or an object that includes the algorithm's definition itself (e.g. an `NMFStrategy` object).

`seeding` get/set the seeding method used to initialise the computation of an object, i.e. usually the function that sets the starting point of an algorithm.

`niter` and `niter<-` get/set the number of iterations performed to compute an object. The function `niter<-` would usually be called just before returning the result of an algorithm, when putting together data about the fit.

`nrun` returns the number of times the algorithm has been run to compute an object. Usually this will be 1, but may be be more if the algorithm involves multiple starting points.

`objective` and `objective<-` get/set the objective function associated with an object. Some methods for `objective` may also compute the objective value with respect to some target/observed data.

`runtime` returns the CPU time required to compute an object. This would generally be an object of class `proc_time`.

`runtime.all` returns the CPU time required to compute a collection of objects, e.g. a sequence of independent fits.

`seqtime` returns the sequential CPU time – that would be – required to compute a collection of objects. It would differ from `runtime.all` if the computations were performed in parallel.

`modelname` returns a the type of model associated with an object.

`run` calls the workhorse function that actually implements a strategy/algorithm, and run it on some data object.

`logs` returns the log messages output during the computation of an object.

`compare` compares objects obtained from running separate algorithms.

Methods

algorithm `signature(object = "NMFfit")`: Returns the name of the algorithm that fitted the NMF model `object`.

algorithm `signature(object = "NMFList")`: Returns the method names used to compute the NMF fits in the list. It returns `NULL` if the list is empty.

See `algorithm,NMFList-method` for more details.

algorithm `signature(object = "NMFfitXn")`: Returns the name of the common NMF algorithm used to compute all fits stored in `object`

Since all fits are computed with the same algorithm, this method returns the name of algorithm that computed the first fit. It returns `NULL` if the object is empty.

algorithm `signature(object = "NMFSeed")`: Returns the workhorse function of the seeding method described by `object`.

algorithm `signature(object = "NMFStrategyFunction")`: Returns the single R function that implements the NMF algorithm – as stored in slot `algorithm`.

algorithm<- `signature(object = "NMFSeed", value = "function")`: Sets the workhorse function of the seeding method described by `object`.

algorithm<- `signature(object = "NMFStrategyFunction", value = "function")`: Sets the function that implements the NMF algorithm, stored in slot `algorithm`.

compare `signature(object = "NMFfitXn")`: Compares the fits obtained by separate runs of NMF, in a single call to `nmf`.

logs `signature(object = "ANY")`: Default method that returns the value of attribute/slot `'logs'` or, if this latter does not exists, the value of element `'logs'` if `object` is a `list`. It returns `NULL` if no logging data was found.

modelname `signature(object = "ANY")`: Default method which returns the class name(s) of `object`. This should work for objects representing models on their own.

For NMF objects, this is the type of NMF model, that corresponds to the name of the S4 sub-class of `NMF`, inherited by `object`.

modelname `signature(object = "NMFfit")`: Returns the type of a fitted NMF model. It is a shortcut for `modelname(fit(object)`.

modelname `signature(object = "NMFfitXn")`: Returns the common type NMF model of all fits stored in `object`

Since all fits are from the same NMF model, this method returns the model type of the first fit. It returns `NULL` if the object is empty.

modelname `signature(object = "NMFStrategy")`: Returns the model(s) that an NMF algorithm can fit.

niter `signature(object = "NMFfit")`: Returns the number of iteration performed to fit an NMF model, typically with function `nmf`.

Currently this data is stored in slot `'extra'`, but this might change in the future.

niter<- `signature(object = "NMFfit", value = "numeric")`: Sets the number of iteration performed to fit an NMF model.

This function is used internally by the function `nmf`. It is not meant to be called by the user, except when developing new NMF algorithms implemented as single function, to set the number of iterations performed by the algorithm on the seed, before returning it (see `NMFStrategyFunction`).

nrun `signature(object = "ANY")`: Default method that returns the value of attribute 'nrun'.

Such an attribute my be attached to objects to keep track of data about the parent fit object (e.g. by method `consensus`), which can be used by subsequent function calls such as plot functions (e.g. see `consensusmap`). This method returns `NULL` if no suitable data was found.

nrun `signature(object = "NMFfitX")`: Returns the number of NMF runs performed to create `object`.

It is a pure virtual method defined to ensure `nrun` is defined for sub-classes of `NMFfitX`, which throws an error if called.

Note that because the `nmf` function allows to run the NMF computation keeping only the best fit, `nrun` may return a value greater than one, while only the result of the best run is stored in the object (cf. option `'k'` in method `nmf`).

nrun `signature(object = "NMFfit")`: This method always returns 1, since an `NMFfit` object is obtained from a single NMF run.

nrun `signature(object = "NMFfitX1")`: Returns the number of NMF runs performed, amongst which `object` was selected as the best fit.

nrun `signature(object = "NMFfitXn")`: Returns the number of runs performed to compute the fits stored in the list (i.e. the length of the list itself).

objective `signature(object = "NMFfit")`: Returns the objective function associated with the algorithm that computed the fitted NMF model `object`, or the objective value with respect to a given target matrix `y` if it is supplied.

See `objective,NMFfit-method` for more details.

runtime `signature(object = "NMFfit")`: Returns the CPU time required to compute a single NMF fit.

runtime `signature(object = "NMFList")`: Returns the CPU time required to compute all NMF fits in the list. It returns `NULL` if the list is empty. If no timing data are available, the sequential time is returned.

See `runtime,NMFList-method` for more details.

runtime.all `signature(object = "NMFfit")`: Identical to `runtime`, since their is a single fit.

runtime.all `signature(object = "NMFfitX")`: Returns the CPU time required to compute all the NMF runs. It returns `NULL` if no CPU data is available.

runtime.all `signature(object = "NMFfitXn")`: If no time data is available from in slot 'runtime.all' and argument `null=TRUE`, then the sequential time as computed by `seqtime` is returned, and a warning is thrown unless `warning=FALSE`.

See `runtime.all,NMFfitXn-method` for more details.

seeding `signature(object = "NMFfit")`: Returns the name of the seeding method that generated the starting point for the NMF algorithm that fitted the NMF model `object`.

seeding `signature(object = "NMFfitXn")`: Returns the name of the common seeding method used the computation of all fits stored in `object`

Since all fits are seeded using the same method, this method returns the name of the seeding method used for the first fit. It returns `NULL` if the object is empty.

seqtime `signature(object = "NMFList")`: Returns the CPU time that would be required to sequentially compute all NMF fits stored in `object`.

This method calls the function `runtime` on each fit and sum up the results. It returns `NULL` on an empty object.

seqtime `signature(object = "NMFfitXn")`: Returns the CPU time that would be required to sequentially compute all NMF fits stored in `object`.

This method calls the function `runtime` on each fit and sum up the results. It returns `NULL` on an empty object.

Interface fo NMF algorithms

This interface is implemented for NMF algorithms by the classes `NMFfit`, `NMFfitX` and `NMFStrategy`, and their respective sub-classes. The examples given in this documentation page are mainly based on this implementation.

Examples

```
#----------
# modelname,ANY-method
#----------
# get the type of an NMF model
modelname(nmfModel(3))
modelname(nmfModel(3, model='NMFns'))
modelname(nmfModel(3, model='NMFOffset'))

#----------
# modelname,NMFStrategy-method
#----------
# get the type of model(s) associated with an NMF algorithm
modelname( nmfAlgorithm('brunet') )
modelname( nmfAlgorithm('nsNMF') )
modelname( nmfAlgorithm('offset') )
```

`basis` *Accessing NMF Factors*

Description

`basis` and `basis<-` are S4 generic functions which respectively extract and set the matrix of basis components of an NMF model (i.e. the first matrix factor).

The methods `.basis`, `.coef` and their replacement versions are implemented as pure virtual methods for the interface class `NMF`, meaning that concrete NMF models must provide a definition for their corresponding class (i.e. sub-classes of class `NMF`). See `NMF` for more details.

`coef` and `coef<-` respectively extract and set the coefficient matrix of an NMF model (i.e. the second matrix factor). For example, in the case of the standard NMF model $V \equiv WH$, the method `coef` will return the matrix H.

`.coef` and `.coef<-` are low-level S4 generics that simply return/set coefficient data in an object, leaving some common processing to be performed in `coef` and `coef<-`.

Methods `coefficients` and `coefficients<-` are simple aliases for methods `coef` and `coef<-` respectively.

`scoef` is similar to `coef`, but returns the mixture coefficient matrix of an NMF model, with the columns scaled so that they sum up to a given value (1 by default).

Usage

```
  basis(object, ...)

  ## S4 method for signature 'NMF'
basis(object, all = TRUE, ...)

  .basis(object, ...)

  basis(object, ...)<-value

  ## S4 replacement method for signature 'NMF'
basis(object, use.dimnames = TRUE,
    ...)<-value

  .basis(object)<-value

  ## S4 method for signature 'NMF'
loadings(x)

  coef(object, ...)

  ## S4 method for signature 'NMF'
coef(object, all = TRUE, ...)

  .coef(object, ...)

  coef(object, ...)<-value

  ## S4 replacement method for signature 'NMF'
coef(object, use.dimnames = TRUE,
```

```
    ...)<-value

  .coef(object)<-value

  coefficients(object, ...)

  ## S4 method for signature 'NMF'
coefficients(object, all = TRUE, ...)

  scoef(object, ...)

  ## S4 method for signature 'NMF'
scoef(object, scale = 1)

  ## S4 method for signature 'matrix'
scoef(object, scale = 1)
```

Arguments

`object`	an object from which to extract the factor matrices, typically an object of class `NMF`.
`...`	extra arguments to allow extension and passed to the low-level access functions `.coef` and `.basis`. Note that these throw an error if used in replacement functions .
`all`	a logical that indicates whether the complete matrix factor should be returned (`TRUE`) or only the non-fixed part. This is relevant only for formula-based NMF models that include fixed basis or coefficient terms.
`use.dimnames`	logical that indicates if the object's dim names should be set using those from the new value, or left unchanged – after truncating them to fit new dimensions if necessary. This is useful to only set the entries of a factor.
`value`	replacement value
`scale`	scaling factor, which indicates to the value the columns of the coefficient matrix should sum up to.
`x`	an object of class `"factanal"` or `"princomp"` or the `loadings` component of such an object.

Details

For example, in the case of the standard NMF model $V \equiv WH$, the method `basis` will return the matrix W.

`basis` and `basis<-` are defined for the top virtual class `NMF` only, and rely internally on the low-level S4 generics `.basis` and `.basis<-` respectively that effectively extract/set the coefficient data. These data are post/pre-processed, e.g., to extract/set only their non-fixed terms or check dimension compatibility.

`coef` and `coef<-` are S4 methods defined for the corresponding generic functions from package `stats` (See coef). Similarly to `basis` and `basis<-`, they are defined for the top virtual class `NMF`

only, and rely internally on the S4 generics `.coef` and `.coef<-` respectively that effectively extract/set the coefficient data. These data are post/pre-processed, e.g., to extract/set only their non-fixed terms or check dimension compatibility.

Methods

basis `signature(object = "ANY")`: Default method returns the value of S3 slot or attribute `'basis'`. It returns `NULL` if none of these are set.

Arguments `...` are not used by this method.

basis `signature(object = "NMFfitXn")`: Returns the basis matrix of the best fit amongst all the fits stored in `object`. It is a shortcut for `basis(fit(object))`.

.basis `signature(object = "NMF")`: Pure virtual method for objects of class `NMF`, that should be overloaded by sub-classes, and throws an error if called.

.basis `signature(object = "NMFstd")`: Get the basis matrix in standard NMF models

This function returns slot `W` of `object`.

.basis `signature(object = "NMFfit")`: Returns the basis matrix from an NMF model fitted with function `nmf`.

It is a shortcut for `.basis(fit(object), ...)`, dispatching the call to the `.basis` method of the actual NMF model.

.basis<- `signature(object = "NMF", value = "matrix")`: Pure virtual method for objects of class `NMF`, that should be overloaded by sub-classes, and throws an error if called.

.basis<- `signature(object = "NMFstd", value = "matrix")`: Set the basis matrix in standard NMF models

This function sets slot `W` of `object`.

.basis<- `signature(object = "NMFfit", value = "matrix")`: Sets the the basis matrix of an NMF model fitted with function `nmf`.

It is a shortcut for `.basis(fit(object)) <- value`, dispatching the call to the `.basis<-` method of the actual NMF model. It is not meant to be used by the user, except when developing NMF algorithms, to update the basis matrix of the seed object before returning it.

basis<- `signature(object = "NMF")`: Default methods that calls `.basis<-` and check the validity of the updated object.

coef `signature(object = "NMFfitXn")`: Returns the coefficient matrix of the best fit amongst all the fits stored in `object`. It is a shortcut for `coef(fit(object))`.

.coef `signature(object = "NMF")`: Pure virtual method for objects of class `NMF`, that should be overloaded by sub-classes, and throws an error if called.

.coef `signature(object = "NMFstd")`: Get the mixture coefficient matrix in standard NMF models

This function returns slot `H` of `object`.

.coef `signature(object = "NMFfit")`: Returns the the coefficient matrix from an NMF model fitted with function `nmf`.

It is a shortcut for `.coef(fit(object), ...)`, dispatching the call to the `.coef` method of the actual NMF model.

.coef<- `signature(object = "NMF", value = "matrix")`: Pure virtual method for objects of class `NMF`, that should be overloaded by sub-classes, and throws an error if called.

.coef<- `signature(object = "NMFstd", value = "matrix")`: Set the mixture coefficient matrix in standard NMF models

This function sets slot `H` of `object`.

.coef<- `signature(object = "NMFfit", value = "matrix")`: Sets the the coefficient matrix of an NMF model fitted with function `nmf`.

It is a shortcut for `.coef(fit(object)) <- value`, dispatching the call to the `.coef<-` method of the actual NMF model. It is not meant to be used by the user, except when developing NMF algorithms, to update the coefficient matrix in the seed object before returning it.

coef<- `signature(object = "NMF")`: Default methods that calls `.coef<-` and check the validity of the updated object.

coefficients `signature(object = "NMF")`: Alias to `coef,NMF`, therefore also pure virtual.

loadings `signature(x = "NMF")`: Method loadings for NMF Models

The method `loadings` is identical to `basis`, but do not accept any extra argument.

The method `loadings` is provided to standardise the NMF interface against the one defined in the `stats` package, and emphasises the similarities between NMF and PCA or factorial analysis (see `loadings`).

See Also

Other NMF-interface: `.DollarNames,NMF-method`, `misc`, `NMF-class`, `$<-,NMF-method`, `$,NMF-method`, `nmfModel`, `nmfModels`, `rnmf`

Examples

```
#----------
# scoef
#----------
# Scaled coefficient matrix
x <- rnmf(3, 10, 5)
scoef(x)
scoef(x, 100)

#----------
# .basis,NMFstd-method
#----------
# random standard NMF model
x <- rnmf(3, 10, 5)
basis(x)
coef(x)

# set matrix factors
basis(x) <- matrix(1, nrow(x), nbasis(x))
coef(x) <- matrix(1, nbasis(x), ncol(x))
# set random factors
basis(x) <- rmatrix(basis(x))
coef(x) <- rmatrix(coef(x))

# incompatible matrices generate an error:
```

```
try( coef(x) <- matrix(1, nbasis(x)-1, nrow(x)) )
# but the low-level method allow it
.coef(x) <- matrix(1, nbasis(x)-1, nrow(x))
try( validObject(x) )
```

---

basiscor *Correlations in NMF Models*

Description

basiscor computes the correlation matrix between basis vectors, i.e. the *columns* of its basis matrix – which is the model's first matrix factor.

profcor computes the correlation matrix between basis profiles, i.e. the *rows* of the coefficient matrix – which is the model's second matrix factor.

Usage

```
basiscor(x, y, ...)

profcor(x, y, ...)
```

Arguments

x	a matrix or an object with suitable methods basis or coef.
y	a matrix or an object with suitable methods basis or coef, and dimensions compatible with x. If missing the correlations are computed between x and y=x.
...	extra arguments passed to cor.

Details

Each generic has methods defined for computing correlations between NMF models and/or compatible matrices. The computation is performed by the base function cor.

Methods

basiscor signature(x = "NMF", y = "matrix"): Computes the correlations between the basis vectors of x and the columns of y.

basiscor signature(x = "matrix", y = "NMF"): Computes the correlations between the columns of x and the the basis vectors of y.

basiscor signature(x = "NMF", y = "NMF"): Computes the correlations between the basis vectors of x and y.

basiscor signature(x = "NMF", y = "missing"): Computes the correlations between the basis vectors of x.

profcor signature(x = "NMF", y = "matrix"): Computes the correlations between the basis profiles of x and the rows of y.

profcor `signature(x = "matrix", y = "NMF")`: Computes the correlations between the rows of `x` and the basis profiles of `y`.

profcor `signature(x = "NMF", y = "NMF")`: Computes the correlations between the basis profiles of `x` and `y`.

profcor `signature(x = "NMF", y = "missing")`: Computes the correlations between the basis profiles of `x`.

Examples

```
# generate two random NMF models
a <- rnmf(3, 100, 20)
b <- rnmf(3, 100, 20)

# Compute auto-correlations
basiscor(a)
profcor(a)
# Compute correlations with b
basiscor(a, b)
profcor(a, b)

# try to recover the underlying NMF model 'a' from noisy data
res <- nmf(fitted(a) + rmatrix(a), 3)

# Compute correlations with the true model
basiscor(a, res)
profcor(a, res)

# Compute correlations with a random compatible matrix
W <- rmatrix(basis(a))
basiscor(a, W)
identical(basiscor(a, W), basiscor(W, a))

H <- rmatrix(coef(a))
profcor(a, H)
identical(profcor(a, H), profcor(H, a))
```

---

basisnames *Dimension names for NMF objects*

Description

The methods `dimnames`, `rownames`, `colnames` and `basisnames` and their respective replacement form allow to get and set the dimension names of the matrix factors in a NMF model.

`dimnames` returns all the dimension names in a single list. Its replacement form `dimnames<-` allows to set all dimension names at once.

`rownames`, `colnames` and `basisnames` provide separate access to each of these dimension names respectively. Their respective replacement form allow to set each dimension names separately.

Usage

```
basisnames(x, ...)

basisnames(x, ...)<-value

## S4 method for signature 'NMF'
dimnames(x)

## S4 replacement method for signature 'NMF'
dimnames(x)<-value
```

Arguments

x	an object with suitable basis and coef methods, such as an object that inherit from NMF.
...	extra argument to allow extension.
value	a character vector, or NULL or, in the case of dimnames<-, a list 2 or 3-length list of character vectors.

Details

The function basisnames is a new S4 generic defined in the package NMF, that returns the names of the basis components of an object. Its default method should work for any object, that has a suitable basis method defined for its class.

The method dimnames is implemented for the base generic dimnames, which make the base function rownames and colnames work directly.

Overall, these methods behave as their equivalent on matrix objects. The function basisnames<- ensures that the dimension names are handled in a consistent way on both factors, enforcing the names on both matrix factors simultaneously.

The function basisnames<- is a new S4 generic defined in the package NMF, that sets the names of the basis components of an object. Its default method should work for any object, that has suitable basis<- and coef<- methods method defined for its class.

Methods

basisnames signature(x = "ANY"): Default method which returns the column names of the basis matrix extracted from x, using the basis method.

For NMF objects these also correspond to the row names of the coefficient matrix.

basisnames<- signature(x = "ANY"): Default method which sets, respectively, the row and the column names of the basis matrix and coefficient matrix of x to value.

dimnames signature(x = "NMF"): Returns the dimension names of the NMF model x.

It returns either NULL if no dimnames are set on the object, or a 3-length list containing the row names of the basis matrix, the column names of the mixture coefficient matrix, and the column names of the basis matrix (i.e. the names of the basis components).

dimnames<- `signature(x = "NMF")`: sets the dimension names of the NMF model `x`.

`value` can be `NULL` which resets all dimension names, or a 1, 2 or 3-length list providing names at least for the rows of the basis matrix.

The optional second element of `value` (NULL if absent) is used to set the column names of the coefficient matrix. The optional third element of `value` (NULL if absent) is used to set both the column names of the basis matrix and the row names of the coefficient matrix.

Examples

```
# create a random NMF object
a <- rnmf(2, 5, 3)

# set dimensions
dims <- list( features=paste('f', 1:nrow(a), sep='')
, samples=paste('s', 1:ncol(a), sep='')
, basis=paste('b', 1:nbasis(a), sep='') )
dimnames(a) <- dims
dimnames(a)
basis(a)
coef(a)

# access the dimensions separately
rownames(a)
colnames(a)
basisnames(a)

# set only the first dimension (rows of basis): the other two dimnames are set to NULL
dimnames(a) <- dims[1]
dimnames(a)
basis(a)
coef(a)

# set only the two first dimensions (rows and columns of basis and coef respectively):
# the basisnames are set to NULL
dimnames(a) <- dims[1:2]
dimnames(a)
basis(a)

# reset the dimensions
dimnames(a) <- NULL
dimnames(a)
basis(a)
coef(a)

# set each dimensions separately
rownames(a) <- paste('X', 1:nrow(a), sep='') # only affect rows of basis
basis(a)

colnames(a) <- paste('Y', 1:ncol(a), sep='') # only affect columns of coef
coef(a)
```

```
basisnames(a) <- paste('Z', 1:nbasis(a), sep='') # affect both basis and coef matrices
basis(a)
coef(a)
```

---

bioc-NMF *Specific NMF Layer for Bioconductor*

Description

The package NMF provides an optional layer for working with common objects and functions defined in the Bioconductor platform.

Details

It provides:

- computation functions that support `ExpressionSet` objects as inputs.
- aliases and methods for generic functions defined and widely used by Bioconductor base packages.
- specialised visualisation methods that adapt the titles and legend using bioinformatics terminology.
- functions to link the results with annotations, etc...

---

canFit *Testing Compatibility of Algorithm and Models*

Description

`canFit` is an S4 generic that tests if an algorithm can fit a particular model.

Usage

```
  canFit(x, y, ...)

  ## S4 method for signature 'NMFStrategy,character'
canFit(x, y,
    exact = FALSE)
```

Arguments

`x`	an object that describes an algorithm
`y`	an object that describes a model
`...`	extra arguments to allow extension
`exact`	for logical that indicates if an algorithm is considered able to fit only the models that it explicitly declares (`TRUE`), or if it should be considered able to also fit models that extend models that it explicitly fits.

Methods

canFit `signature(x = "NMFStrategy", y = "character")`: Tells if an NMF algorithm can fit a given class of NMF models

canFit `signature(x = "NMFStrategy", y = "NMF")`: Tells if an NMF algorithm can fit the same class of models as y

canFit `signature(x = "character", y = "ANY")`: Tells if a registered NMF algorithm can fit a given NMF model

See Also

Other regalgo: nmfAlgorithm

---

compare-NMF *Comparing Results from Different NMF Runs*

Description

The functions documented here allow to compare the fits computed in different NMF runs. The fits do not need to be from the same algorithm, nor have the same dimension.

Usage

```
  ## S4 method for signature 'NMFfit'
compare(object, ...)

  ## S4 method for signature 'list'
compare(object, ...)

  ## S4 method for signature 'NMFList'
summary(object, sort.by = NULL,
    select = NULL, ...)

  ## S4 method for signature 'NMFList,missing'
plot(x, y, skip = -1, ...)

  ## S4 method for signature 'NMF.rank'
consensusmap(object, ...)

  ## S4 method for signature 'list'
consensusmap(object, layout,
    Rowv = FALSE, main = names(object), ...)
```

Arguments

`...`	extra arguments passed by `compare` to `summary,NMFList` or to the `summary` method of each fit.
`select`	the columns to be output in the result `data.frame`. The column are given by their names (partially matched). The column names are the names of the summary measures returned by the `summary` methods of the corresponding NMF results.
`sort.by`	the sorting criteria, i.e. a partial match of a column name, by which the result `data.frame` is sorted. The sorting direction (increasing or decreasing) is computed internally depending on the chosen criteria (e.g. decreasing for the cophenetic coefficient, increasing for the residuals).
`x`	an `NMFList` object that contains fits from separate NMF runs.
`y`	missing
`layout`	specification of the layout. It may be a single numeric or a numeric couple, to indicate a square or rectangular layout respectively, that is filled row by row. It may also be a matrix that is directly passed to the function `layout` from the package `graphics`.
`object`	an object computed using some algorithm, or that describes an algorithm itself.
`skip`	an integer that indicates the number of points to skip/remove from the beginning of the curve. If `skip=1L` (default) only the initial residual – that is computed before any iteration, is skipped, if present in the track (it associated with iteration 0).
`Rowv`	clustering specification(s) for the rows. It allows to specify the distance/clustering/ordering/display parameters to be used for the *rows only*. Possible values are:

- `TRUE` or `NULL` (to be consistent with `heatmap`): compute a dendrogram from hierarchical clustering using the distance and clustering methods `distfun` and `hclustfun`.
- `NA`: disable any ordering. In this case, and if not otherwise specified with argument `revC=FALSE`, the heatmap shows the input matrix with the rows in their original order, with the first row on top to the last row at the bottom. Note that this differ from the behaviour or `heatmap`, but seemed to be a more sensible choice when vizualizing a matrix without reordering.
- an integer vector of length the number of rows of the input matrix (`nrow(x)`), that specifies the row order. As in the case `Rowv=NA`, the ordered matrix is shown first row on top, last row at the bottom.
- a character vector or a list specifying values to use instead of arguments `distfun`, `hclustfun` and `reorderfun` when clustering the rows (see the respective argument descriptions for a list of accepted values). If `Rowv` has no names, then the first element is used for `distfun`, the second (if present) is used for `hclustfun`, and the third (if present) is used for `reorderfun`.
- a numeric vector of weights, of length the number of rows of the input matrix, used to reorder the internally computed dendrogram `d` by `reorderfun(d, Rowv)`.
- `FALSE`: the dendrogram *is* computed using methods `distfun`, `hclustfun`, and `reorderfun` but is not shown.

- a single integer that specifies how many subtrees (i.e. clusters) from the computed dendrogram should have their root faded out. This can be used to better highlight the different clusters.
- a single double that specifies how much space is used by the computed dendrogram. That is that this value is used in place of `treeheight`.

`main` Main title as a character string or a grob.

Details

The methods `compare` enables to compare multiple NMF fits either passed as arguments or as a list of fits. These methods eventually call the method `summary,NMFList`, so that all its arguments can be passed **named** in `...`.

Methods

compare `signature(object = "NMFfit")`: Compare multiple NMF fits passed as arguments.

compare `signature(object = "list")`: Compares multiple NMF fits passed as a standard list.

consensusmap `signature(object = "NMF.rank")`: Draw a single plot with a heatmap of the consensus matrix obtained for each value of the rank, in the range tested with `nmfEstimateRank`.

consensusmap `signature(object = "list")`: Draw a single plot with a heatmap of the consensus matrix of each element in the list `object`.

plot `signature(x = "NMFList", y = "missing")`: `plot` plot on a single graph the residuals tracks for each fit in `x`. See function `nmf` for details on how to enable the tracking of residuals.

summary `signature(object = "NMFList")`: `summary,NMFList` computes summary measures for each NMF result in the list and return them in rows in a `data.frame`. By default all the measures are included in the result, and `NA` values are used where no data is available or the measure does not apply to the result object (e.g. the dispersion for single' NMF runs is not meaningful). This method is very useful to compare and evaluate the performance of different algorithms.

Examples

```
#----------
# compare,NMFfit-method
#----------
x <- rmatrix(20,10)
res <- nmf(x, 3)
res2 <- nmf(x, 2, 'lee')

# compare arguments
compare(res, res2, target=x)

#----------
# compare,list-method
#----------
# compare elements of a list
compare(list(res, res2), target=x)
```

connectivity *Clustering Connectivity and Consensus Matrices*

Description

connectivity is an S4 generic that computes the connectivity matrix based on the clustering of samples obtained from a model's predict method.

The consensus matrix has been proposed by *Brunet et al. (2004)* to help visualising and measuring the stability of the clusters obtained by NMF approaches. For objects of class NMF (e.g. results of a single NMF run, or NMF models), the consensus matrix reduces to the connectivity matrix.

Usage

```
  connectivity(object, ...)

  ## S4 method for signature 'NMF'
connectivity(object, no.attrib = FALSE)

  consensus(object, ...)
```

Arguments

object	an object with a suitable predict method.
...	extra arguments to allow extension. They are passed to predict, except for the vector and factor methods.
no.attrib	a logical that indicates if attributes containing information about the NMF model should be attached to the result (TRUE) or not (FALSE).

Details

The connectivity matrix of a given partition of a set of samples (e.g. given as a cluster membership index) is the matrix C containing only 0 or 1 entries such that:

$$C_{ij} = \begin{cases} 1 \text{ if sample } i \text{ belongs to the same cluster as sample } j \\ 0 \text{ otherwise} \end{cases} .$$

Value

a square matrix of dimension the number of samples in the model, full of 0s or 1s.

Methods

connectivity signature(object = "ANY"): Default method which computes the connectivity matrix using the result of predict(x, ...) as cluster membership index.

connectivity signature(object = "factor"): Computes the connectivity matrix using x as cluster membership index.

connectivity `signature(object = "numeric")`: Equivalent to `connectivity(as.factor(x))`.

connectivity `signature(object = "NMF")`: Computes the connectivity matrix for an NMF model, for which cluster membership is given by the most contributing basis component in each sample. See `predict,NMF-method`.

consensus `signature(object = "NMFfitX")`: Pure virtual method defined to ensure `consensus` is defined for sub-classes of `NMFfitX`. It throws an error if called.

consensus `signature(object = "NMF")`: This method is provided for completeness and is identical to `connectivity`, and returns the connectivity matrix, which, in the case of a single NMF model, is also the consensus matrix.

consensus `signature(object = "NMFfitX1")`: The result is the matrix stored in slot 'consensus'. This method returns `NULL` if the consensus matrix is empty.

See `consensus,NMFfitX1-method` for more details.

consensus `signature(object = "NMFfitXn")`: This method returns `NULL` on an empty object. The result is a matrix with several attributes attached, that are used by plotting functions such as `consensusmap` to annotate the plots.

See `consensus,NMFfitXn-method` for more details.

References

Brunet J, Tamayo P, Golub TR and Mesirov JP (2004). "Metagenes and molecular pattern discovery using matrix factorization." _Proceedings of the National Academy of Sciences of the United States of America_, *101*(12), pp. 4164-9. ISSN 0027-8424, <URL: http://dx.doi.org/10.1073/pnas.0308531101>, <URL: http://www.ncbi.nlm.nih.gov/pubmed/15016911>.

See Also

`predict`

Examples

```
#----------
# connectivity,ANY-method
#----------
# clustering of random data
h <- hclust(dist(rmatrix(10,20)))
connectivity(cutree(h, 2))

#----------
# connectivity,factor-method
#----------
connectivity(gl(2, 4))
```

`consensus,NMFfitX1-method`

Returns the consensus matrix computed while performing all NMF runs, amongst which `object` *was selected as the best fit.*

Description

The result is the matrix stored in slot 'consensus'. This method returns `NULL` if the consensus matrix is empty.

Usage

```
  ## S4 method for signature 'NMFfitX1'
consensus(object, no.attrib = FALSE)
```

Arguments

`object`	an object with a suitable `predict` method.
`no.attrib`	a logical that indicates if attributes containing information about the NMF model should be attached to the result (`TRUE`) or not (`FALSE`).

`consensus,NMFfitXn-method`

Computes the consensus matrix of the set of fits stored in `object`*, as the mean connectivity matrix across runs.*

Description

This method returns `NULL` on an empty object. The result is a matrix with several attributes attached, that are used by plotting functions such as `consensusmap` to annotate the plots.

Usage

```
  ## S4 method for signature 'NMFfitXn'
consensus(object, ...,
    no.attrib = FALSE)
```

Arguments

`object`	an object with a suitable `predict` method.
`...`	extra arguments to allow extension. They are passed to `predict`, except for the `vector` and `factor` methods.
`no.attrib`	a logical that indicates if attributes containing information about the NMF model should be attached to the result (`TRUE`) or not (`FALSE`).

consensushc *Hierarchical Clustering of a Consensus Matrix*

Description

The function consensushc computes the hierarchical clustering of a consensus matrix, using the matrix itself as a similarity matrix and average linkage. It is

Usage

```
  consensushc(object, ...)

  ## S4 method for signature 'matrix'
consensushc(object,
    method = "average", dendrogram = TRUE)

  ## S4 method for signature 'NMFfitX'
consensushc(object,
    what = c("consensus", "fit"), ...)
```

Arguments

object	a matrix or an NMFfitX object, as returned by multiple NMF runs.
...	extra arguments passed to next method calls
method	linkage method passed to hclust.
dendrogram	a logical that specifies if the result of the hierarchical clustering (en hclust object) should be converted into a dendrogram. Default value is TRUE.
what	character string that indicates which matrix to use in the computation.

Value

an object of class dendrogram or hclust depending on the value of argument dendrogram.

Methods

consensushc signature(object = "matrix"): Workhorse method for matrices.

consensushc signature(object = "NMF"): Compute the hierarchical clustering on the connectivity matrix of object.

consensushc signature(object = "NMFfitX"): Compute the hierarchical clustering on the consensus matrix of object, or on the connectivity matrix of the best fit in object.

cophcor *Cophenetic Correlation Coefficient*

Description

The function cophcor computes the cophenetic correlation coefficient from consensus matrix object, e.g. as obtained from multiple NMF runs.

Usage

```
  cophcor(object, ...)

  ## S4 method for signature 'matrix'
cophcor(object, linkage = "average")
```

Arguments

object	an object from which is extracted a consensus matrix.
...	extra arguments to allow extension and passed to subsequent calls.
linkage	linkage method used in the hierarchical clustering. It is passed to hclust.

Details

The cophenetic correlation coeffificient is based on the consensus matrix (i.e. the average of connectivity matrices) and was proposed by *Brunet et al. (2004)* to measure the stability of the clusters obtained from NMF.

It is defined as the Pearson correlation between the samples' distances induced by the consensus matrix (seen as a similarity matrix) and their cophenetic distances from a hierachical clustering based on these very distances (by default an average linkage is used). See *Brunet et al. (2004)*.

Methods

cophcor `signature(object = "matrix")`: Workhorse method for matrices.

cophcor `signature(object = "NMFfitX")`: Computes the cophenetic correlation coefficient on the consensus matrix of object. All arguments in ... are passed to the method cophcor,matrix.

References

Brunet J, Tamayo P, Golub TR and Mesirov JP (2004). "Metagenes and molecular pattern discovery using matrix factorization." _Proceedings of the National Academy of Sciences of the United States of America_, *101*(12), pp. 4164-9. ISSN 0027-8424, <URL: http://dx.doi.org/10.1073/pnas.0308531101>, <URL: http://www.ncbi.nlm.nih.gov/pubmed/15016911>.

See Also

cophenetic

deviance *Distances and Objective Functions*

Description

The NMF package defines methods for the generic deviance from the package stats, to compute approximation errors between NMF models and matrices, using a variety of objective functions.

nmfDistance returns a function that computes the distance between an NMF model and a compatible matrix.

Usage

```
  deviance(object, ...)

  ## S4 method for signature 'NMF'
deviance(object, y,
    method = c("", "KL", "euclidean"), ...)

  nmfDistance(method = c("", "KL", "euclidean"))

  ## S4 method for signature 'NMFfit'
deviance(object, y, method, ...)

  ## S4 method for signature 'NMFStrategy'
deviance(object, x, y, ...)
```

Arguments

y	a matrix compatible with the NMF model object, i.e. y must have the same dimension as fitted(object).
method	a character string or a function with signature (x="NMF", y="matrix", ...) that implements a distance measure between an NMF model x and a target matrix y, i.e. an objective function to use to compute the deviance. In deviance, it is passed to nmfDistance to get the function that effectively computes the deviance.
...	extra parameters passed to the objective function.
x	an NMF model that estimates y.
object	an object for which the deviance is desired.

Value

deviance returns a nonnegative numerical value

nmfDistance returns a function with least two arguments: an NMF model and a matrix.

Methods

deviance `signature(object = "NMF")`: Computes the distance between a matrix and the estimate of an `NMF` model.

deviance `signature(object = "NMFfit")`: Returns the deviance of a fitted NMF model.

This method returns the final residual value if the target matrix `y` is not supplied, or the approximation error between the fitted NMF model stored in `object` and `y`. In this case, the computation is performed using the objective function `method` if not missing, or the objective of the algorithm that fitted the model (stored in slot `'distance'`).

If not computed by the NMF algorithm itself, the value is automatically computed at the end of the fitting process by the function `nmf`, using the objective function associated with the NMF algorithm, so that it should always be available.

deviance `signature(object = "NMFfitX")`: Returns the deviance achieved by the best fit object, i.e. the lowest deviance achieved across all NMF runs.

deviance `signature(object = "NMFStrategy")`: Computes the value of the objective function between the estimate `x` and the target `y`.

See Also

Other stats: `deviance,NMF-method`, `hasTrack`, `residuals`, `residuals<-`, `trackError`

`dispersion` *Dispersion of a Matrix*

Description

Computes the dispersion coefficient of a – consensus – matrix `object`, generally obtained from multiple NMF runs.

Usage

```
dispersion(object, ...)
```

Arguments

`object`	an object from which the dispersion is computed
`...`	extra arguments to allow extension

Details

The dispersion coefficient is based on the consensus matrix (i.e. the average of connectivity matrices) and was proposed by *Kim et al. (2007)* to measure the reproducibility of the clusters obtained from NMF.

It is defined as:

$$\rho = \sum_{i,j=1}^{n} 4(C_{ij} - \frac{1}{2})^2,$$

where n is the total number of samples.

By construction, $0 \leq \rho \leq 1$ and $\rho = 1$ only for a perfect consensus matrix, where all entries 0 or 1. A perfect consensus matrix is obtained only when all the connectivity matrices are the same, meaning that the algorithm gave the same clusters at each run. See *Kim et al. (2007)*.

Methods

dispersion `signature(object = "matrix")`: Workhorse method that computes the dispersion on a given matrix.

dispersion `signature(object = "NMFfitX")`: Computes the dispersion on the consensus matrix obtained from multiple NMF runs.

References

Kim H and Park H (2007). "Sparse non-negative matrix factorizations via alternating non-negativity-constrained least squares for microarray data analysis." _Bioinformatics (Oxford, England)_, *23*(12), pp. 1495-502. ISSN 1460-2059, <URL: http://dx.doi.org/10.1093/bioinformatics/btm134>, <URL: http://www.ncbi.nlm.nih.gov/pubmed/17483501>.

`esGolub` *Golub ExpressionSet*

Description

This data comes originally from the gene expression data from *Golub et al. (1999)*. The version included in the package is the one used and referenced in *Brunet et al. (2004)*. The samples are from 27 patients with acute lymphoblastic leukemia (ALL) and 11 patients with acute myeloid leukemia (AML).

Format

There are 3 covariates listed.

- Samples: The original sample labels.
- ALL.AML: Whether the patient had AML or ALL. It is a `factor` with levels `c('ALL', 'AML')`.
- Cell: ALL arises from two different types of lymphocytes (T-cell and B-cell). This specifies which for the ALL patients; There is no such information for the AML samples. It is a `factor` with levels `c('T-cell', 'B-cell', NA)`.

Details

The samples were assayed using Affymetrix Hgu6800 chips and the original data on the expression of 7129 genes (Affymetrix probes) are available on the Broad Institute web site (see references below).

The data in `esGolub` were obtained from the web page related to the paper from *Brunet et al. (2004)*, which describes an application of Nonnegative Matrix Factorization to gene expression clustering. (see link in section *Source*).

They contain the 5,000 most highly varying genes according to their coefficient of variation, and were installed in an object of class *ExpressionSet*.

Source

Original data from Golub et al.:
`http://www-genome.wi.mit.edu/mpr/data_set_ALL_AML.html`

References

Golub TR, Slonim DK, Tamayo P, Huard C, Gaasenbeek M, Mesirov JP, Coller H, Loh ML, Downing JR, Caligiuri Ma, Bloomfield CD and Lander ES (1999). "Molecular classification of cancer: class discovery and class prediction by gene expression monitoring." _Science (New York, N.Y.)_, *286*(5439), pp. 531-7. ISSN 0036-8075, <URL: http://www.ncbi.nlm.nih.gov/pubmed/10521349>.

Brunet J, Tamayo P, Golub TR and Mesirov JP (2004). "Metagenes and molecular pattern discovery using matrix factorization." _Proceedings of the National Academy of Sciences of the United States of America_, *101*(12), pp. 4164-9. ISSN 0027-8424, <URL: http://dx.doi.org/10.1073/pnas.0308531101>, <URL: http://www.ncbi.nlm.nih.gov/pubmed/15016911>.

Examples

```
# requires package Biobase to be installed
if(requireNamespace("Biobase", quietly=TRUE)){

data(esGolub)
esGolub
## Not run: pData(esGolub)

}
```

`fcnnls` *Fast Combinatorial Nonnegative Least-Square*

Description

This function solves the following nonnegative least square linear problem using normal equations and the fast combinatorial strategy from *Van Benthem et al. (2004)*:

$$\min \|Y - XK\|_F$$
$$\text{s.t. } K >= 0$$

where Y and X are two real matrices of dimension $n \times p$ and $n \times r$ respectively, and $\|.\|_F$ is the Frobenius norm.

The algorithm is very fast compared to other approaches, as it is optimised for handling multiple right-hand sides.

Usage

```
  fcnnls(x, y, ...)

  ## S4 method for signature 'matrix,matrix'
fcnnls(x, y, verbose = FALSE,
    pseudo = TRUE, ...)
```

Arguments

`...`	extra arguments passed to the internal function `.fcnnls`. Currently not used.
`verbose`	toggle verbosity (default is `FALSE`).
`x`	the coefficient matrix
`y`	the target matrix to be approximated by XK.
`pseudo`	By default (`pseudo=FALSE`) the algorithm uses Gaussian elimination to solve the successive internal linear problems, using the `solve` function. If `pseudo=TRUE` the algorithm uses Moore-Penrose generalized `pseudoinverse` from the `corpcor` package instead of solve.

Details

Within the `NMF` package, this algorithm is used internally by the SNMF/R(L) algorithm from *Kim et al. (2007)* to solve general Nonnegative Matrix Factorization (NMF) problems, using alternating nonnegative constrained least-squares. That is by iteratively and alternatively estimate each matrix factor.

The algorithm is an active/passive set method, which rearrange the right-hand side to reduce the number of pseudo-inverse calculations. It uses the unconstrained solution K_u obtained from the unconstrained least squares problem, i.e. $\min \|Y - XK\|_F^2$, so as to determine the initial passive sets.

The function `fcnnls` is provided separately so that it can be used to solve other types of nonnegative least squares problem. For faster computation, when multiple nonnegative least square fits are needed, it is recommended to directly use the function `.fcnnls`.

The code of this function is a port from the original MATLAB code provided by *Kim et al. (2007)*.

Value

A list containing the following components:

`x`	the estimated optimal matrix K.
`fitted`	the fitted matrix XK.
`residuals`	the residual matrix $Y - XK$.
`deviance`	the residual sum of squares between the fitted matrix XK and the target matrix Y. That is the sum of the square residuals.
`passive`	a rxp logical matrix containing the passive set, that is the set of entries in K that are not null (i.e. strictly positive).
`pseudo`	a logical that is `TRUE` if the computation was performed using the pseudoinverse. See argument `pseudo`.

Methods

fcnnls `signature(x = "matrix", y = "matrix")`: This method wraps a call to the internal function `.fcnnls`, and formats the results in a similar way as other lest-squares methods such as `lm`.

fcnnls `signature(x = "numeric", y = "matrix")`: Shortcut for `fcnnls(as.matrix(x), y, ...)`.

fcnnls `signature(x = "ANY", y = "numeric")`: Shortcut for `fcnnls(x, as.matrix(y), ...)`.

Author(s)

Original MATLAB code : Van Benthem and Keenan

Adaption of MATLAB code for SNMF/R(L): H. Kim

Adaptation to the NMF package framework: Renaud Gaujoux

References

Original MATLAB code from Van Benthem and Keenan, slightly modified by H. Kim:(http://www.cc.gatech.edu/~hpark/soft

Van Benthem M and Keenan MR (2004). "Fast algorithm for the solution of large-scale non-negativity-constrained least squares problems." _Journal of Chemometrics_, *18*(10), pp. 441-450. ISSN 0886-9383, <URL: http://dx.doi.org/10.1002/cem.889>, <URL: http://doi.wiley.com/10.1002/cem.889>.

Kim H and Park H (2007). "Sparse non-negative matrix factorizations via alternating non-negativity-constrained least squares for microarray data analysis." _Bioinformatics (Oxford, England)_, *23*(12), pp. 1495-502. ISSN 1460-2059, <URL: http://dx.doi.org/10.1093/bioinformatics/btm134>, <URL: http://www.ncbi.nlm.nih.gov/pubmed/17483501>.

See Also

`nmf`

Examples

```
## Define a random nonnegative matrix matrix
n <- 200; p <- 20; r <- 3
V <- rmatrix(n, p)

## Compute the optimal matrix K for a given X matrix
X <- rmatrix(n, r)
res <- fcnnls(X, V)

## Compute the same thing using the Moore-Penrose generalized pseudoinverse
res <- fcnnls(X, V, pseudo=TRUE)

## It also works in the case of single vectors
y <- runif(n)
res <- fcnnls(X, y)
# or
res <- fcnnls(X[,1], y)
```

---

featureScore *Feature Selection in NMF Models*

Description

The function `featureScore` implements different methods to computes basis-specificity scores for each feature in the data.

The function `extractFeatures` implements different methods to select the most basis-specific features of each basis component.

Usage

```
  featureScore(object, ...)

  ## S4 method for signature 'matrix'
featureScore(object,
    method = c("kim", "max"))

  extractFeatures(object, ...)

  ## S4 method for signature 'matrix'
extractFeatures(object,
    method = c("kim", "max"),
    format = c("list", "combine", "subset"), nodups = TRUE)
```

Arguments

`object`	an object from which scores/features are computed/extracted
`...`	extra arguments to allow extension

`method` scoring or selection method. It specifies the name of one of the method described in sections *Feature scores* and *Feature selection*.

Additionally for `extractFeatures`, it may be an integer vector that indicates the number of top most contributing features to extract from each column of `object`, when ordered in decreasing order, or a numeric value between 0 and 1 that indicates the minimum relative basis contribution above which a feature is selected (i.e. basis contribution threshold). In the case of a single numeric value (integer or percentage), it is used for all columns.

Note that `extractFeatures(x, 1)` means relative contribution threshold of 100%, to select the top contributing features one must explicitly specify an integer value as in `extractFeatures(x, 1L)`. However, if all elements in methods are > 1, they are automatically treated as if they were integers: `extractFeatures(x, 2)` means the top-2 most contributing features in each component.

`format` output format. The following values are accepted:

'list' (default) returns a list with one element per column in `object`, each containing the indexes of the selected features, as an integer vector. If `object` has row names, these are used to name each index vector. Components for which no feature were selected are assigned a `NA` value.

'combine' returns all indexes in a single vector. Duplicated indexes are made unique if `nodups=TRUE` (default).

'subset' returns an object of the same class as `object`, but subset with the selected indexes, so that it contains data only from basis-specific features.

`nodups` logical that indicates if duplicated indexes, i.e. features selected on multiple basis components (which should in theory not happen), should be only appear once in the result. Only used when `format='combine'`.

Details

One of the properties of Nonnegative Matrix Factorization is that is tend to produce sparse representation of the observed data, leading to a natural application to bi-clustering, that characterises groups of samples by a small number of features.

In NMF models, samples are grouped according to the basis components that contributes the most to each sample, i.e. the basis components that have the greatest coefficient in each column of the coefficient matrix (see `predict,NMF-method`). Each group of samples is then characterised by a set of features selected based on basis-specifity scores that are computed on the basis matrix.

Value

`featureScore` returns a numeric vector of the length the number of rows in `object` (i.e. one score per feature).

`extractFeatures` returns the selected features as a list of indexes, a single integer vector or an object of the same class as `object` that only contains the selected features.

Methods

extractFeatures `signature(object = "matrix")`: Select features on a given matrix, that contains the basis component in columns.

extractFeatures `signature(object = "NMF")`: Select basis-specific features from an NMF model, by applying the method `extractFeatures,matrix` to its basis matrix.

featureScore `signature(object = "matrix")`: Computes feature scores on a given matrix, that contains the basis component in columns.

featureScore `signature(object = "NMF")`: Computes feature scores on the basis matrix of an NMF model.

Feature scores

The function `featureScore` can compute basis-specificity scores using the following methods:

'kim' Method defined by *Kim et al. (2007)*.

The score for feature i is defined as:

$$S_i = 1 + \frac{1}{\log_2 k} \sum_{q=1}^{k} p(i,q) \log_2 p(i,q)$$

,

where $p(i,q)$ is the probability that the i-th feature contributes to basis q:

$$p(i,q) = \frac{W(i,q)}{\sum_{r=1}^{k} W(i,r)}$$

The feature scores are real values within the range [0,1]. The higher the feature score the more basis-specific the corresponding feature.

'max' Method defined by *Carmona-Saez et al. (2006)*.

The feature scores are defined as the row maximums.

Feature selection

The function `extractFeatures` can select features using the following methods:

'kim' uses *Kim et al. (2007)* scoring schema and feature selection method.

The features are first scored using the function `featureScore` with method 'kim'. Then only the features that fulfil both following criteria are retained:

- score greater than $\hat{\mu}+3\hat{\sigma}$, where $\hat{\mu}$ and $\hat{\sigma}$ are the median and the median absolute deviation (MAD) of the scores respectively;
- the maximum contribution to a basis component is greater than the median of all contributions (i.e. of all elements of W).

'max' uses the selection method used in the `bioNMF` software package and described in *Carmona-Saez et al. (2006)*.

For each basis component, the features are first sorted by decreasing contribution. Then, one selects only the first consecutive features whose highest contribution in the basis matrix is effectively on the considered basis.

References

Kim H and Park H (2007). "Sparse non-negative matrix factorizations via alternating non-negativity-constrained least squares for microarray data analysis." _Bioinformatics (Oxford, England)_, *23*(12), pp. 1495-502. ISSN 1460-2059, <URL: http://dx.doi.org/10.1093/bioinformatics/btm134>, <URL: http://www.ncbi.nlm.nih.gov/pubmed/17483501>.

Carmona-Saez P, Pascual-Marqui RD, Tirado F, Carazo JM and Pascual-Montano A (2006). "Biclustering of gene expression data by Non-smooth Non-negative Matrix Factorization." _BMC bioinformatics_, *7*, pp. 78. ISSN 1471-2105, <URL: http://dx.doi.org/10.1186/1471-2105-7-78>, <URL: http://www.ncbi.nlm.nih.gov/pubmed/16503973>.

Examples

```
# random NMF model
x <- rnmf(3, 50,20)

# probably no feature is selected
extractFeatures(x)
# extract top 5 for each basis
extractFeatures(x, 5L)
# extract features that have a relative basis contribution above a threshold
extractFeatures(x, 0.5)
# ambiguity?
extractFeatures(x, 1) # means relative contribution above 100%
extractFeatures(x, 1L) # means top contributing feature in each component
```

`fit` *Extracting Fitted Models*

Description

The functions `fit` and `minfit` are S4 genetics that extract the best model object and the best fit object respectively, from a collection of models or from a wrapper object.

`fit<-` sets the fitted model in a fit object. It is meant to be called only when developing new NMF algorithms, e.g. to update the value of the model stored in the starting point.

Usage

```
fit(object, ...)

fit(object)<-value

minfit(object, ...)
```

Arguments

object	an object fitted by some algorithm, e.g. as returned by the function nmf.
value	replacement value
...	extra arguments to allow extension

Details

A fit object differs from a model object in that it contains data about the fit, such as the initial RNG settings, the CPU time used, etc..., while a model object only contains the actual modelling data such as regression coefficients, loadings, etc...

That best model is generally defined as the one that achieves the maximum/minimum some quantitative measure, amongst all models in a collection.

In the case of NMF models, the best model is the one that achieves the best approximation error, according to the objective function associated with the algorithm that performed the fit(s).

Methods

fit `signature(object = "NMFfit")`: Returns the NMF model object stored in slot `'fit'`.

fit `signature(object = "NMFfitX")`: Returns the model object that achieves the lowest residual approximation error across all the runs.

It is a pure virtual method defined to ensure `fit` is defined for sub-classes of `NMFfitX`, which throws an error if called.

fit `signature(object = "NMFfitX1")`: Returns the model object associated with the best fit, amongst all the runs performed when fitting `object`.

Since `NMFfitX1` objects only hold the best fit, this method simply returns the NMF model fitted by `object` – that is stored in slot 'fit'.

fit `signature(object = "NMFfitXn")`: Returns the best NMF fit object amongst all the fits stored in `object`, i.e. the fit that achieves the lowest estimation residuals.

fit<- `signature(object = "NMFfit", value = "NMF")`: Updates the NMF model object stored in slot `'fit'` with a new value.

minfit `signature(object = "NMFfit")`: Returns the object its self, since there it is the result of a single NMF run.

minfit `signature(object = "NMFfitX")`: Returns the fit object that achieves the lowest residual approximation error across all the runs.

It is a pure virtual method defined to ensure `minfit` is defined for sub-classes of `NMFfitX`, which throws an error if called.

minfit `signature(object = "NMFfitX1")`: Returns the fit object associated with the best fit, amongst all the runs performed when fitting `object`.

Since `NMFfitX1` objects only hold the best fit, this method simply returns `object` coerced into an `NMFfit` object.

minfit `signature(object = "NMFfitXn")`: Returns the best NMF model in the list, i.e. the run that achieved the lower estimation residuals.

The model is selected based on its `deviance` value.

`fitted` *Fitted Matrix in NMF Models*

Description

Computes the estimated target matrix based on a given *NMF* model. The estimation depends on the underlying NMF model. For example in the standard model $V \equiv WH$, the target matrix is estimated by the matrix product WH. In other models, the estimate may depend on extra parameters/matrix (cf. Non-smooth NMF in `NMFns-class`).

Usage

```
  fitted(object, ...)

  ## S4 method for signature 'NMFstd'
fitted(object, W, H, ...)

  ## S4 method for signature 'NMFOffset'
fitted(object, W, H,
    offset = object@offset)

  ## S4 method for signature 'NMFns'
fitted(object, W, H, S, ...)
```

Arguments

`object`	an object that inherit from class `NMF`
`...`	extra arguments to allow extension
`W`	a matrix to use in the computation as the basis matrix in place of `basis(object)`. It must be compatible with the coefficient matrix used in the computation (i.e. number of columns in `W` = number of rows in `H`).
`H`	a matrix to use in the computation as the coefficient matrix in place of `coef(object)`. It must be compatible with the basis matrix used in the computation (i.e. number of rows in `H` = number of columns in `W`).
`offset`	offset vector
`S`	smoothing matrix to use instead of `smoothing(object)` It must be a square matrix compatible with the basis and coefficient matrices used in the computation.

Details

This function is a S4 generic function imported from fitted in the package *stats*. It is implemented as a pure virtual method for objects of class `NMF`, meaning that concrete NMF models must provide a definition for their corresponding class (i.e. sub-classes of class `NMF`). See `NMF` for more details.

Value

the target matrix estimate as fitted by the model `object`

Methods

fitted `signature(object = "NMF")`: Pure virtual method for objects of class `NMF`, that should be overloaded by sub-classes, and throws an error if called.

fitted `signature(object = "NMFstd")`: Compute the target matrix estimate in *standard NMF models*.

The estimate matrix is computed as the product of the two matrix slots `W` and `H`:

$$\hat{V} = WH$$

fitted `signature(object = "NMFOffset")`: Computes the target matrix estimate for an NMFOffset object.

The estimate is computed as:

$$WH + offset$$

fitted `signature(object = "NMFns")`: Compute estimate for an NMFns object, according to the Nonsmooth NMF model (cf. `NMFns-class`).

Extra arguments in `...` are passed to method `smoothing`, and are typically used to pass a value for `theta`, which is used to compute the smoothing matrix instead of the one stored in `object`.

fitted `signature(object = "NMFfit")`: Computes and return the estimated target matrix from an NMF model fitted with function `nmf`.

It is a shortcut for `fitted(fit(object), ...)`, dispatching the call to the `fitted` method of the actual NMF model.

Examples

```
# random standard NMF model
x <- rnmf(3, 10, 5)
all.equal(fitted(x), basis(x) %*% coef(x))
```

`getRNG1` *Extracting RNG Data from NMF Objects*

Description

The `nmf` function returns objects that contain embedded RNG data, that can be used to exactly reproduce any computation. These data can be extracted using dedicated methods for the S4 generics `getRNG` and `getRNG1`.

Usage

```
getRNG1(object, ...)

.getRNG(object, ...)
```

Arguments

`object`	an R object from which RNG settings can be extracted, e.g. an integer vector containing a suitable value for `.Random.seed` or embedded RNG data, e.g., in S3/S4 slot `rng` or `rng$noise`.
`...`	extra arguments to allow extension and passed to a suitable S4 method `.getRNG` or `.setRNG`.

Methods

.getRNG `signature(object = "NMFfitXn")`: Returns the RNG settings used for the best fit.

This method throws an error if the object is empty.

getRNG1 `signature(object = "NMFfitX")`: Returns the RNG settings used for the first NMF run of multiple NMF runs.

getRNG1 `signature(object = "NMFfitX1")`: Returns the RNG settings used to compute the first of all NMF runs, amongst which `object` was selected as the best fit.

getRNG1 `signature(object = "NMFfitXn")`: Returns the RNG settings used for the first run.

This method throws an error if the object is empty.

Examples

```
# For multiple NMF runs, the RNG settings used for the first run is also stored
V <- rmatrix(20,10)
res <- nmf(V, 3, nrun=3)
# RNG used for the best fit
getRNG(res)
# RNG used for the first of all fits
getRNG1(res)
# they may differ if the best fit is not the first one
rng.equal(res, getRNG1(res))
```

---

heatmap-NMF *Heatmaps of NMF Factors*

Description

The NMF package ships an advanced heatmap engine implemented by the function `aheatmap`. Some convenience heatmap functions have been implemented for NMF models, which redefine default values for some of the arguments of `aheatmap`, hence tuning the output specifically for NMF models.

Usage

```
  basismap(object, ...)

  ## S4 method for signature 'NMF'
basismap(object, color = "YlOrRd:50",
    scale = "r1", Rowv = TRUE, Colv = NA,
    subsetRow = FALSE, annRow = NA, annCol = NA,
    tracks = "basis", main = "Basis components",
    info = FALSE, ...)

  coefmap(object, ...)

  ## S4 method for signature 'NMF'
coefmap(object, color = "YlOrRd:50",
    scale = "c1", Rowv = NA, Colv = TRUE, annRow = NA,
    annCol = NA, tracks = "basis",
    main = "Mixture coefficients", info = FALSE, ...)

  consensusmap(object, ...)

  ## S4 method for signature 'NMFfitX'
consensusmap(object, annRow = NA,
    annCol = NA,
    tracks = c("basis:", "consensus:", "silhouette:"),
    main = "Consensus matrix", info = FALSE, ...)

  ## S4 method for signature 'matrix'
consensusmap(object,
    color = "-RdYlBu",
    distfun = function(x) as.dist(1 - x),
    hclustfun = "average", Rowv = TRUE, Colv = "Rowv",
   main = if (is.null(nr) || nr > 1) "Consensus matrix" else "Connectiviy matrix",
    info = FALSE, ...)

  ## S4 method for signature 'NMFfitX'
coefmap(object, Colv = TRUE,
    annRow = NA, annCol = NA,
    tracks = c("basis", "consensus:"), ...)
```

Arguments

object	an object from which is extracted NMF factors or a consensus matrix
...	extra arguments passed to aheatmap.
subsetRow	Argument that specifies how to filter the rows that will appear in the heatmap. When FALSE (default), all rows are used. Besides the values supported by argument subsetRow of aheatmap, other possible values are: • TRUE: only the rows that are basis-specific are used. The default selection method is from *KimH2007*. This is equivalent to subsetRow='kim'.

- a single character string or numeric value that specifies the method to use to select the basis-specific rows, that should appear in the heatmap (cf. argument method for function extractFeatures).
  Note extractFeatures is called with argument nodups=TRUE, so that features that are selected for multiple components only appear once.

tracks — Special additional annotation tracks to highlight associations between basis components and sample clusters:

basis matches each row (resp. column) to the most contributing basis component in basismap (resp. coefmap). In basismap (resp. coefmap), adding a track ':basis' to annCol (resp. annRow) makes the column (resp. row) corresponding to the component being also highlited using the mathcing colours.

info — if TRUE then the name of the algorithm that fitted the NMF model is displayed at the bottom of the plot, if available. Other wise it is passed as is to aheatmap.

color — colour specification for the heatmap. Default to palette '-RdYlBu2:100', i.e. reversed palette 'RdYlBu2' (a slight modification of RColorBrewer's palette 'RdYlBu') with 100 colors. Possible values are:

- a character/integer vector of length greater than 1 that is directly used and assumed to contain valid R color specifications.
- a single color/integer (between 0 and 8)/other numeric value that gives the dominant colors. Numeric values are converted into a pallete by rev(sequential_hcl(2, h = x, l = c(50, 95))). Other values are concatenated with the grey colour '#F1F1F1'.
- one of RColorBrewer's palette name (see display.brewer.all) , or one of 'RdYlBu2', 'rainbow', 'heat', 'topo', 'terrain', 'cm'.

When the coluor palette is specified with a single value, and is negative or preceded a minus ('-'), the reversed palette is used. The number of breaks can also be specified after a colon (':'). For example, the default colour palette is specified as '-RdYlBu2:100'.

scale — character indicating how the values should scaled in either the row direction or the column direction. Note that the scaling is performed after row/column clustering, so that it has no effect on the row/column ordering. Possible values are:

- "row": center and standardize each row separately to row Z-scores
- "column": center and standardize each column separately to column Z-scores
- "r1": scale each row to sum up to one
- "c1": scale each column to sum up to one
- "none": no scaling

Rowv — clustering specification(s) for the rows. It allows to specify the distance/clustering/ordering/display parameters to be used for the *rows only*. Possible values are:

- TRUE or NULL (to be consistent with heatmap): compute a dendrogram from hierarchical clustering using the distance and clustering methods distfun and hclustfun.

- NA: disable any ordering. In this case, and if not otherwise specified with argument revC=FALSE, the heatmap shows the input matrix with the rows in their original order, with the first row on top to the last row at the bottom. Note that this differ from the behaviour or heatmap, but seemed to be a more sensible choice when vizualizing a matrix without reordering.
- an integer vector of length the number of rows of the input matrix (nrow(x)), that specifies the row order. As in the case Rowv=NA, the ordered matrix is shown first row on top, last row at the bottom.
- a character vector or a list specifying values to use instead of arguments distfun, hclustfun and reorderfun when clustering the rows (see the respective argument descriptions for a list of accepted values). If Rowv has no names, then the first element is used for distfun, the second (if present) is used for hclustfun, and the third (if present) is used for reorderfun.
- a numeric vector of weights, of length the number of rows of the input matrix, used to reorder the internally computed dendrogram d by reorderfun(d, Rowv).
- FALSE: the dendrogram *is* computed using methods distfun, hclustfun, and reorderfun but is not shown.
- a single integer that specifies how many subtrees (i.e. clusters) from the computed dendrogram should have their root faded out. This can be used to better highlight the different clusters.
- a single double that specifies how much space is used by the computed dendrogram. That is that this value is used in place of treeheight.

Colv — clustering specification(s) for the columns. It accepts the same values as argument Rowv (modulo the expected length for vector specifications), and allow specifying the distance/clustering/ordering/display parameters to be used for the *columns only*. Colv may also be set to "Rowv", in which case the dendrogram or ordering specifications applied to the rows are also applied to the columns. Note that this is allowed only for square input matrices, and that the row ordering is in this case by default reversed (revC=TRUE) to obtain the diagonal in the standard way (from top-left to bottom-right). See argument Rowv for other possible values.

annRow — specifications of row annotation tracks displayed as coloured columns on the left of the heatmaps. The annotation tracks are drawn from left to right. The same conversion, renaming and colouring rules as for argument annCol apply.

annCol — specifications of column annotation tracks displayed as coloured rows on top of the heatmaps. The annotation tracks are drawn from bottom to top. A single annotation track can be specified as a single vector; multiple tracks are specified as a list, a data frame, or an *ExpressionSet* object, in which case the phenotypic data is used (pData(eset)). Character or integer vectors are converted and displayed as factors. Unnamed tracks are internally renamed into Xi, with i being incremented for each unamed track, across both column and row annotation tracks. For each track, if no corresponding colour is specified in argument annColors, a palette or a ramp is automatically computed and named after the track's name.

main — Main title as a character string or a grob.

`distfun` default distance measure used in clustering rows and columns. Possible values are:

- all the distance methods supported by `dist` (e.g. "euclidean" or "maximum").
- all correlation methods supported by `cor`, such as `"pearson"` or `"spearman"`. The pairwise distances between rows/columns are then computed as `d <- dist(1 - cor(..., method = distfun))`.
  One may as well use the string "correlation" which is an alias for "pearson".
- an object of class `dist` such as returned by `dist` or `as.dist`.

`hclustfun` default clustering method used to cluster rows and columns. Possible values are:

- a method name (a character string) supported by `hclust` (e.g. `'average'`).
- an object of class `hclust` such as returned by `hclust`
- a dendrogram

Details

IMPORTANT: although they essentially have the same set of arguments, their order sometimes differ between them, as well as from `aheatmap`. We therefore strongly recommend to use fully named arguments when calling these functions.

`basimap` default values for the following arguments of `aheatmap`:

- the color palette;
- the scaling specification, which by default scales each row separately so that they sum up to one (`scale='r1'`);
- the column ordering which is disabled;
- allowing for passing feature extraction methods in argument `subsetRow`, that are passed to `extractFeatures`. See argument description here and therein.
- the addition of a default named annotation track, that shows the dominant basis component for each row (i.e. each feature).
  This track is specified in argument `tracks` (see its argument description). By default, a matching column annotation track is also displayed, but may be disabled using `tracks=':basis'`.
- a suitable title and extra information like the fitting algorithm, when `object` is a fitted NMF model.

`coefmap` redefines default values for the following arguments of `aheatmap`:

- the color palette;
- the scaling specification, which by default scales each column separately so that they sum up to one (`scale='c1'`);
- the row ordering which is disabled;
- the addition of a default annotation track, that shows the most contributing basis component for each column (i.e. each sample).
  This track is specified in argument `tracks` (see its argument description). By default, a matching row annotation track is also displayed, but can be disabled using `tracks='basis:'`.

- a suitable title and extra information like the fitting algorithm, when `object` is a fitted NMF model.

`consensusmap` redefines default values for the following arguments of `aheatmap`:

- the colour palette;
- the column ordering which is set equal to the row ordering, since a consensus matrix is symmetric;
- the distance and linkage methods used to order the rows (and columns). The default is to use 1 minus the consensus matrix itself as distance, and average linkage.
- the addition of two special named annotation tracks, `'basis:'` and `'consensus:'`, that show, for each column (i.e. each sample), the dominant basis component in the best fit and the hierarchical clustering of the consensus matrix respectively (using 1-consensus as distance and average linkage).

  These tracks are specified in argument `tracks`, which behaves as in `basismap`.
- a suitable title and extra information like the type of NMF model or the fitting algorithm, when `object` is a fitted NMF model.

Methods

basismap `signature(object = "NMF")`: Plots a heatmap of the basis matrix of the NMF model `object`. This method also works for fitted NMF models (i.e. `NMFfit` objects).

basismap `signature(object = "NMFfitX")`: Plots a heatmap of the basis matrix of the best fit in `object`.

coefmap `signature(object = "NMF")`: The default method for NMF objects has special default values for some arguments of `aheatmap` (see argument description).

coefmap `signature(object = "NMFfitX")`: Plots a heatmap of the coefficient matrix of the best fit in `object`.

This method adds:

- an extra special column annotation track for multi-run NMF fits, `'consensus:'`, that shows the consensus cluster associated to each sample.
- a column sorting schema `'consensus'` that can be passed to argument `Colv` and orders the columns using the hierarchical clustering of the consensus matrix with average linkage, as returned by `consensushc(object)`. This is also the ordering that is used by default for the heatmap of the consensus matrix as ploted by `consensusmap`.

consensusmap `signature(object = "NMFfitX")`: Plots a heatmap of the consensus matrix obtained when fitting an NMF model with multiple runs.

consensusmap `signature(object = "NMF")`: Plots a heatmap of the connectivity matrix of an NMF model.

consensusmap `signature(object = "matrix")`: Main method that redefines default values for arguments of `aheatmap`.

Examples

```
#----------
# heatmap-NMF
#----------
## More examples are provided in demo `heatmaps`
## Not run:
demo(heatmaps)

## End(Not run)
##

# random data with underlying NMF model
v <- syntheticNMF(20, 3, 10)
# estimate a model
x <- nmf(v, 3)

#----------
# basismap
#----------
# show basis matrix
basismap(x)
## Not run:
# without the default annotation tracks
basismap(x, tracks=NA)

## End(Not run)

#----------
# coefmap
#----------
# coefficient matrix
coefmap(x)
## Not run:
# without the default annotation tracks
coefmap(x, tracks=NA)

## End(Not run)

#----------
# consensusmap
#----------
## Not run:
res <- nmf(x, 3, nrun=3)
consensusmap(res)

## End(Not run)
```

---

ibterms *Fixed Terms in NMF Models*

Description

Formula-based NMF models may contain fixed basis and/or coefficient terms. The functions documented here provide access to these data, which are read-only and defined when the model object is instantiated (e.g., see `nmfModel,formula-method`).

`ibterms`, `icterms` and `iterms` respectively return the indexes of the fixed basis terms, the fixed coefficient terms and all fixed terms, within the basis and/or coefficient matrix of an NMF model.

`nterms`, `nbterms`, and `ncterms` return, respectively, the number of all fixed terms, fixed basis terms and fixed coefficient terms in an NMF model. In particular: i.e. `nterms(object)` = `nbterms(object)` + `ncterms(object)`.

`bterms` and `cterms` return, respectively, the primary data for fixed basis and coefficient terms in an NMF model – as stored in slots `bterms` and `cterms` . These are factors or numeric vectors which define fixed basis components, e.g., used for defining separate offsets for different *a priori* groups of samples, or to incorporate/correct for some known covariate.

`ibasis` and `icoef` return, respectively, the indexes of all latent basis vectors and estimated coefficients within the basis or coefficient matrix of an NMF model.

Usage

```
ibterms(object, ...)

icterms(object, ...)

iterms(object, ...)

nterms(object)

nbterms(object)

ncterms(object)

bterms(object)

cterms(object)

ibasis(object, ...)

icoef(object, ...)
```

Arguments

`object`	NMF object
`...`	extra parameters to allow extension (currently not used)

Methods

ibterms `signature(object = "NMF")`: Default pure virtual method that ensure a method is defined for concrete NMF model classes.

ibterms `signature(object = "NMFstd")`: Method for standard NMF models, which returns the integer vector that is stored in slot `ibterms` when a formula-based NMF model is instantiated.

ibterms `signature(object = "NMFfit")`: Method for single NMF fit objects, which returns the indexes of fixed basis terms from the fitted model.

ibterms `signature(object = "NMFfitX")`: Method for multiple NMF fit objects, which returns the indexes of fixed basis terms from the best fitted model.

icterms `signature(object = "NMF")`: Default pure virtual method that ensure a method is defined for concrete NMF model classes.

icterms `signature(object = "NMFstd")`: Method for standard NMF models, which returns the integer vector that is stored in slot `icterms` when a formula-based NMF model is instantiated.

icterms `signature(object = "NMFfit")`: Method for single NMF fit objects, which returns the indexes of fixed coefficient terms from the fitted model.

`is.nmf` *Testing NMF Objects*

Description

The functions documented here tests different characteristics of NMF objects.

`is.nmf` tests if an object is an NMF model or a class that extends the class NMF.

`hasBasis` tests whether an objects contains a basis matrix – returned by a suitable method `basis` – with at least one row.

`hasBasis` tests whether an objects contains a coefficient matrix – returned by a suitable method `coef` – with at least one column.

`is.partial.nmf` tests whether an NMF model object contains either an empty basis or coefficient matrix. It is a shorcut for `!hasCoef(x) || !hasBasis(x)`.

Usage

```
is.nmf(x)

is.empty.nmf(x, ...)

hasBasis(x)

hasCoef(x)

is.partial.nmf(x)

isNMFfit(object, recursive = TRUE)
```

Arguments

x	an R object. See section *Details*, for how each function uses this argument.
...	extra parameters to allow extension or passed to subsequent calls
object	any R object.
recursive	if TRUE and object is a plain list then isNMFfit tests each element of the list. Note that the recursive test only applies in the case of lists that are not themselves NMFfit objects, like NMFfitXn objects for which the result of isNMFfit will always be TRUE, although they are list objects (a single logical value).

Details

is.nmf tests if object is the name of a class (if a character string), or inherits from a class, that extends NMF.

is.empty.nmf returns TRUE if the basis and coefficient matrices of x have respectively zero rows and zero columns. It returns FALSE otherwise.

In particular, this means that an empty model can still have a non-zero number of basis components, i.e. a factorization rank that is not null. This happens, for example, in the case of NMF models created calling the factory method nmfModel with a value only for the factorization rank.

isNMFfit checks if object inherits from class NMFfit or NMFfitX, which are the two types of objects returned by the function nmf. If object is a plain list and recursive=TRUE, then the test is performed on each element of the list, and the return value is a logical vector (or a list if object is a list of list) of the same length as object.

Value

isNMFfit returns a logical vector (or a list if object is a list of list) of the same length as object.

Note

The function is.nmf does some extra work with the namespace as this function needs to return correct results even when called in .onLoad. See discussion on r-devel: https://stat.ethz.ch/pipermail/r-devel/2011-June/061357.html

See Also

NMFfit, NMFfitX, NMFfitXn

Examples

```
#----------
# is.nmf
#----------
# test if an object is an NMF model, i.e. that it implements the NMF interface
is.nmf(1:4)
is.nmf( nmfModel(3) )
is.nmf( nmf(rmatrix(10, 5), 2) )
```

```
#----------
# is.empty.nmf
#----------
# empty model
is.empty.nmf( nmfModel(3) )
# non empty models
is.empty.nmf( nmfModel(3, 10, 0) )
is.empty.nmf( rnmf(3, 10, 5) )

#----------
# isNMFfit
#----------
## Testing results of fits
# generate a random
V <- rmatrix(20, 10)

# single run -- using very low value for maxIter to speed up the example
res <- nmf(V, 3, maxIter=3L)
isNMFfit(res)

# multiple runs - keeping single fit
resm <- nmf(V, 3, nrun=2, maxIter=3L)
isNMFfit(resm)

# with a list of results
isNMFfit(list(res, resm, 'not a result'))
isNMFfit(list(res, resm, 'not a result'), recursive=FALSE)
```

`isCRANcheck` *Package Check Utils*

Description

`isCRANcheck` **tries** to identify if one is running CRAN-like checks.

Usage

```
isCRANcheck(...)

isCHECK()
```

Arguments

`...`	each argument specifies a set of tests to do using an AND operator. The final result tests if any of the test set is true. Possible values are: `'timing'` Check if the environment variable `_R_CHECK_TIMINGS_` is set, as with the flag `'--timing'` was set. `'cran'` Check if the environment variable `_R_CHECK_CRAN_INCOMING_` is set, as with the flag `'--as-cran'` was set.

Details

Currently `isCRANcheck` returns `TRUE` if the check is run with either environment variable `_R_CHECK_TIMINGS_` (as set by flag `'--timings'`) or `_R_CHECK_CRAN_INCOMINGS_` (as set by flag `'--as-cran'`).

Warning: the checks performed on CRAN check machines are on purpose not always run with such flags, so that users cannot effectively "trick" the checks. As a result, there is no guarantee this function effectively identifies such checks. If really needed for honest reasons, CRAN recommends users rely on custom dedicated environment variables to enable specific tests or examples.

Functions

- `isCHECK`: tries harder to test if running under `R CMD check`. It will definitely identifies check runs for:
  - unit tests that use the unified unit test framework defined by **pkgmaker** (see `utest`);
  - examples that are run with option `R_CHECK_RUNNING_EXAMPLES_ = TRUE`, which is automatically set for man pages generated with a fork of **roxygen2** (see *References*).

  Currently, `isCHECK` checks both CRAN expected flags, the value of environment variable `_R_CHECK_RUNNING_UTESTS_`, and the value of option `R_CHECK_RUNNING_EXAMPLES_`. It will return `TRUE` if any of these environment variables is set to anything not equivalent to `FALSE`, or if the option is `TRUE`. For example, the function `utest` sets it to the name of the package being checked (`_R_CHECK_RUNNING_UTESTS_=<pkgname>`), but unit tests run as part of unit tests vignettes are run with `_R_CHECK_RUNNING_UTESTS_=FALSE`, so that all tests are run and reported when generating them.

References

Adapted from the function `CRAN` in the **fda** package.

https://github.com/renozao/roxygen

Examples

```
isCHECK()
```

---

latex_preamble *LaTeX Utilities for Vignettes*

Description

`latex_preamble` outputs/returns command definition LaTeX commands to be put in the preamble of vignettes.

Usage

```
latex_preamble(
  PACKAGE,
  R = TRUE,
  CRAN = TRUE,
  Bioconductor = TRUE,
  GEO = TRUE,
  ArrayExpress = TRUE,
  biblatex = FALSE,
  only = FALSE,
  file = ""
)

latex_bibliography(PACKAGE, file = "")
```

Arguments

PACKAGE	package name
R	logical that indicate if general R commands should be added (e.g. package names, inline R code format commands)
CRAN	logical that indicate if general CRAN commands should be added (e.g. CRAN package citations)
Bioconductor	logical that indicate if general Bioconductor commands should be added (e.g. Bioc package citations)
GEO	logical that indicate if general GEOmnibus commands should be added (e.g. urls to GEO datasets)
ArrayExpress	logical that indicate if general ArrayExpress commands should be added (e.g. urls to ArrayExpress datasets)
biblatex	logical that indicates if a \bibliography command should be added to include references from the package's REFERENCES.bib file.
only	a logical that indicates if the only the commands whose dedicated argument is not missing should be considered.
file	connection where to print. If NULL the result is returned silently.

Details

Argument PACKAGE is not required for latex_preamble, but must be correctly specified to ensure biblatex=TRUE generates the correct bibliography command.

Functions

- latex_bibliography: latex_bibliography prints or return a LaTeX command that includes a package bibliography file if it exists.

Examples

```
latex_preamble()
latex_preamble(R=TRUE, only=TRUE)
latex_preamble(R=FALSE, CRAN=FALSE, GEO=FALSE)
latex_preamble(GEO=TRUE, only=TRUE)
```

---

match_atrack *Extending Annotation Vectors*

Description

Extends a vector used as an annotation track to match the number of rows and the row names of a given data.

Usage

```
match_atrack(x, data = NULL)
```

Arguments

x	annotation vector
data	reference data

Value

a vector of the same type as x

---

methods-NMF *Registry for NMF Algorithms*

Description

Registry for NMF Algorithms

selectNMFMethod tries to select an appropriate NMF algorithm that is able to fit a given the NMF model.

getNMFMethod retrieves NMF algorithm objects from the registry.

existsNMFMethod tells if an NMF algorithm is registered under the

removeNMFMethod removes an NMF algorithm from the registry.

Usage

```
selectNMFMethod(name, model, load = FALSE, exact = FALSE,
  all = FALSE, quiet = FALSE)

getNMFMethod(...)

existsNMFMethod(name, exact = TRUE)

removeNMFMethod(name, ...)
```

Arguments

name	name of a registered NMF algorithm
model	class name of an NMF model, i.e. a class that inherits from class NMF.
load	a logical that indicates if the selected algorithms should be loaded into NMFStrategy objects
all	a logical that indicates if all algorithms that can fit model should be returned or only the default or first found.
quiet	a logical that indicates if the operation should be performed quietly, without throwing errors or warnings.
...	extra arguments passed to pkgreg_fetch or pkgreg_remove.
exact	a logical that indicates if the access key should be matched exactly (TRUE) or partially (FALSE).

Value

selectNMFMethod returns a character vector or NMFStrategy objects, or NULL if no suitable algorithm was found.

---

nbasis *Dimension of NMF Objects*

Description

The methods dim, nrow, ncol and nbasis return the different dimensions associated with an NMF model.

dim returns all dimensions in a length-3 integer vector: the number of row and columns of the estimated target matrix, as well as the factorization rank (i.e. the number of basis components).

nrow, ncol and nbasis provide separate access to each of these dimensions respectively.

Usage

```
  nbasis(x, ...)

  ## S4 method for signature 'NMF'
dim(x)

  ## S4 method for signature 'NMFfitXn'
dim(x)
```

Arguments

x	an object with suitable basis and coef methods, such as an object that inherit from NMF.
...	extra arguments to allow extension.

Details

The NMF package does not implement specific functions nrow and ncol, but rather the S4 method dim for objects of class NMF. This allows the base methods nrow and ncol to directly work with such objects, to get the number of rows and columns of the target matrix estimated by an NMF model.

The function nbasis is a new S4 generic defined in the package NMF, that returns the number of basis components of an object. Its default method should work for any object, that has a suitable basis method defined for its class.

Value

a single integer value or, for dim, a length-3 integer vector, e.g. c(2000, 30, 3) for an NMF model that fits a 2000 x 30 matrix using 3 basis components.

Methods

dim signature(x = "NMF"): method for NMF objects for the base generic dim. It returns all dimensions in a length-3 integer vector: the number of row and columns of the estimated target matrix, as well as the factorization rank (i.e. the number of basis components).

dim signature(x = "NMFfitXn"): Returns the dimension common to all fits.

Since all fits have the same dimensions, it returns the dimension of the first fit. This method returns NULL if the object is empty.

nbasis signature(x = "ANY"): Default method which returns the number of columns of the basis matrix extracted from x using a suitable method basis, or, if the latter is NULL, the value of attributes 'nbasis'.

For NMF models, this also corresponds to the number of rows in the coefficient matrix.

nbasis signature(x = "NMFfitXn"): Returns the number of basis components common to all fits.

Since all fits have been computed using the same rank, it returns the factorization rank of the first fit. This method returns NULL if the object is empty.

nmf *Running NMF algorithms*

Description

The function nmf is a S4 generic defines the main interface to run NMF algorithms within the framework defined in package NMF. It has many methods that facilitates applying, developing and testing NMF algorithms.

The package vignette vignette('NMF') contains an introduction to the interface, through a sample data analysis.

Usage

```
  nmf(x, rank, method, ...)

  ## S4 method for signature 'matrix,numeric,NULL'
nmf(x, rank, method,
    seed = NULL, model = NULL, ...)

  ## S4 method for signature 'matrix,numeric,list'
nmf(x, rank, method, ...,
    .parameters = list())

  ## S4 method for signature 'matrix,numeric,function'
nmf(x, rank, method,
    seed, model = "NMFstd", ..., name,
    objective = "euclidean", mixed = FALSE)

  ## S4 method for signature 'matrix,NMF,ANY'
nmf(x, rank, method, seed,
    ...)

  ## S4 method for signature 'matrix,NULL,ANY'
nmf(x, rank, method, seed,
    ...)

  ## S4 method for signature 'matrix,matrix,ANY'
nmf(x, rank, method, seed,
    model = list(), ...)

  ## S4 method for signature 'formula,ANY,ANY'
nmf(x, rank, method, ...,
    model = NULL)

  ## S4 method for signature 'matrix,numeric,NMFStrategy'
nmf(x, rank,
```

```
    method, seed = nmf.getOption("default.seed"),
    rng = NULL, nrun = if (length(rank) > 1) 30 else 1,
    model = NULL, .options = list(),
    .pbackend = nmf.getOption("pbackend"),
    .callback = NULL, ...)
```

Arguments

x	target data to fit, i.e. a matrix-like object
rank	specification of the factorization rank. It is usually a single numeric value, but other type of values are possible (e.g. matrix), for which specific methods are implemented. See for example methods `nmf,matrix,matrix,ANY`. If `rank` is a numeric vector with more than one element, e.g. a range of ranks, then `nmf` performs the estimation procedure described in `nmfEstimateRank`.
method	specification of the NMF algorithm. The most common way of specifying the algorithm is to pass the access key (i.e. a character string) of an algorithm stored in the package's dedicated registry, but methods exists that handle other types of values, such as `function` or `list` object. See their descriptions in section *Methods*. If `method` is missing the algorithm to use is obtained from the option `nmf.getOption('default.algorit` unless it can be infer from the type of NMF model to fit, if this later is available from other arguments. Factory fresh default value is 'brunet', which corresponds to the standard NMF algorithm from *Brunet2004* (see section *Algorithms*). Cases where the algorithm is inferred from the call are when an NMF model is passed in arguments `rank` or `seed` (see description for `nmf,matrix,numeric,NULL` in section *Methods*).
...	extra arguments to allow extension of the generic. Arguments that are not used in the chain of internal calls to `nmf` methods are passed to the function that effectively implements the algorithm that fits an NMF model on `x`.
.parameters	list of method-specific parameters. Its elements must have names matching a single method listed in `method`, and be lists of named values that are passed to the corresponding method.
name	name associated with the NMF algorithm implemented by the function `method` [only used when `method` is a function].
objective	specification of the objective function associated with the algorithm implemented by the function `method` [only used when `method` is a function]. It may be either `'euclidean'` or `'KL'` for specifying the euclidean distance (Frobenius norm) or the Kullback-Leibler divergence respectively, or a function with signature `(x="NMF", y="matrix", ...)` that computes the objective value for an NMF model `x` on a target matrix `y`, i.e. the residuals between the target matrix and its NMF estimate. Any extra argument may be specified, e.g. `function(x, y, alpha, beta=2, ...)`.
mixed	a logical that indicates if the algorithm implemented by the function `method` support mixed-sign target matrices, i.e. that may contain negative values [only used when `method` is a function].

`seed` specification of the starting point or seeding method, which will compute a starting point, usually using data from the target matrix in order to provide a good guess.

The seeding method may be specified in the following way:

a `character` string: giving the name of a *registered* seeding method. The corresponding method will be called to compute the starting point.
Available methods can be listed via `nmfSeed()`. See its dedicated documentation for details on each available registered methods (`nmfSeed`).

a `list`: giving the name of a *registered* seeding method and, optionally, extra parameters to pass to it.

a single `numeric`: that is used to seed the random number generator, before generating a random starting point.
Note that when performing multiple runs, the L'Ecuyer's RNG is used in order to produce a sequence of random streams, that is used in way that ensures that parallel computation are fully reproducible.

an object that inherits from `NMF`: it should contain the data of an initialised NMF model, i.e. it must contain valid basis and mixture coefficient matrices, directly usable by the algorithm's workhorse function.

a `function`: that computes the starting point. It must have signature `(object="NMF", target="matrix", ...)` and return an object that inherits from class `NMF`.
It is recommended to use argument `object` as a template for the returned object, by only updating the basis and coefficient matrices, using `basis<-` and `coef<-` respectively.

`rng` rng specification for the run(s). This argument should be used to set the the RNG seed, while still specifying the seeding method argument *seed*.

`model` specification of the type of NMF model to use.

It is used to instantiate the object that inherits from class `NMF`, that will be passed to the seeding method. The following values are supported:

- `NULL`, the default model associated to the NMF algorithm is instantiated and `...` is looked-up for arguments with names that correspond to slots in the model class, which are passed to the function `nmfModel` to instantiate the model. Arguments in `...` that do not correspond to slots are passed to the algorithm.
- a single `character` string, that is the name of the NMF model class to be instantiate. In this case, arguments in `...` are handled in the same way as when `model` is `NULL`.
- a `list` that contains named values that are passed to the function `nmfModel` to instantiate the model. In this case, `...` is not looked-up at all, and passed entirely to the algorithm. This means that all necessary model parameters must be specified in `model`.

Argument/slot conflicts: In the case a parameter of the algorithm has the same name as a model slot, then `model` MUST be a list – possibly empty –, if one wants this parameter to be effectively passed to the algorithm.

If a variable appears in both arguments `model` and `...`, the former will be used to initialise the NMF model, the latter will be passed to the NMF algorithm. See code examples for an illustration of this situation.

`nrun` number of runs to perform. It specifies the number of runs to perform. By default only one run is performed, except if `rank` is a numeric vector with more than one element, in which case a default of 30 runs per value of the rank are performed, allowing the computation of a consensus matrix that is used in selecting the appropriate rank (see `consensus`).

When using a random seeding method, multiple runs are generally required to achieve stability and avoid *bad* local minima.

`.options` this argument is used to set runtime options.

It can be a `list` containing named options with their values, or, in the case only boolean/integer options need to be set, a character string that specifies which options are turned on/off or their value, in a unix-like command line argument way.

The string must be composed of characters that correspond to a given option (see mapping below), and modifiers '+' and '-' that toggle options on and off respectively. E.g. `.options='tv'` will toggle on options `track` and `verbose`, while `.options='t-v'` will toggle on option `track` and toggle off option `verbose`.

Modifiers '+' and '-' apply to all option character found after them: `t-vp+k` means `track=TRUE`, `verbose=parallel=FALSE`, and `keep.all=TRUE`. The default behaviour is to assume that `.options` starts with a '+'.

for options that accept integer values, the value may be appended to the option's character e.g. `'p4'` for asking for 4 processors or `'v3'` for showing verbosity message up to level 3.

The following options are available (the characters after "-" are those to use to encode `.options` as a string):

debug - d Toggle debug mode (default: `FALSE`). Like option `verbose` but with more information displayed.

keep.all - k used when performing multiple runs (`nrun`>1): if `TRUE`, all factorizations are saved and returned (default: `FALSE`). Otherwise only the factorization achieving the minimum residuals is returned.

parallel - p this option is useful on multicore *nix or Mac machine only, when performing multiple runs (`nrun` > 1) (default: `TRUE`). If `TRUE`, the runs are performed using the parallel foreach backend defined in argument `.pbackend`. If this is set to `'mc'` or `'par'` then `nmf` tries to perform the runs using multiple cores with package `doParallel` – which therefore needs to be installed. If equal to an integer, then `nmf` tries to perform the computation on the specified number of processors. When passing options as a string the number is appended to the option's character e.g. `'p4'` for asking for 4 processors.
If `FALSE`, then the computation is performed sequentially using the base function `sapply`.
Unlike option 'P' (capital 'P'), if the computation cannot be performed in parallel, then it will still be carried on sequentially.
IMPORTANT NOTE FOR MAC OS X USERS: The parallel computation is based on the `doMC` and `multicore` packages, so the same care should be taken as stated in the vignette of `doMC`: *"it is not safe to use doMC from R.app on Mac OS X. Instead, you should use doMC from a terminal session, starting R from the command line."*

parallel.required - P Same as p, but an error is thrown if the computation cannot be performed in parallel or with the specified number of processors.

shared.memory - m toggle usage of shared memory (requires the package *synchronicity*). Default is as defined by `nmf.getOption('shared.memory')`.

restore.seed - r deprecated option since version 0.5.99. Will throw a warning if used.

simplifyCB - S toggle simplification of the callback results. Default is `TRUE`

track - t enables error tracking (default: FALSE). If `TRUE`, the returned object's slot `residuals` contains the trajectory of the objective values, which can be retrieved via `residuals(res, track=TRUE)` This tracking functionality is available for all built-in algorithms.

verbose - v Toggle verbosity (default: `FALSE`). If `TRUE`, messages about the configuration and the state of the current run(s) are displayed. The level of verbosity may be specified with an integer value, the greater the level the more messages are displayed. Value `FALSE` means no messages are displayed, while value `TRUE` is equivalent to verbosity level 1.

`.pbackend` specification of the `foreach` parallel backend to register and/or use when running in parallel mode. See options `p` and `P` in argument `.options` for how to enable this mode. Note that any backend that is internally registered is cleaned-up on exit, so that the calling foreach environment should not be affected by a call to `nmf` – except when `.pbackend=NULL`.

Currently it accepts the following values:

'par' use the backend(s) defined by the package `doParallel`;

a numeric value use the specified number of cores with `doParallel` backend;

'seq' use the foreach sequential backend `doSEQ`;

`NULL` use currently registered backend;

`NA` do not compute using a foreach loop – and therefore not in parallel – but rather use a call to standard `sapply`. This is useful for when developing/debugging NMF algorithms, as foreach loop handling may sometime get in the way.
Note that this is equivalent to using `.options='-p'` or `.options='p0'`, but takes precedence over any option specified in `.options`: e.g. `nmf(..., .options='P10', .pbackend=NA)` performs all runs sequentially using `sapply`.
Use `nmf.options(pbackend=NA)` to completely disable foreach/parallel computations for all subsequent `nmf` calls.

'mc' identical to 'par' and defined to ensure backward compatibility.

`.callback` Used when option `keep.all=FALSE` (default). It allows to pass a callback function that is called after each run when performing multiple runs (i.e. with `nrun>1`). This is useful for example if one is also interested in saving summary measures or process the result of each NMF fit before it gets discarded. After each run, the callback function is called with two arguments, the `NMFfit` object that as just been fitted and the run number: `.callback(res, i)`. For convenience, a function that takes only one argument or has signature `(x, ...)` can still be passed in `.callback`. It is wrapped internally into a dummy function with two arguments, only the first of which is passed to the actual callback function (see example with `summary`).

The call is wrapped into a tryCatch so that callback errors do not stop the whole computation (see below).

The results of the different calls to the callback function are stored in a miscellaneous slot accessible using the method `$` for `NMFfit` objects: `res$.callback`. By default `nmf` tries to simplify the list of callback result using `sapply`, unless option `'simplifyCB'` is `FASE`.

If no error occurs `res$.callback` contains the list of values that resulted from the calling the callback function –, ordered as the fits. If any error occurs in one of the callback calls, then the whole computation is **not** stopped, but the error message is stored in `res$.callback`, in place of the result.

See the examples for sample code.

Details

The `nmf` function has multiple methods that compose a very flexible interface allowing to:

- combine NMF algorithms with seeding methods and/or stopping/convergence criterion at runtime;
- perform multiple NMF runs, which are computed in parallel whenever the host machine allows it;
- run multiple algorithms with a common set of parameters, ensuring a consistent environment (notably the RNG settings).

The workhorse method is `nmf,matrix,numeric,NMFStrategy`, which is eventually called by all other methods. The other methods provides convenient ways of specifying the NMF algorithm(s), the factorization rank, or the seed to be used. Some allow to directly run NMF algorithms on different types of objects, such as `data.frame` or *ExpressionSet* objects.

Value

The returned value depends on the run mode:

`Single run:` An object of class `NMFfit`.

`Multiple runs, single method:`
When `nrun > 1` and `method` is not `list`, this method returns an object of class `NMFfitX`.

`Multiple runs, multiple methods:`
When `nrun > 1` and `method` is a `list`, this method returns an object of class `NMFList`.

Methods

nmf `signature(x = "data.frame", rank = "ANY", method = "ANY")`: Fits an NMF model on a `data.frame`.

The target `data.frame` is coerced into a matrix with `as.matrix`.

nmf `signature(x = "matrix", rank = "numeric", method = "NULL")`: Fits an NMF model using an appropriate algorithm when `method` is not supplied.

This method tries to select an appropriate algorithm amongst the NMF algorithms stored in the internal algorithm registry, which contains the type of NMF models each algorithm can fit. This is possible when the type of NMF model to fit is available from argument seed, i.e. if it is an NMF model itself. Otherwise the algorithm to use is obtained from nmf.getOption('default.algorithm').

This method is provided for internal usage, when called from other nmf methods with argument method missing in the top call (e.g. nmf,matrix,numeric,missing).

nmf signature(x = "matrix", rank = "numeric", method = "list"): Fits multiple NMF models on a common matrix using a list of algorithms.

The models are fitted sequentially with nmf using the same options and parameters for all algorithms. In particular, irrespective of the way the computation is seeded, this method ensures that all fits are performed using the same initial RNG settings.

This method returns an object of class NMFList, that is essentially a list containing each fit.

nmf signature(x = "matrix", rank = "numeric", method = "character"): Fits an NMF model on x using an algorithm registered with access key method.

Argument method is partially match against the access keys of all registered algorithms (case insensitive). Available algorithms are listed in section *Algorithms* below or the introduction vignette. A vector of their names may be retrieved via nmfAlgorithm().

nmf signature(x = "matrix", rank = "numeric", method = "function"): Fits an NMF model on x using a custom algorithm defined the function method.

The supplied function must have signature (x=matrix, start=NMF, ...) and return an object that inherits from class NMF. It will be called internally by the workhorse nmf method, with an NMF model to be used as a starting point passed in its argument start.

Extra arguments in ... are passed to method from the top nmf call. Extra arguments that have no default value in the definition of the function method are required to run the algorithm (e.g. see argument alpha of myfun in the examples).

If the algorithm requires a specific type of NMF model, this can be specified in argument model that is handled as in the workhorse nmf method (see description for this argument).

nmf signature(x = "matrix", rank = "NMF", method = "ANY"): Fits an NMF model using the NMF model rank to seed the computation, i.e. as a starting point.

This method is provided for convenience as a shortcut for nmf(x, nbasis(object), method, seed=object, ...) It discards any value passed in argument seed and uses the NMF model passed in rank instead. It throws a warning if argument seed not missing.

If method is missing, this method will call the method nmf,matrix,numeric,NULL, which will infer an algorithm suitable for fitting an NMF model of the class of rank.

nmf signature(x = "matrix", rank = "NULL", method = "ANY"): Fits an NMF model using the NMF model supplied in seed, to seed the computation, i.e. as a starting point.

This method is provided for completeness and is equivalent to nmf(x, seed, method, ...).

nmf signature(x = "matrix", rank = "missing", method = "ANY"): Method defined to ensure the correct dispatch to workhorse methods in case of argument rank is missing.

nmf signature(x = "matrix", rank = "numeric", method = "missing"): Method defined to ensure the correct dispatch to workhorse methods in case of argument method is missing.

nmf signature(x = "matrix", rank = "matrix", method = "ANY"): Fits an NMF model partially seeding the computation with a given matrix passed in rank.

The matrix `rank` is used either as initial value for the basis or mixture coefficient matrix, depending on its dimension.

Currently, such partial NMF model is directly used as a seed, meaning that the remaining part is left uninitialised, which is not accepted by all NMF algorithm. This should change in the future, where the missing part of the model will be drawn from some random distribution.

Amongst built-in algorithms, only 'snmf/l' and 'snmf/r' support partial seeds, with only the coefficient or basis matrix initialised respectively.

nmf `signature(x = "matrix", rank = "data.frame", method = "ANY")`: Shortcut for `nmf(x, as.matrix(rank), method, ...)`.

nmf `signature(x = "formula", rank = "ANY", method = "ANY")`: This method implements the interface for fitting formula-based NMF models. See `nmfModel`.

Argument `rank` target matrix or formula environment. If not missing, `model` must be a `list`, a `data.frame` or an `environment` in which formula variables are searched for.

Optimized C++ vs. plain R

Lee and Seung's multiplicative updates are used by several NMF algorithms. To improve speed and memory usage, a C++ implementation of the specific matrix products is used whenever possible. It directly computes the updates for each entry in the updated matrix, instead of using multiple standard matrix multiplication.

The algorithms that benefit from this optimization are: 'brunet', 'lee', 'nsNMF' and 'offset'. However there still exists plain R versions for these methods, which implement the updates as standard matrix products. These are accessible by adding the prefix '.R#' to their name: '.R#brunet', '.R#lee', '.R#nsNMF' and '.R#offset'.

Algorithms

All algorithms are accessible by their respective access key as listed below. The following algorithms are available:

'brunet' Standard NMF, based on the Kullback-Leibler divergence, from *Brunet et al. (2004)*. It uses simple multiplicative updates from *Lee et al. (2001)*, enhanced to avoid numerical underflow.

Default stopping criterion: invariance of the connectivity matrix (see `nmf.stop.connectivity`).

'lee' Standard NMF based on the Euclidean distance from *Lee et al. (2001)*. It uses simple multiplicative updates.

Default stopping criterion: invariance of the connectivity matrix (see `nmf.stop.connectivity`).

ls-nmf Least-Square NMF from *Wang et al. (2006)*. It uses modified versions of Lee and Seung's multiplicative updates for the Euclidean distance, which incorporates weights on each entry of the target matrix, e.g. to reflect measurement uncertainty.

Default stopping criterion: stationarity of the objective function (see `nmf.stop.stationary`).

'nsNMF' Nonsmooth NMF from *Pascual-Montano et al. (2006)*. It uses a modified version of Lee and Seung's multiplicative updates for the Kullback-Leibler divergence *Lee et al. (2001)*, to fit a extension of the standard NMF model, that includes an intermediate smoothing matrix, meant meant to produce sparser factors.

Default stopping criterion: invariance of the connectivity matrix (see `nmf.stop.connectivity`).

'offset' NMF with offset from *Badea (2008)*. It uses a modified version of Lee and Seung's multiplicative updates for Euclidean distance *Lee et al. (2001)*, to fit an NMF model that includes an intercept, meant to capture a common baseline and shared patterns, in order to produce cleaner basis components.

Default stopping criterion: invariance of the connectivity matrix (see `nmf.stop.connectivity`).

'pe-nmf' Pattern-Expression NMF from *Zhang2008*. It uses multiplicative updates to minimize an objective function based on the Euclidean distance, that is regularized for effective expression of patterns with basis vectors.

Default stopping criterion: stationarity of the objective function (see `nmf.stop.stationary`).

'snmf/r', 'snmf/l' Alternating Least Square (ALS) approach from *Kim et al. (2007)*. It applies the nonnegative least-squares algorithm from *Van Benthem et al. (2004)* (i.e. fast combinatorial nonnegative least-squares for multiple right-hand), to estimate the basis and coefficient matrices alternatively (see `fcnnls`). It minimises an Euclidean-based objective function, that is regularized to favour sparse basis matrices (for 'snmf/l') or sparse coefficient matrices (for 'snmf/r').

Stopping criterion: built-in within the internal workhorse function `nmf_snmf`, based on the KKT optimality conditions.

Seeding methods

The purpose of seeding methods is to compute initial values for the factor matrices in a given NMF model. This initial guess will be used as a starting point by the chosen NMF algorithm.

The seeding method to use in combination with the algorithm can be passed to interface `nmf` through argument `seed`. The seeding seeding methods available in registry are listed by the function `nmfSeed` (see list therein).

Detailed examples of how to specify the seeding method and its parameters can be found in the *Examples* section of this man page and in the package's vignette.

References

Brunet J, Tamayo P, Golub TR and Mesirov JP (2004). "Metagenes and molecular pattern discovery using matrix factorization." _Proceedings of the National Academy of Sciences of the United States of America_, *101*(12), pp. 4164-9. ISSN 0027-8424, <URL: http://dx.doi.org/10.1073/pnas.0308531101>, <URL: http://www.ncbi.nlm.nih.gov/pubmed/15016911>.

Lee DD and Seung H (2001). "Algorithms for non-negative matrix factorization." _Advances in neural information processing systems_. <URL: http://scholar.google.com/scholar?q=intitle:Algorithms+for+non-negative+matrix+factorization>.

Wang G, Kossenkov AV and Ochs MF (2006). "LS-NMF: a modified non-negative matrix factorization algorithm utilizing uncertainty estimates." _BMC bioinformatics_, *7*, pp. 175. ISSN 1471-2105, <URL: http://dx.doi.org/10.1186/1471-2105-7-175>, <URL: http://www.ncbi.nlm.nih.gov/pubmed/16569230>.

Pascual-Montano A, Carazo JM, Kochi K, Lehmann D and Pascual-marqui RD (2006). "Nonsmooth nonnegative matrix factorization (nsNMF)." _IEEE Trans. Pattern Anal. Mach. Intell._, *28*, pp. 403-415.

Badea L (2008). "Extracting gene expression profiles common to colon and pancreatic adenocarcinoma using simultaneous nonnegative matrix factorization." _Pacific Symposium on Biocomputing. Pacific Symposium on Biocomputing_, *290*, pp. 267-78. ISSN 1793-5091, <URL: http://www.ncbi.nlm.nih.gov/pubmed/18229692>.

Kim H and Park H (2007). "Sparse non-negative matrix factorizations via alternating non-negativity-constrained least squares for microarray data analysis." _Bioinformatics (Oxford, England)_, *23*(12), pp. 1495-502. ISSN 1460-2059, <URL: http://dx.doi.org/10.1093/bioinformatics/btm134>, <URL: http://www.ncbi.nlm.nih.gov/pubmed/17483501>.

Van Benthem M and Keenan MR (2004). "Fast algorithm for the solution of large-scale non-negativity-constrained least squares problems." _Journal of Chemometrics_, *18*(10), pp. 441-450. ISSN 0886-9383, <URL: http://dx.doi.org/10.1002/cem.889>, <URL: http://doi.wiley.com/10.1002/cem.889>.

See Also

nmfAlgorithm

Examples

```
# Only basic calls are presented in this manpage.
# Many more examples are provided in the demo file nmf.R
## Not run:
demo('nmf')

## End(Not run)

# random data
x <- rmatrix(20,10)

# run default algorithm with rank 2
res <- nmf(x, 2)

# specify the algorithm
res <- nmf(x, 2, 'lee')

# get verbose message on what is going on
res <- nmf(x, 2, .options='v')
## Not run:
# more messages
res <- nmf(x, 2, .options='v2')
# even more
res <- nmf(x, 2, .options='v3')
# and so on ...

## End(Not run)
```

NMF-class *Generic Interface for Nonnegative Matrix Factorisation Models*

Description

The class NMF is a *virtual class* that defines a common interface to handle Nonnegative Matrix Factorization models (NMF models) in a generic way. Provided a minimum set of generic methods

is implemented by concrete model classes, these benefit from a whole set of functions and utilities to perform common computations and tasks in the context of Nonnegative Matrix Factorization.

The function `misc` provides access to miscellaneous data members stored in slot `misc` (as a `list`), which allow extensions of NMF models to be implemented, without defining a new S4 class.

Usage

```
  misc(object, ...)

  ## S4 method for signature 'NMF'
x$name

  ## S4 replacement method for signature 'NMF'
x$name<-value

  ## S4 method for signature 'NMF'
.DollarNames(x, pattern = "")
```

Arguments

`object`	an object that inherit from class `NMF`
`...`	extra arguments (not used)
`x`	object from which to extract element(s) or in which to replace element(s).
`name`	A literal character string or a name (possibly backtick quoted). For extraction, this is normally (see under 'Environments') partially matched to the `names` of the object.
`value`	typically an array-like R object of a similar class as `x`.
`pattern`	A regular expression. Only matching names are returned.

Details

Class `NMF` makes it easy to develop new models that integrate well into the general framework implemented by the *NMF* package.

Following a few simple guidelines, new types of NMF models benefit from all the functionalities available for the built-in NMF models – that derive themselves from class `NMF`. See section *Implementing NMF models* below.

See `NMFstd`, and references and links therein for details on the built-in implementations of the standard NMF model and its extensions.

Slots

misc A list that is used internally to temporarily store algorithm parameters during the computation.

Methods

[`signature(x = "NMF")`: This method provides a convenient way of sub-setting objects of class `NMF`, using a matrix-like syntax.

It allows to consistently subset one or both matrix factors in the NMF model, as well as retrieving part of the basis components or part of the mixture coefficients with a reduced amount of code.

See `[,NMF-method` for more details.

$ `signature(x = "NMF")`: shortcut for `x@misc[[name, exact=TRUE]]` respectively.

$ `signature(x = "NMF")`: shortcut for `x@misc[[name, exact=TRUE]]` respectively.

$<- `signature(x = "NMF")`: shortcut for `x@misc[[name]] <- value`

$<- `signature(x = "NMF")`: shortcut for `x@misc[[name]] <- value`

.basis `signature(object = "NMF")`: Pure virtual method for objects of class `NMF`, that should be overloaded by sub-classes, and throws an error if called.

.basis<- `signature(object = "NMF", value = "matrix")`: Pure virtual method for objects of class `NMF`, that should be overloaded by sub-classes, and throws an error if called.

basis<- `signature(object = "NMF")`: Default methods that calls `.basis<-` and check the validity of the updated object.

basiscor `signature(x = "NMF", y = "matrix")`: Computes the correlations between the basis vectors of `x` and the columns of `y`.

basiscor `signature(x = "NMF", y = "NMF")`: Computes the correlations between the basis vectors of `x` and `y`.

basiscor `signature(x = "NMF", y = "missing")`: Computes the correlations between the basis vectors of `x`.

basismap `signature(object = "NMF")`: Plots a heatmap of the basis matrix of the NMF model `object`. This method also works for fitted NMF models (i.e. `NMFfit` objects).

c `signature(x = "NMF")`: Binds compatible matrices and NMF models together.

.coef `signature(object = "NMF")`: Pure virtual method for objects of class `NMF`, that should be overloaded by sub-classes, and throws an error if called.

.coef<- `signature(object = "NMF", value = "matrix")`: Pure virtual method for objects of class `NMF`, that should be overloaded by sub-classes, and throws an error if called.

coef<- `signature(object = "NMF")`: Default methods that calls `.coef<-` and check the validity of the updated object.

coefficients `signature(object = "NMF")`: Alias to `coef,NMF`, therefore also pure virtual.

coefmap `signature(object = "NMF")`: The default method for NMF objects has special default values for some arguments of `aheatmap` (see argument description).

connectivity `signature(object = "NMF")`: Computes the connectivity matrix for an NMF model, for which cluster membership is given by the most contributing basis component in each sample. See `predict,NMF-method`.

consensus `signature(object = "NMF")`: This method is provided for completeness and is identical to `connectivity`, and returns the connectivity matrix, which, in the case of a single NMF model, is also the consensus matrix.

consensushc `signature(object = "NMF")`: Compute the hierarchical clustering on the connectivity matrix of `object`.

consensusmap `signature(object = "NMF")`: Plots a heatmap of the connectivity matrix of an NMF model.

deviance `signature(object = "NMF")`: Computes the distance between a matrix and the estimate of an `NMF` model.

dim `signature(x = "NMF")`: method for NMF objects for the base generic `dim`. It returns all dimensions in a length-3 integer vector: the number of row and columns of the estimated target matrix, as well as the factorization rank (i.e. the number of basis components).

dimnames `signature(x = "NMF")`: Returns the dimension names of the NMF model `x`.

It returns either NULL if no dimnames are set on the object, or a 3-length list containing the row names of the basis matrix, the column names of the mixture coefficient matrix, and the column names of the basis matrix (i.e. the names of the basis components).

dimnames<- `signature(x = "NMF")`: sets the dimension names of the NMF model `x`.

`value` can be `NULL` which resets all dimension names, or a 1, 2 or 3-length list providing names at least for the rows of the basis matrix.

See `dimnames` for more details.

.DollarNames `signature(x = "NMF")`: Auto-completion for `NMF` objects

.DollarNames `signature(x = "NMF")`: Auto-completion for `NMF` objects

extractFeatures `signature(object = "NMF")`: Select basis-specific features from an NMF model, by applying the method `extractFeatures,matrix` to its basis matrix.

featureScore `signature(object = "NMF")`: Computes feature scores on the basis matrix of an NMF model.

fitted `signature(object = "NMF")`: Pure virtual method for objects of class `NMF`, that should be overloaded by sub-classes, and throws an error if called.

ibterms `signature(object = "NMF")`: Default pure virtual method that ensure a method is defined for concrete NMF model classes.

icterms `signature(object = "NMF")`: Default pure virtual method that ensure a method is defined for concrete NMF model classes.

loadings `signature(x = "NMF")`: Method loadings for NMF Models

The method `loadings` is identical to `basis`, but do not accept any extra argument.

See `loadings,NMF-method` for more details.

metaHeatmap `signature(object = "NMF")`: Deprecated method that is substituted by `coefmap` and `basismap`.

nmf.equal `signature(x = "NMF", y = "NMF")`: Compares two NMF models.

Arguments in `...` are used only when `identical=FALSE` and are passed to `all.equal`.

nmf.equal `signature(x = "NMF", y = "NMFfit")`: Compares two NMF models when at least one comes from a NMFfit object, i.e. an object returned by a single run of `nmf`.

nmf.equal `signature(x = "NMF", y = "NMFfitX")`: Compares two NMF models when at least one comes from multiple NMF runs.

nneg `signature(object = "NMF")`: Apply `nneg` to the basis matrix of an `NMF` object (i.e. `basis(object)`). All extra arguments in `...` are passed to the method `nneg,matrix`.

predict `signature(object = "NMF")`: Default method for NMF models

profcor `signature(x = "NMF", y = "matrix")`: Computes the correlations between the basis profiles of `x` and the rows of `y`.

profcor `signature(x = "NMF", y = "NMF")`: Computes the correlations between the basis profiles of `x` and `y`.

profcor `signature(x = "NMF", y = "missing")`: Computes the correlations between the basis profiles of `x`.

rmatrix `signature(x = "NMF")`: Returns the target matrix estimate of the NMF model `x`, perturbated by adding a random matrix generated using the default method of `rmatrix`: it is a equivalent to `fitted(x) + rmatrix(fitted(x), ...)`.

This method can be used to generate random target matrices that depart from a known NMF model to a controlled extend. This is useful to test the robustness of NMF algorithms to the presence of certain types of noise in the data.

rnmf `signature(x = "NMF", target = "numeric")`: Generates a random NMF model of the same class and rank as another NMF model.

This is the workhorse method that is eventually called by all other methods. It generates an NMF model of the same class and rank as `x`, compatible with the dimensions specified in `target`, that can be a single or 2-length numeric vector, to specify a square or rectangular target matrix respectively.

See `rnmf,NMF,numeric-method` for more details.

rnmf `signature(x = "NMF", target = "missing")`: Generates a random NMF model of the same dimension as another NMF model.

It is a shortcut for `rnmf(x, nrow(x), ncol(x), ...)`, which returns a random NMF model of the same class and dimensions as `x`.

rposneg `signature(object = "NMF")`: Apply `rposneg` to the basis matrix of an `NMF` object.

show `signature(object = "NMF")`: Show method for objects of class `NMF`

sparseness `signature(x = "NMF")`: Compute the sparseness of an object of class `NMF`, as the sparseness of the basis and coefficient matrices computed separately.

It returns the two values in a numeric vector with names 'basis' and 'coef'.

summary `signature(object = "NMF")`: Computes summary measures for a single NMF model.

The following measures are computed:

See `summary,NMF-method` for more details.

Implementing NMF models

The class `NMF` only defines a basic data/low-level interface for NMF models, as a collection of generic methods, responsible with data handling, upon which relies a comprehensive set of functions, composing a rich higher-level interface.

Actual NMF models are defined as sub-classes that inherits from class `NMF`, and implement the management of data storage, providing definitions for the interface's pure virtual methods.

The minimum requirement to define a new NMF model that integrates into the framework of the *NMF* package are the followings:

- Define a class that inherits from class `NMF` and implements the new model, say class `myNMF`.

- Implement the following S4 methods for the new class myNMF:

  fitted `signature(object = "myNMF", value = "matrix")`: Must return the estimated target matrix as fitted by the NMF model `object`.

  basis `signature(object = "myNMF")`: Must return the basis matrix(e.g. the first matrix factor in the standard NMF model).

  basis<- `signature(object = "myNMF", value = "matrix")`: Must return `object` with the basis matrix set to `value`.

  coef `signature(object = "myNMF")`: Must return the matrix of mixture coefficients (e.g. the second matrix factor in the standard NMF model).

  coef<- `signature(object = "myNMF", value = "matrix")`: Must return `object` with the matrix of mixture coefficients set to `value`.

  The *NMF* package provides "pure virtual" definitions of these methods for class `NMF` (i.e. with signatures `(object='NMF', ...)` and `(object='NMF', value='matrix')`) that throw an error if called, so as to force their definition for model classes.

- Optionally, implement method `rnmf`(signature(x="myNMF", target="ANY")). This method should call `callNextMethod(x=x, target=target, ...)` and fill the returned NMF model with its specific data suitable random values.

For concrete examples of NMF models implementations, see class `NMFstd` and its extensions (e.g. classes `NMFOffset` or `NMFns`).

Creating NMF objects

Strictly speaking, because class `NMF` is virtual, no object of class `NMF` can be instantiated, only objects from its sub-classes. However, those objects are sometimes shortly referred in the documentation and vignettes as "`NMF` objects" instead of "objects that inherits from class `NMF`".

For built-in models or for models that inherit from the standard model class `NMFstd`, the factory method `nmfModel` enables to easily create valid `NMF` objects in a variety of common situations. See documentation for the the factory method `nmfModel` for more details.

References

Definition of Nonnegative Matrix Factorization in its modern formulation: *Lee et al. (1999)*

Historical first definition and algorithms: *Paatero et al. (1994)*

Lee DD and Seung HS (1999). "Learning the parts of objects by non-negative matrix factorization." _Nature_, *401*(6755), pp. 788-91. ISSN 0028-0836, <URL: http://dx.doi.org/10.1038/44565>, <URL: http://www.ncbi.nlm.nih.gov/pubmed/10548103>.

Paatero P and Tapper U (1994). "Positive matrix factorization: A non-negative factor model with optimal utilization of error estimates of data values." _Environmetrics_, *5*(2), pp. 111-126. <URL: http://dx.doi.org/10.1002/env.3170050203>, <URL: http://www3.interscience.wiley.com/cgi-bin/abstract/113468839/ABSTRACT>.

See Also

Main interface to perform NMF in `nmf-methods`.

Built-in NMF models and factory method in `nmfModel`.

Method seed to set NMF objects with values suitable to start algorithms with.

Other NMF-interface: basis, .basis, .basis<-, basis<-, coef, .coef, .coef<-, coef<-, coefficients, loadings,NMF-method, nmfModel, nmfModels, rnmf, scoef

Examples

```
# show all the NMF models available (i.e. the classes that inherit from class NMF)
nmfModels()
# show all the built-in NMF models available
nmfModels(builtin.only=TRUE)

# class NMF is a virtual class so cannot be instantiated:
try( new('NMF') )

# To instantiate an NMF model, use the factory method nmfModel. see ?nmfModel
nmfModel()
nmfModel(3)
nmfModel(3, model='NMFns')
```

NMF-defunct *Defunct Functions and Classes in the NMF Package*

Description

Defunct Functions and Classes in the NMF Package

Usage

```
metaHeatmap(object, ...)
```

Arguments

object	an R object
...	other arguments

Methods

metaHeatmap signature(object = "matrix"): Defunct method substituted by aheatmap.

metaHeatmap signature(object = "NMF"): Deprecated method that is substituted by coefmap and basismap.

metaHeatmap signature(object = "NMFfitX"): Deprecated method subsituted by consensusmap.

NMF-deprecated *Deprecated Functions in the Package NMF*

Description

Deprecated Functions in the Package NMF

Arguments

object	an R object
...	extra arguments

nmf.equal *Testing Equality of NMF Models*

Description

The function nmf.equal tests if two NMF models are the same, i.e. they contain – almost – identical data: same basis and coefficient matrices, as well as same extra parameters.

Usage

```
  nmf.equal(x, y, ...)

  ## S4 method for signature 'NMF,NMF'
nmf.equal(x, y, identical = TRUE,
    ...)

  ## S4 method for signature 'list,list'
nmf.equal(x, y, ..., all = FALSE,
    vector = FALSE)
```

Arguments

x	an NMF model or an object that is associated with an NMF model, e.g. the result from a fit with nmf.
y	an NMF model or an object that is associated with an NMF model, e.g. the result from a fit with nmf.
identical	a logical that indicates if the comparison should be made using the function identical (TRUE) or all.equal (FALSE). See description for method nmf.equal,NMF,NMF.
...	extra arguments to allow extension, and passed to subsequent calls
all	a logical that indicates if all fits should be compared separately or only the best fits
vector	a logical, only used when all=TRUE, that indicates if all fits must be equal for x and y to be declared equal, or if one wants to return the result of each comparison in a vector.

Details

nmf.equal compares two NMF models, and return TRUE iff they are identical acording to the function identical when identical=TRUE, or equal up to some tolerance acording to the function all.equal. This means that all data contained in the objects are compared, which includes at least the basis and coefficient matrices, as well as the extra parameters stored in slot 'misc'.

If extra arguments are specified in ..., then the comparison is performed using all.equal, irrespective of the value of argument identical.

Methods

nmf.equal signature(x = "NMF", y = "NMF"): Compares two NMF models.

Arguments in ... are used only when identical=FALSE and are passed to all.equal.

nmf.equal signature(x = "NMFfit", y = "NMF"): Compares two NMF models when at least one comes from a NMFfit object, i.e. an object returned by a single run of nmf.

nmf.equal signature(x = "NMF", y = "NMFfit"): Compares two NMF models when at least one comes from a NMFfit object, i.e. an object returned by a single run of nmf.

nmf.equal signature(x = "NMFfit", y = "NMFfit"): Compares two fitted NMF models, i.e. objects returned by single runs of nmf.

nmf.equal signature(x = "NMFfitX", y = "NMF"): Compares two NMF models when at least one comes from multiple NMF runs.

nmf.equal signature(x = "NMF", y = "NMFfitX"): Compares two NMF models when at least one comes from multiple NMF runs.

nmf.equal signature(x = "NMFfitX1", y = "NMFfitX1"): Compares the NMF models fitted by multiple runs, that only kept the best fits.

nmf.equal signature(x = "list", y = "list"): Compares the results of multiple NMF runs.

This method either compare the two best fit, or all fits separately. All extra arguments in ... are passed to each internal call to nmf.equal.

nmf.equal signature(x = "list", y = "missing"): Compare all elements in x to x[[1]].

---

nmfAlgorithm *Listing and Retrieving NMF Algorithms*

Description

nmfAlgorithm lists access keys or retrieves NMF algorithms that are stored in registry. It allows to list

Usage

```
nmfAlgorithm(name = NULL, version = NULL, all = FALSE,
  ...)
```

Arguments

name	Access key. If not missing, it must be a single character string that is partially matched against the available algorithms in the registry. In this case, if all=FALSE (default), then the algorithm is returned as an NMFStrategy object that can be directly passed to nmf. An error is thrown if no matching algorithm is found. If missing or NULL, then access keys of algorithms – that match the criteria version, are returned. This argument is assumed to be regular expression if all=TRUE or version is not NULL.
version	version of the algorithm(s) to retrieve. Currently only value 'R' is supported, which searched for plain R implementations.
all	a logical that indicates if all algorithm keys should be returned, including the ones from alternative algorithm versions (e.g. plain R implementations of algorithms, for which a version based on optimised C updates is used by default).
...	extra arguments passed to getNMFMethod when name is not NULL and all=FALSE. It is not used otherwise.

Value

an NMFStrategy object if name is not NULL and all=FALSE, or a named character vector that contains the access keys of the matching algorithms. The names correspond to the access key of the primary algorithm: e.g. algorithm 'lee' has two registered versions, one plain R ('.R#lee') and the other uses optimised C updates ('lee'), which will all get named 'lee'.

See Also

Other regalgo: canFit

Examples

```
# list all main algorithms
nmfAlgorithm()
# list all versions of algorithms
nmfAlgorithm(all=TRUE)
# list all plain R versions
nmfAlgorithm(version='R')
```

---

nmfAlgorithm.SNMF_R *NMF Algorithm - Sparse NMF via Alternating NNLS*

Description

NMF algorithms proposed by *Kim et al. (2007)* that enforces sparsity constraint on the basis matrix (algorithm 'SNMF/L') or the mixture coefficient matrix (algorithm 'SNMF/R').

Usage

```
nmfAlgorithm.SNMF_R(..., maxIter = 20000L, eta = -1,
  beta = 0.01, bi_conv = c(0, 10), eps_conv = 1e-04)

nmfAlgorithm.SNMF_L(..., maxIter = 20000L, eta = -1,
  beta = 0.01, bi_conv = c(0, 10), eps_conv = 1e-04)
```

Arguments

`maxIter`	maximum number of iterations.
`eta`	parameter to suppress/bound the L2-norm of `W` and in `H` in 'SNMF/R' and 'SNMF/L' respectively. If `eta < 0`, then it is set to the maximum value in the target matrix is used.
`beta`	regularisation parameter for sparsity control, which balances the trade-off between the accuracy of the approximation and the sparseness of `H` and `W` in 'SNMF/R' and 'SNMF/L' respectively. Larger beta generates higher sparseness on `H` (resp. `W`). Too large beta is not recommended.
`bi_conv`	parameter of the biclustering convergence test. It must be a size 2 numeric vector `bi_conv=c(wminchange, iconv)`, with: `wminchange`: the minimal allowance of change in row-clusters. `iconv`: decide convergence if row-clusters (within the allowance of `wminchange`) and column-clusters have not changed for `iconv` convergence checks. Convergence checks are performed every 5 iterations.
`eps_conv`	threshold for the KKT convergence test.
`...`	extra argument not used.

Details

The algorithm 'SNMF/R' solves the following NMF optimization problem on a given target matrix A of dimension $n \times p$:

$$\begin{aligned} & \min_{W,H} \tfrac{1}{2} \left(||A - WH||_F^2 + \eta ||W||_F^2 + \beta (\textstyle\sum_{j=1}^{p} ||H_{\cdot j}||_1^2) \right) \\ s.t. \quad & W \geq 0, H \geq 0 \end{aligned}$$

The algorithm 'SNMF/L' solves a similar problem on the transposed target matrix A, where H and W swap roles, i.e. with sparsity constraints applied to `W`.

References

Kim H and Park H (2007). "Sparse non-negative matrix factorizations via alternating non-negativity-constrained least squares for microarray data analysis." _Bioinformatics (Oxford, England)_, *23*(12), pp. 1495-502. ISSN 1460-2059, <URL: http://dx.doi.org/10.1093/bioinformatics/btm134>, <URL: http://www.ncbi.nlm.nih.gov/pubmed/17483501>.

nmfApply *Apply Function for NMF Objects*

Description

The function nmfApply provides exteneded apply-like functionality for objects of class NMF. It enables to easily apply a function over different margins of NMF models.

Usage

```
nmfApply(X, MARGIN, FUN, ..., simplify = TRUE,
  USE.NAMES = TRUE)
```

Arguments

X	an object that has suitable basis and coef methods, e.g. an NMF model.
MARGIN	a single numeric (integer) value that specifies over which margin(s) the function FUN is applied. See section *Details* for a list of possible values.
FUN	a function to apply over the specified margins.
...	extra arguments passed to FUN
simplify	a logical only used when MARGIN=3, that indicates if sapply should try to simplify result if possible. Since this argument follows '...' its name cannot be abbreviated.
USE.NAMES	a logical only used when MARGIN=3, that indicates if sapply should use the names of the basis components to name the results if present. Since this argument follows '...' its name cannot be abbreviated.

Details

The function FUN is applied via a call to apply or sapply according to the value of argument MARGIN as follows:

MARGIN=1 apply FUN to each *row* of the basis matrix: apply(basis(X), 1L, FUN, ...).

MARGIN=2 apply FUN to each *column* of the coefficient matrix: apply(coef(X), 2L, FUN, ...).

MARGIN=3 apply FUN to each *pair* of associated basis component and basis profile: more or less sapply(seq(nbasis(X)), function(i, ...) FUN(basis(X)[,i], coef(X)[i,], ...), ...). In this case FUN must be have at least two arguments, to which are passed each basis components and basis profiles respectively – as numeric vectors.

MARGIN=4 apply FUN to each *column* of the basis matrix, i.e. to each basis component: apply(basis(X), 2L, FUN, ...).

MARGIN=5 apply FUN to each *row* of the coefficient matrix: apply(coef(X), 1L, FUN, ...).

Value

a vector or a list. See apply and sapply for more details on the output format.

nmfCheck *Checking NMF Algorithm*

Description

nmfCheck enables to quickly check that a given NMF algorithm runs properly, by applying it to some small random data.

Usage

```
nmfCheck(method = NULL, rank = max(ncol(x)/5, 3),
  x = NULL, seed = 1234, ...)
```

Arguments

method	name of the NMF algorithm to be tested.
rank	rank of the factorization
x	target data. If NULL, a random 20 x 10 matrix is generated
seed	specifies a seed or seeding method for the computation.
...	other arguments passed to the call to nmf.

Value

the result of the NMF fit invisibly.

Examples

```
# test default algorithm
nmfCheck()

# test 'lee' algorithm
nmfCheck('lee')
```

nmfEstimateRank *Estimate Rank for NMF Models*

Description

A critical parameter in NMF algorithms is the factorization rank r. It defines the number of basis effects used to approximate the target matrix. Function nmfEstimateRank helps in choosing an optimal rank by implementing simple approaches proposed in the literature.

Note that from version *0.7*, one can equivalently call the function nmf with a range of ranks.

In the plot generated by plot.NMF.rank, each curve represents a summary measure over the range of ranks in the survey. The colours correspond to the type of data to which the measure is related: coefficient matrix, basis component matrix, best fit, or consensus matrix.

Usage

```
nmfEstimateRank(x, range,
  method = nmf.getOption("default.algorithm"), nrun = 30,
  model = NULL, ..., verbose = FALSE, stop = FALSE)

## S3 method for class 'NMF.rank'
plot(x, y = NULL,
  what = c("all", "cophenetic", "rss", "residuals", "dispersion", "evar",
      "sparseness", "sparseness.basis", "sparseness.coef", "silhouette",
      "silhouette.coef", "silhouette.basis", "silhouette.consensus"),
  na.rm = FALSE, xname = "x", yname = "y",
  xlab = "Factorization rank", ylab = "",
  main = "NMF rank survey", ...)
```

Arguments

x	For nmfEstimateRank a target object to be estimated, in one of the format accepted by interface nmf. For plot.NMF.rank an object of class NMF.rank as returned by function nmfEstimateRank.
range	a numeric vector containing the ranks of factorization to try. Note that duplicates are removed and values are sorted in increasing order. The results are notably returned in this order.
method	A single NMF algorithm, in one of the format accepted by the function nmf.
nrun	a numeric giving the number of run to perform for each value in range.
model	model specification passed to each nmf call. In particular, when x is a formula, it is passed to argument data of nmfModel to determine the target matrix – and fixed terms.
verbose	toggle verbosity. This parameter only affects the verbosity of the outer loop over the values in range. To print verbose (resp. debug) messages from each NMF run, one can use .options='v' (resp. .options='d') that will be passed to the function nmf.
stop	logical flag for running the estimation process with fault tolerance. When TRUE, the whole execution will stop if any error is raised. When FALSE (default), the runs that raise an error will be skipped, and the execution will carry on. The summary measures for the runs with errors are set to NA values, and a warning is thrown.
...	For nmfEstimateRank, these are extra parameters passed to interface nmf. Note that the same parameters are used for each value of the rank. See nmf. For plot.NMF.rank, these are extra graphical parameter passed to the standard function plot. See plot.
y	reference object of class NMF.rank, as returned by function nmfEstimateRank. The measures contained in y are used and plotted as a reference. It is typically used to plot results obtained from randomized data. The associated curves are drawn in *red* (and *pink*), while those from x are drawn in *blue* (and *green*).

what	a character vector whose elements partially match one of the following item, which correspond to the measures computed by summary on each – multi-run – NMF result: 'all', 'cophenetic', 'rss', 'residuals', 'dispersion', 'evar', 'silhouette' (and more specific *.coef, *.basis, *.consensus), 'sparseness' (and more specific *.coef, *.basis). It specifies which measure must be plotted (what='all' plots all the measures).
na.rm	single logical that specifies if the rank for which the measures are NA values should be removed from the graph or not (default to FALSE). This is useful when plotting results which include NAs due to error during the estimation process. See argument stop for nmfEstimateRank.
xname, yname	legend labels for the curves corresponding to measures from x and y respectively
xlab	x-axis label
ylab	y-axis label
main	main title

Details

Given a NMF algorithm and the target matrix, a common way of estimating r is to try different values, compute some quality measures of the results, and choose the best value according to this quality criteria. See *Brunet et al. (2004)* and *Hutchins et al. (2008)*.

The function nmfEstimateRank allows to perform this estimation procedure. It performs multiple NMF runs for a range of rank of factorization and, for each, returns a set of quality measures together with the associated consensus matrix.

In order to avoid overfitting, it is recommended to run the same procedure on randomized data. The results on the original and the randomised data may be plotted on the same plots, using argument y.

Value

nmfEstimateRank returns a S3 object (i.e. a list) of class NMF.rank with the following elements:

measures	a data.frame containing the quality measures for each rank of factorizations in range. Each row corresponds to a measure, each column to a rank.
consensus	a list of consensus matrices, indexed by the rank of factorization (as a character string).
fit	a list of the fits, indexed by the rank of factorization (as a character string).

References

Brunet J, Tamayo P, Golub TR and Mesirov JP (2004). "Metagenes and molecular pattern discovery using matrix factorization." _Proceedings of the National Academy of Sciences of the United States of America_, *101*(12), pp. 4164-9. ISSN 0027-8424, <URL: http://dx.doi.org/10.1073/pnas.0308531101>, <URL: http://www.ncbi.nlm.nih.gov/pubmed/15016911>.

Hutchins LN, Murphy SM, Singh P and Graber JH (2008). "Position-dependent motif characterization using non-negative matrix factorization." _Bioinformatics (Oxford, England)_, *24*(23), pp. 2684-90. ISSN 1367-4811, <URL: http://dx.doi.org/10.1093/bioinformatics/btn526>, <URL: http://www.ncbi.nlm.nih.gov/pubmed/18852176>.

Examples

```
if( !isCHECK() ){

set.seed(123456)
n <- 50; r <- 3; m <- 20
V <- syntheticNMF(n, r, m)

# Use a seed that will be set before each first run
res <- nmfEstimateRank(V, seq(2,5), method='brunet', nrun=10, seed=123456)
# or equivalently
res <- nmf(V, seq(2,5), method='brunet', nrun=10, seed=123456)

# plot all the measures
plot(res)
# or only one: e.g. the cophenetic correlation coefficient
plot(res, 'cophenetic')

# run same estimation on randomized data
rV <- randomize(V)
rand <- nmfEstimateRank(rV, seq(2,5), method='brunet', nrun=10, seed=123456)
plot(res, rand)
}
```

NMFfit-class *Base Class for to store Nonnegative Matrix Factorisation results*

Description

Base class to handle the results of general **Nonnegative Matrix Factorisation** algorithms (NMF).

The function `NMFfit` is a factory method for NMFfit objects, that should not need to be called by the user. It is used internally by the functions `nmf` and `seed` to instantiate the starting point of NMF algorithms.

Usage

```
NMFfit(fit = nmfModel(), ..., rng = NULL)
```

Arguments

`fit`	an NMF model
`...`	extra argument used to initialise slots in the instantiating `NMFfit` object.
`rng`	RNG settings specification (typically a suitable value for `.Random.seed`).

Details

It provides a general structure and generic functions to manage the results of NMF algorithms. It contains a slot with the fitted NMF model (see slot `fit`) as well as data about the methods and parameters used to compute the factorization.

The purpose of this class is to handle in a generic way the results of NMF algorithms. Its slot `fit` contains the fitted NMF model as an object of class `NMF`.

Other slots contains data about how the factorization has been computed, such as the algorithm and seeding method, the computation time, the final residuals, etc...

Class `NMFfit` acts as a wrapper class for its slot `fit`. It inherits from interface class `NMF` defined for generic NMF models. Therefore, all the methods defined by this interface can be called directly on objects of class `NMFfit`. The calls are simply dispatched on slot `fit`, i.e. the results are the same as if calling the methods directly on slot `fit`.

Slots

fit An object that inherits from class `NMF`, and contains the fitted NMF model.

NB: class `NMF` is a virtual class. The default class for this slot is `NMFstd`, that implements the standard NMF model.

residuals A `numeric` vector that contains the final residuals or the residuals track between the target matrix and its NMF estimate(s). Default value is `numeric()`.

See method `residuals` for details on accessor methods and main interface `nmf` for details on how to compute NMF with residuals tracking.

method a single `character` string that contains the name of the algorithm used to fit the model. Default value is `''`.

seed a single `character` string that contains the name of the seeding method used to seed the algorithm that fitted the NMF model. Default value is `''`. See `nmf` for more details.

rng an object that contains the RNG settings used for the fit. Currently the settings are stored as an integer vector, the value of `.Random.seed` at the time the object is created. It is initialized by the `initialized` method. See `getRNG` for more details.

distance either a single `"character"` string that contains the name of the built-in objective function, or a `function` that measures the residuals between the target matrix and its NMF estimate. See `objective` and `deviance,NMF-method`.

parameters a `list` that contains the extra parameters – usually specific to the algorithm – that were used to fit the model.

runtime object of class `"proc_time"` that contains various measures of the time spent to fit the model. See `system.time`

options a `list` that contains the options used to compute the object.

extra a `list` that contains extra miscellaneous data for internal usage only. For example it can be used to store extra parameters or temporary data, without the need to explicitly extend the `NMFfit` class. Currently built-in algorithms only use this slot to store the number of iterations performed to fit the object.

Data that need to be easily accessible by the end-user should rather be set using the methods `$<-` that sets elements in the `list` slot `misc` – that is inherited from class `NMF`.

call stored call to the last `nmf` method that generated the object.

Methods

algorithm `signature(object = "NMFfit")`: Returns the name of the algorithm that fitted the NMF model `object`.

.basis `signature(object = "NMFfit")`: Returns the basis matrix from an NMF model fitted with function `nmf`.

It is a shortcut for `.basis(fit(object), ...)`, dispatching the call to the `.basis` method of the actual NMF model.

.basis<- `signature(object = "NMFfit", value = "matrix")`: Sets the the basis matrix of an NMF model fitted with function `nmf`.

It is a shortcut for `.basis(fit(object)) <- value`, dispatching the call to the `.basis<-` method of the actual NMF model. It is not meant to be used by the user, except when developing NMF algorithms, to update the basis matrix of the seed object before returning it.

.coef `signature(object = "NMFfit")`: Returns the the coefficient matrix from an NMF model fitted with function `nmf`.

It is a shortcut for `.coef(fit(object), ...)`, dispatching the call to the `.coef` method of the actual NMF model.

.coef<- `signature(object = "NMFfit", value = "matrix")`: Sets the the coefficient matrix of an NMF model fitted with function `nmf`.

It is a shortcut for `.coef(fit(object)) <- value`, dispatching the call to the `.coef<-` method of the actual NMF model. It is not meant to be used by the user, except when developing NMF algorithms, to update the coefficient matrix in the seed object before returning it.

compare `signature(object = "NMFfit")`: Compare multiple NMF fits passed as arguments.

deviance `signature(object = "NMFfit")`: Returns the deviance of a fitted NMF model.

This method returns the final residual value if the target matrix `y` is not supplied, or the approximation error between the fitted NMF model stored in `object` and `y`. In this case, the computation is performed using the objective function `method` if not missing, or the objective of the algorithm that fitted the model (stored in slot `'distance'`).

See `deviance,NMFfit-method` for more details.

fit `signature(object = "NMFfit")`: Returns the NMF model object stored in slot `'fit'`.

fit<- `signature(object = "NMFfit", value = "NMF")`: Updates the NMF model object stored in slot `'fit'` with a new value.

fitted `signature(object = "NMFfit")`: Computes and return the estimated target matrix from an NMF model fitted with function `nmf`.

It is a shortcut for `fitted(fit(object), ...)`, dispatching the call to the `fitted` method of the actual NMF model.

ibterms `signature(object = "NMFfit")`: Method for single NMF fit objects, which returns the indexes of fixed basis terms from the fitted model.

icterms `signature(object = "NMFfit")`: Method for single NMF fit objects, which returns the indexes of fixed coefficient terms from the fitted model.

icterms `signature(object = "NMFfit")`: Method for multiple NMF fit objects, which returns the indexes of fixed coefficient terms from the best fitted model.

minfit `signature(object = "NMFfit")`: Returns the object its self, since there it is the result of a single NMF run.

modelname `signature(object = "NMFfit")`: Returns the type of a fitted NMF model. It is a shortcut for `modelname(fit(object))`.

niter `signature(object = "NMFfit")`: Returns the number of iteration performed to fit an NMF model, typically with function `nmf`.

Currently this data is stored in slot `'extra'`, but this might change in the future.

niter<- `signature(object = "NMFfit", value = "numeric")`: Sets the number of iteration performed to fit an NMF model.

This function is used internally by the function `nmf`. It is not meant to be called by the user, except when developing new NMF algorithms implemented as single function, to set the number of iterations performed by the algorithm on the seed, before returning it (see `NMFStrategyFunction`).

nmf.equal `signature(x = "NMFfit", y = "NMF")`: Compares two NMF models when at least one comes from a NMFfit object, i.e. an object returned by a single run of `nmf`.

nmf.equal `signature(x = "NMFfit", y = "NMFfit")`: Compares two fitted NMF models, i.e. objects returned by single runs of `nmf`.

NMFfitX `signature(object = "NMFfit")`: Creates an `NMFfitX1` object from a single fit. This is used in `nmf` when only the best fit is kept in memory or on disk.

nrun `signature(object = "NMFfit")`: This method always returns 1, since an `NMFfit` object is obtained from a single NMF run.

objective `signature(object = "NMFfit")`: Returns the objective function associated with the algorithm that computed the fitted NMF model `object`, or the objective value with respect to a given target matrix `y` if it is supplied.

offset `signature(object = "NMFfit")`: Returns the offset from the fitted model.

plot `signature(x = "NMFfit", y = "missing")`: Plots the residual track computed at regular interval during the fit of the NMF model `x`.

residuals `signature(object = "NMFfit")`: Returns the residuals – track – between the target matrix and the NMF fit `object`.

runtime `signature(object = "NMFfit")`: Returns the CPU time required to compute a single NMF fit.

runtime.all `signature(object = "NMFfit")`: Identical to `runtime`, since their is a single fit.

seeding `signature(object = "NMFfit")`: Returns the name of the seeding method that generated the starting point for the NMF algorithm that fitted the NMF model `object`.

show `signature(object = "NMFfit")`: Show method for objects of class `NMFfit`

summary `signature(object = "NMFfit")`: Computes summary measures for a single fit from `nmf`.

This method adds the following measures to the measures computed by the method `summary,NMF`:

See `summary,NMFfit-method` for more details.

Examples

```
# run default NMF algorithm on a random matrix
n <- 50; r <- 3; p <- 20
V <- rmatrix(n, p)
```

```
res <- nmf(V, r)

# result class is NMFfit
class(res)
isNMFfit(res)

# show result
res

# compute summary measures
summary(res, target=V)
```

NMFfitX-class *Virtual Class to Handle Results from Multiple Runs of NMF Algorithms*

Description

This class defines a common interface to handle the results from multiple runs of a single NMF algorithm, performed with the nmf method.

Details

Currently, this interface is implemented by two classes, NMFfitX1 and NMFfitXn, which respectively handle the case where only the best fit is kept, and the case where the list of all the fits is returned.

See nmf for more details on the method arguments.

Slots

runtime.all Object of class proc_time that contains CPU times required to perform all the runs.

Methods

basismap `signature(object = "NMFfitX")`: Plots a heatmap of the basis matrix of the best fit in `object`.

coefmap `signature(object = "NMFfitX")`: Plots a heatmap of the coefficient matrix of the best fit in `object`.

This method adds:

- an extra special column annotation track for multi-run NMF fits, `'consensus:'`, that shows the consensus cluster associated to each sample.
- a column sorting schema `'consensus'` that can be passed to argument `Colv` and orders the columns using the hierarchical clustering of the consensus matrix with average linkage, as returned by `consensushc(object)`. This is also the ordering that is used by default for the heatmap of the consensus matrix as ploted by consensusmap.

consensus `signature(object = "NMFfitX")`: Pure virtual method defined to ensure `consensus` is defined for sub-classes of `NMFfitX`. It throws an error if called.

consensushc `signature(object = "NMFfitX")`: Compute the hierarchical clustering on the consensus matrix of `object`, or on the connectivity matrix of the best fit in `object`.

consensusmap `signature(object = "NMFfitX")`: Plots a heatmap of the consensus matrix obtained when fitting an NMF model with multiple runs.

cophcor `signature(object = "NMFfitX")`: Computes the cophenetic correlation coefficient on the consensus matrix of `object`. All arguments in `...` are passed to the method `cophcor,matrix`.

deviance `signature(object = "NMFfitX")`: Returns the deviance achieved by the best fit object, i.e. the lowest deviance achieved across all NMF runs.

dispersion `signature(object = "NMFfitX")`: Computes the dispersion on the consensus matrix obtained from multiple NMF runs.

fit `signature(object = "NMFfitX")`: Returns the model object that achieves the lowest residual approximation error across all the runs.

It is a pure virtual method defined to ensure `fit` is defined for sub-classes of `NMFfitX`, which throws an error if called.

getRNG1 `signature(object = "NMFfitX")`: Returns the RNG settings used for the first NMF run of multiple NMF runs.

ibterms `signature(object = "NMFfitX")`: Method for multiple NMF fit objects, which returns the indexes of fixed basis terms from the best fitted model.

metaHeatmap `signature(object = "NMFfitX")`: Deprecated method subsituted by `consensusmap`.

minfit `signature(object = "NMFfitX")`: Returns the fit object that achieves the lowest residual approximation error across all the runs.

It is a pure virtual method defined to ensure `minfit` is defined for sub-classes of `NMFfitX`, which throws an error if called.

nmf.equal `signature(x = "NMFfitX", y = "NMF")`: Compares two NMF models when at least one comes from multiple NMF runs.

NMFfitX `signature(object = "NMFfitX")`: Provides a way to aggregate `NMFfitXn` objects into an `NMFfitX1` object.

nrun `signature(object = "NMFfitX")`: Returns the number of NMF runs performed to create `object`.

It is a pure virtual method defined to ensure `nrun` is defined for sub-classes of `NMFfitX`, which throws an error if called.

See `nrun,NMFfitX-method` for more details.

predict `signature(object = "NMFfitX")`: Returns the cluster membership index from an NMF model fitted with multiple runs.

Besides the type of clustering available for any NMF models (`'columns'`, `'rows'`, `'samples'`, `'features'`), this method can return the cluster membership index based on the consensus matrix, computed from the multiple NMF runs.

See `predict,NMFfitX-method` for more details.

residuals `signature(object = "NMFfitX")`: Returns the residuals achieved by the best fit object, i.e. the lowest residual approximation error achieved across all NMF runs.

runtime.all `signature(object = "NMFfitX")`: Returns the CPU time required to compute all the NMF runs. It returns `NULL` if no CPU data is available.

show `signature(object = "NMFfitX")`: Show method for objects of class `NMFfitX`

summary `signature(object = "NMFfitX")`: Computes a set of measures to help evaluate the quality of the *best fit* of the set. The result is similar to the result from the `summary` method of `NMFfit` objects. See `NMF` for details on the computed measures. In addition, the cophenetic correlation (`cophcor`) and `dispersion` coefficients of the consensus matrix are returned, as well as the total CPU time (`runtime.all`).

See Also

Other multipleNMF: `NMFfitX1-class`, `NMFfitXn-class`

Examples

```
# generate a synthetic dataset with known classes
n <- 20; counts <- c(5, 2, 3);
V <- syntheticNMF(n, counts)

# perform multiple runs of one algorithm (default is to keep only best fit)
res <- nmf(V, 3, nrun=3)
res

# plot a heatmap of the consensus matrix
## Not run:  consensusmap(res)
```

`NMFfitX1-class` *Structure for Storing the Best Fit Amongst Multiple NMF Runs*

Description

This class is used to return the result from a multiple run of a single NMF algorithm performed with function `nmf` with the – default – option `keep.all=FALSE` (cf. `nmf`).

Details

It extends both classes `NMFfitX` and `NMFfit`, and stores a the result of the best fit in its `NMFfit` structure.

Beside the best fit, this class allows to hold data about the computation of the multiple runs, such as the number of runs, the CPU time used to perform all the runs, as well as the consensus matrix.

Due to the inheritance from class `NMFfit`, objects of class `NMFfitX1` can be handled exactly as the results of single NMF run – as if only the best run had been performed.

Slots

consensus object of class `matrix` used to store the consensus matrix based on all the runs.

nrun an `integer` that contains the number of runs performed to compute the object.

rng1 an object that contains RNG settings used for the first run. See `getRNG1`.

Methods

consensus `signature(object = "NMFfitX1")`: The result is the matrix stored in slot 'consensus'. This method returns `NULL` if the consensus matrix is empty.

fit `signature(object = "NMFfitX1")`: Returns the model object associated with the best fit, amongst all the runs performed when fitting `object`.

Since `NMFfitX1` objects only hold the best fit, this method simply returns the NMF model fitted by `object` – that is stored in slot 'fit'.

getRNG1 `signature(object = "NMFfitX1")`: Returns the RNG settings used to compute the first of all NMF runs, amongst which `object` was selected as the best fit.

minfit `signature(object = "NMFfitX1")`: Returns the fit object associated with the best fit, amongst all the runs performed when fitting `object`.

Since `NMFfitX1` objects only hold the best fit, this method simply returns `object` coerced into an `NMFfit` object.

nmf.equal `signature(x = "NMFfitX1", y = "NMFfitX1")`: Compares the NMF models fitted by multiple runs, that only kept the best fits.

nrun `signature(object = "NMFfitX1")`: Returns the number of NMF runs performed, amongst which `object` was selected as the best fit.

show `signature(object = "NMFfitX1")`: Show method for objects of class `NMFfitX1`

See Also

Other multipleNMF: `NMFfitX-class`, `NMFfitXn-class`

Examples

```
# generate a synthetic dataset with known classes
n <- 15; counts <- c(5, 2, 3);
V <- syntheticNMF(n, counts, factors = TRUE)

# get the class factor
groups <- V$pData$Group

# perform multiple runs of one algorithm, keeping only the best fit (default)
#i.e.: the implicit nmf options are .options=list(keep.all=FALSE) or .options='-k'
res <- nmf(V[[1]], 3, nrun=2)
res

# compute summary measures
summary(res)
# get more info
summary(res, target=V[[1]], class=groups)

# show computational time
runtime.all(res)

# plot the consensus matrix, as stored (pre-computed) in the object
## Not run:  consensusmap(res, annCol=groups)
```

NMFfitXn-class *Structure for Storing All Fits from Multiple NMF Runs*

Description

This class is used to return the result from a multiple run of a single NMF algorithm performed with function `nmf` with option `keep.all=TRUE` (cf. `nmf`).

Details

It extends both classes `NMFfitX` and `list`, and stores the result of each run (i.e. a `NMFfit` object) in its `list` structure.

IMPORTANT NOTE: This class is designed to be **read-only**, even though all the `list`-methods can be used on its instances. Adding or removing elements would most probably lead to incorrect results in subsequent calls. Capability for concatenating and merging NMF results is for the moment only used internally, and should be included and supported in the next release of the package.

Slots

.Data standard slot that contains the S3 `list` object data. See R documentation on S3/S4 classes for more details (e.g., `setOldClass`).

Methods

algorithm `signature(object = "NMFfitXn")`: Returns the name of the common NMF algorithm used to compute all fits stored in `object`

Since all fits are computed with the same algorithm, this method returns the name of algorithm that computed the first fit. It returns `NULL` if the object is empty.

basis `signature(object = "NMFfitXn")`: Returns the basis matrix of the best fit amongst all the fits stored in `object`. It is a shortcut for `basis(fit(object))`.

coef `signature(object = "NMFfitXn")`: Returns the coefficient matrix of the best fit amongst all the fits stored in `object`. It is a shortcut for `coef(fit(object))`.

compare `signature(object = "NMFfitXn")`: Compares the fits obtained by separate runs of NMF, in a single call to `nmf`.

consensus `signature(object = "NMFfitXn")`: This method returns `NULL` on an empty object. The result is a matrix with several attributes attached, that are used by plotting functions such as `consensusmap` to annotate the plots.

dim `signature(x = "NMFfitXn")`: Returns the dimension common to all fits.

Since all fits have the same dimensions, it returns the dimension of the first fit. This method returns `NULL` if the object is empty.

entropy `signature(x = "NMFfitXn", y = "ANY")`: Computes the best or mean entropy across all NMF fits stored in `x`.

fit `signature(object = "NMFfitXn")`: Returns the best NMF fit object amongst all the fits stored in `object`, i.e. the fit that achieves the lowest estimation residuals.

.getRNG `signature(object = "NMFfitXn")`: Returns the RNG settings used for the best fit.

This method throws an error if the object is empty.

getRNG1 `signature(object = "NMFfitXn")`: Returns the RNG settings used for the first run.

This method throws an error if the object is empty.

minfit `signature(object = "NMFfitXn")`: Returns the best NMF model in the list, i.e. the run that achieved the lower estimation residuals.

The model is selected based on its `deviance` value.

modelname `signature(object = "NMFfitXn")`: Returns the common type NMF model of all fits stored in `object`

Since all fits are from the same NMF model, this method returns the model type of the first fit. It returns `NULL` if the object is empty.

nbasis `signature(x = "NMFfitXn")`: Returns the number of basis components common to all fits.

Since all fits have been computed using the same rank, it returns the factorization rank of the first fit. This method returns `NULL` if the object is empty.

nrun `signature(object = "NMFfitXn")`: Returns the number of runs performed to compute the fits stored in the list (i.e. the length of the list itself).

purity `signature(x = "NMFfitXn", y = "ANY")`: Computes the best or mean purity across all NMF fits stored in `x`.

runtime.all `signature(object = "NMFfitXn")`: If no time data is available from in slot 'runtime.all' and argument `null=TRUE`, then the sequential time as computed by `seqtime` is returned, and a warning is thrown unless `warning=FALSE`.

seeding `signature(object = "NMFfitXn")`: Returns the name of the common seeding method used the computation of all fits stored in `object`

Since all fits are seeded using the same method, this method returns the name of the seeding method used for the first fit. It returns `NULL` if the object is empty.

seqtime `signature(object = "NMFfitXn")`: Returns the CPU time that would be required to sequentially compute all NMF fits stored in `object`.

This method calls the function `runtime` on each fit and sum up the results. It returns `NULL` on an empty object.

show `signature(object = "NMFfitXn")`: Show method for objects of class `NMFfitXn`

See Also

Other multipleNMF: `NMFfitX1-class`, `NMFfitX-class`

Examples

```
# generate a synthetic dataset with known classes
n <- 15; counts <- c(5, 2, 3);
V <- syntheticNMF(n, counts, factors = TRUE)

# get the class factor
groups <- V$pData$Group
```

```
# perform multiple runs of one algorithm, keeping all the fits
res <- nmf(V[[1]], 3, nrun=2, .options='k') # .options=list(keep.all=TRUE) also works
res

summary(res)
# get more info
summary(res, target=V[[1]], class=groups)

# compute/show computational times
runtime.all(res)
seqtime(res)

# plot the consensus matrix, computed on the fly
## Not run:  consensusmap(res, annCol=groups)
```

---

nmfFormals *Showing Arguments of NMF Algorithms*

Description

This function returns the extra arguments that can be passed to a given NMF algorithm in call to nmf.

nmfArgs is a shortcut for args(nmfWrapper(x)), to display the arguments of a given NMF algorithm.

Usage

```
nmfFormals(x, ...)

nmfArgs(x)
```

Arguments

x	algorithm specification
...	extra argument to allow extension

Examples

```
# show arguments of an NMF algorithm
nmfArgs('brunet')
nmfArgs('snmf/r')
```

NMFList-class *Class for Storing Heterogeneous NMF fits*

Description

This class wraps a list of NMF fit objects, which may come from different runs of the function nmf, using different parameters, methods, etc.. These can be either from a single run (NMFfit) or multiple runs (NMFfitX).

Note that its definition/interface is very likely to change in the future.

Methods

algorithm `signature(object = "NMFList")`: Returns the method names used to compute the NMF fits in the list. It returns `NULL` if the list is empty.

runtime `signature(object = "NMFList")`: Returns the CPU time required to compute all NMF fits in the list. It returns `NULL` if the list is empty. If no timing data are available, the sequential time is returned.

seqtime `signature(object = "NMFList")`: Returns the CPU time that would be required to sequentially compute all NMF fits stored in `object`.

This method calls the function `runtime` on each fit and sum up the results. It returns `NULL` on an empty object.

show `signature(object = "NMFList")`: Show method for objects of class `NMFList`

nmfModel *Factory Methods NMF Models*

Description

`nmfModel` is a S4 generic function which provides a convenient way to build NMF models. It implements a unified interface for creating `NMF` objects from any NMF models, which is designed to resolve potential dimensions inconsistencies.

`nmfModels` lists all available NMF models currently defined that can be used to create NMF objects, i.e. – more or less – all S4 classes that inherit from class NMF.

Usage

```
  nmfModel(rank, target = 0L, ...)

  ## S4 method for signature 'numeric,numeric'
nmfModel(rank, target,
    ncol = NULL, model = "NMFstd", W, H, ...,
    force.dim = TRUE, order.basis = TRUE)
```

```
  ## S4 method for signature 'numeric,matrix'
nmfModel(rank, target, ...,
    use.names = TRUE)

  ## S4 method for signature 'formula,ANY'
nmfModel(rank, target, ...,
    data = NULL, no.attrib = FALSE)

  nmfModels(builtin.only = FALSE)
```

Arguments

rank	specification of the target factorization rank (i.e. the number of components).
target	an object that specifies the dimension of the estimated target matrix.
...	extra arguments to allow extension, that are passed down to the workhorse method nmfModel,numeric.numeric, where they are used to initialise slots specific to the instantiating NMF model class.
ncol	a numeric value that specifies the number of columns of the target matrix, fitted the NMF model. It is used only if not missing and when argument target is a single numeric value.
model	the class of the object to be created. It must be a valid class name that inherits from class NMF. Default is the standard NMF model NMFstd.
W	value for the basis matrix. data.frame objects are converted into matrices with as.matrix.
H	value for the mixture coefficient matrix data.frame objects are converted into matrices with as.matrix.
force.dim	logical that indicates whether the method should try lowering the rank or shrinking dimensions of the input matrices to make them compatible
order.basis	logical that indicates whether the basis components should reorder the rows of the mixture coefficient matrix to match the order of the basis components, based on their respective names. It is only used if the basis and coefficient matrices have common unique column and row names respectively.
use.names	a logical that indicates whether the dimension names of the target matrix should be set on the returned NMF model.
data	Optional argument where to look for the variables used in the formula.
no.attrib	logical that indicate if attributes containing data related to the formula should be attached as attributes. If FALSE attributes 'target' and 'formula' contain the target matrix, and a list describing each formula part (response, regressors, etc.).
builtin.only	logical that indicates whether only built-in NMF models, i.e. defined within the NMF package, should be listed.

Details

All nmfModel methods return an object that inherits from class NMF, that is suitable for seeding NMF algorithms via arguments rank or seed of the nmf method, in which case the factorisation rank is implicitly set by the number of basis components in the seeding model (see nmf).

For convenience, shortcut methods and internal conversions for working on `data.frame` objects directly are implemented. However, note that conversion of a `data.frame` into a `matrix` object may take some non-negligible time, for large datasets. If using this method or other NMF-related methods several times, consider converting your data `data.frame` object into a matrix once for good, when first loaded.

Value

an object that inherits from class `NMF`.

a list

Methods

nmfModel `signature(rank = "numeric", target = "numeric")`: Main factory method for NMF models

This method is the workhorse method that is eventually called by all other methods. See section *Main factory method* for more details.

nmfModel `signature(rank = "numeric", target = "missing")`: Creates an empty NMF model of a given rank.

This call is equivalent to `nmfModel(rank, 0L, ...)`, which creates *empty* `NMF` object with a basis and mixture coefficient matrix of dimension 0 x `rank` and `rank` x 0 respectively.

nmfModel `signature(rank = "missing", target = "ANY")`: Creates an empty NMF model of null rank and a given dimension.

This call is equivalent to `nmfModel(0, target, ...)`.

nmfModel `signature(rank = "NULL", target = "ANY")`: Creates an empty NMF model of null rank and given dimension.

This call is equivalent to `nmfModel(0, target, ...)`, and is meant for internal usage only.

nmfModel `signature(rank = "missing", target = "missing")`: Creates an empty NMF model or from existing factors

This method is equivalent to `nmfModel(0, 0, ..., force.dim=FALSE)`. This means that the dimensions of the NMF model will be taken from the optional basis and mixture coefficient arguments `W` and `H`. An error is thrown if their dimensions are not compatible.

Hence, this method may be used to generate an NMF model from existing factor matrices, by providing the named arguments `W` and/or `H`:

`nmfModel(W=w)` or `nmfModel(H=h)` or `nmfModel(W=w, H=h)`

Note that this may be achieved using the more convenient interface is provided by the method `nmfModel,matrix,matrix` (see its dedicated description).

See the description of the appropriate method below.

nmfModel `signature(rank = "numeric", target = "matrix")`: Creates an NMF model compatible with a target matrix.

This call is equivalent to `nmfModel(rank, dim(target), ...)`. That is that the returned NMF object fits a target matrix of the same dimension as `target`.

Only the dimensions of `target` are used to construct the `NMF` object. The matrix slots are filled with `NA` values if these are not specified in arguments `W` and/or `H`. However, dimension names are set on the return NMF model if present in `target` and argument `use.names=TRUE`.

nmfModel `signature(rank = "matrix", target = "matrix")`: Creates an NMF model based on two existing factors.

This method is equivalent to `nmfModel(0, 0, W=rank, H=target..., force.dim=FALSE)`. This allows for a natural shortcut for wrapping existing **compatible** matrices into NMF models: '`nmfModel(w, h)`'

Note that an error is thrown if their dimensions are not compatible.

nmfModel `signature(rank = "data.frame", target = "data.frame")`: Same as `nmfModel('matrix', 'matrix')` but for `data.frame` objects, which are generally produced by `read.delim`-like functions.

The input `data.frame` objects are converted into matrices with `as.matrix`.

nmfModel `signature(rank = "matrix", target = "ANY")`: Creates an NMF model with arguments `rank` and `target` swapped.

This call is equivalent to `nmfModel(rank=target, target=rank, ...)`. This allows to call the `nmfModel` function with arguments `rank` and `target` swapped. It exists for convenience:

- allows typing `nmfModel(V)` instead of `nmfModel(target=V)` to create a model compatible with a given matrix `V` (i.e. of dimension `nrow(V), 0, ncol(V)`)
- one can pass the arguments in any order (the one that comes to the user's mind first) and it still works as expected.

nmfModel `signature(rank = "formula", target = "ANY")`: Build a formula-based NMF model, that can incorporate fixed basis or coefficient terms.

Main factory method

The main factory engine of NMF models is implemented by the method with signature `numeric, numeric`. Other factory methods provide convenient ways of creating NMF models from e.g. a given target matrix or known basis/coef matrices (see section *Other Factory Methods*).

This method creates an object of class `model`, using the extra arguments in `...` to initialise slots that are specific to the given model.

All NMF models implement get/set methods to access the matrix factors (see `basis`), which are called to initialise them from arguments `W` and `H`. These argument names derive from the definition of all built-in models that inherit derive from class `NMFstd`, which has two slots, W and H, to hold the two factors – following the notations used in *Lee et al. (1999)*.

If argument `target` is missing, the method creates a standard NMF model of dimension 0x`rank`x0. That is that the basis and mixture coefficient matrices, W and H, have dimension 0x`rank` and `rank`x0 respectively.

If target dimensions are also provided in argument `target` as a 2-length vector, then the method creates an `NMF` object compatible to fit a target matrix of dimension `target[1]`x`target[2]`. That is that the basis and mixture coefficient matrices, W and H, have dimension `target[1]`x`rank` and `rank`x`target[2]` respectively. The target dimensions can also be specified using both arguments `target` and `ncol` to define the number of rows and the number of columns of the target matrix respectively. If no other argument is provided, these matrices are filled with NAs.

If arguments `W` and/or `H` are provided, the method creates a NMF model where the basis and mixture coefficient matrices, W and H, are initialised using the values of `W` and/or `H`.

The dimensions given by `target`, `W` and `H`, must be compatible. However if `force.dim=TRUE`, the method will reduce the dimensions to the achieve dimension compatibility whenever possible.

When W and H are both provided, the NMF object created is suitable to seed a NMF algorithm in a call to the nmf method. Note that in this case the factorisation rank is implicitly set by the number of basis components in the seed.

References

Lee DD and Seung HS (1999). "Learning the parts of objects by non-negative matrix factorization." _Nature_, *401*(6755), pp. 788-91. ISSN 0028-0836, <URL: http://dx.doi.org/10.1038/44565>, <URL: http://www.ncbi.nlm.nih.gov/pubmed/10548103>.

See Also

is.empty.nmf

Other NMF-interface: basis, .basis, .basis<-, basis<-, coef, .coef, .coef<-, coef<-, coefficients, .DollarNames,NMF-method, loadings,NMF-method, misc, NMF-class, $<-,NMF-method, $,NMF-method, rnmf, scoef

Examples

```
#----------
# nmfModel,numeric,numeric-method
#----------
# data
n <- 20; r <- 3; p <- 10
V <- rmatrix(n, p) # some target matrix

# create a r-ranked NMF model with a given target dimensions n x p as a 2-length vector
nmfModel(r, c(n,p)) # directly
nmfModel(r, dim(V)) # or from an existing matrix <=> nmfModel(r, V)
# or alternatively passing each dimension separately
nmfModel(r, n, p)

# trying to create a NMF object based on incompatible matrices generates an error
w <- rmatrix(n, r)
h <- rmatrix(r+1, p)
try( new('NMFstd', W=w, H=h) )
try( nmfModel(w, h) )
try( nmfModel(r+1, W=w, H=h) )
# The factory method can be force the model to match some target dimensions
# but warnings are thrown
nmfModel(r, W=w, H=h)
nmfModel(r, n-1, W=w, H=h)

#----------
# nmfModel,numeric,missing-method
#----------
## Empty model of given rank
nmfModel(3)

#----------
# nmfModel,missing,ANY-method
```

```
#----------
nmfModel(target=10) #square
nmfModel(target=c(10, 5))

#----------
# nmfModel,missing,missing-method
#----------
# Build an empty NMF model
nmfModel()

# create a NMF object based on one random matrix: the missing matrix is deduced
# Note this only works when using factory method NMF
n <- 50; r <- 3;
w <- rmatrix(n, r)
nmfModel(W=w)

# create a NMF object based on random (compatible) matrices
p <- 20
h <- rmatrix(r, p)
nmfModel(H=h)

# specifies two compatible matrices
nmfModel(W=w, H=h)
# error if not compatible
try( nmfModel(W=w, H=h[-1,]) )

#----------
# nmfModel,numeric,matrix-method
#----------
# create a r-ranked NMF model compatible with a given target matrix
obj <- nmfModel(r, V)
all(is.na(basis(obj)))

#----------
# nmfModel,matrix,matrix-method
#----------
## From two existing factors

# allows a convenient call without argument names
w <- rmatrix(n, 3); h <- rmatrix(3, p)
nmfModel(w, h)

# Specify the type of NMF model (e.g. 'NMFns' for non-smooth NMF)
mod <- nmfModel(w, h, model='NMFns')
mod

# One can use such an NMF model as a seed when fitting a target matrix with nmf()
V <- rmatrix(mod)
res <- nmf(V, mod)
nmf.equal(res, nmf(V, mod))

# NB: when called only with such a seed, the rank and the NMF algorithm
# are selected based on the input NMF model.
```

```
# e.g. here rank was 3 and the algorithm "nsNMF" is used, because it is the default
# algorithm to fit "NMFns" models (See ?nmf).

#----------
# nmfModel,matrix,ANY-method
#----------
## swapped arguments `rank` and `target`
V <- rmatrix(20, 10)
nmfModel(V) # equivalent to nmfModel(target=V)
nmfModel(V, 3) # equivalent to nmfModel(3, V)

#----------
# nmfModel,formula,ANY-method
#----------
# empty 3-rank model
nmfModel(~ 3)

# 3-rank model that fits a given data matrix
x <- rmatrix(20,10)
nmfModel(x ~ 3)

# add fixed coefficient term defined by a factor
gr <- gl(2, 5)
nmfModel(x ~ 3 + gr)

# add fixed coefficient term defined by a numeric covariate
nmfModel(x ~ 3 + gr + b, data=list(b=runif(10)))

# 3-rank model that fits a given ExpressionSet (with fixed coef terms)
if(requireNamespace("Biobase", quietly=TRUE)){
e <- Biobase::ExpressionSet(x)
pData(e) <- data.frame(a=runif(10))
nmfModel(e ~ 3 + gr + a) # `a` is looked up in the phenotypic data of x pData(x)
}

#----------
# nmfModels
#----------
# show all the NMF models available (i.e. the classes that inherit from class NMF)
nmfModels()
# show all the built-in NMF models available
nmfModels(builtin.only=TRUE)
```

`NMFns-class` *NMF Model - Nonsmooth Nonnegative Matrix Factorization*

Description

This class implements the *Nonsmooth Nonnegative Matrix Factorization* (nsNMF) model, required by the Nonsmooth NMF algorithm.

The Nonsmooth NMF algorithm is defined by *Pascual-Montano et al. (2006)* as a modification of the standard divergence based NMF algorithm (see section Details and references below). It aims at obtaining sparser factor matrices, by the introduction of a smoothing matrix.

Details

The Nonsmooth NMF algorithm is a modification of the standard divergence based NMF algorithm (see NMF). Given a non-negative $n \times p$ matrix V and a factorization rank r, it fits the following model:

$$V \equiv WS(\theta)H,$$

where:

- W and H are such as in the standard model, i.e. non-negative matrices of dimension $n \times r$ and $r \times p$ respectively;
- S is a $r \times r$ square matrix whose entries depends on an extra parameter $0 \leq \theta \leq 1$ in the following way:

  $$S = (1-\theta)I + \frac{\theta}{r}11^T,$$

  where I is the identity matrix and 1 is a vector of ones.

The interpretation of S as a smoothing matrix can be explained as follows: Let X be a positive, nonzero, vector. Consider the transformed vector $Y = SX$. If $\theta = 0$, then $Y = X$ and no smoothing on X has occurred. However, as $\theta \to 1$, the vector Y tends to the constant vector with all elements almost equal to the average of the elements of X. This is the smoothest possible vector in the sense of non-sparseness because all entries are equal to the same nonzero value, instead of having some values close to zero and others clearly nonzero.

Methods

fitted `signature(object = "NMFns")`: Compute estimate for an NMFns object, according to the Nonsmooth NMF model (cf. NMFns-class).

Extra arguments in `...` are passed to method `smoothing`, and are typically used to pass a value for `theta`, which is used to compute the smoothing matrix instead of the one stored in `object`.

show `signature(object = "NMFns")`: Show method for objects of class `NMFns`

Creating objects from the Class

Object of class `NMFns` can be created using the standard way with operator `new`

However, as for all NMF model classes – that extend class NMF, objects of class `NMFns` should be created using factory method `nmfModel` :

```
new('NMFns')

nmfModel(model='NMFns')

nmfModel(model='NMFns', W=w, theta=0.3
```

See `nmfModel` for more details on how to use the factory method.

Algorithm

The Nonsmooth NMF algorithm uses a modified version of the multiplicative update equations in Lee & Seung's method for Kullback-Leibler divergence minimization. The update equations are modified to take into account the – constant – smoothing matrix. The modification reduces to using matrix WS instead of matrix W in the update of matrix H, and similarly using matrix SH instead of matrix H in the update of matrix W.

After the matrix W has been updated, each of its columns is scaled so that it sums up to 1.

References

Pascual-Montano A, Carazo JM, Kochi K, Lehmann D and Pascual-marqui RD (2006). "Nonsmooth nonnegative matrix factorization (nsNMF)." _IEEE Trans. Pattern Anal. Mach. Intell_, *28*, pp. 403-415.

See Also

Other NMF-model: initialize,NMFOffset-method, NMFOffset-class, NMFstd-class

Examples

```
# create a completely empty NMFns object
new('NMFns')

# create a NMF object based on random (compatible) matrices
n <- 50; r <- 3; p <- 20
w <- rmatrix(n, r)
h <- rmatrix(r, p)
nmfModel(model='NMFns', W=w, H=h)

# apply Nonsmooth NMF algorithm to a random target matrix
V <- rmatrix(n, p)
## Not run: nmf(V, r, 'ns')

# random nonsmooth NMF model
rnmf(3, 10, 5, model='NMFns', theta=0.3)
```

---

nmfObject *Updating NMF Objects*

Description

This function serves to update an objects created with previous versions of the NMF package, which would otherwise be incompatible with the current version, due to changes in their S4 class definition.

Usage

```
nmfObject(object, verbose = FALSE)
```

Arguments

object	an R object created by the NMF package, e.g., an object of class NMF or NMFfit.
verbose	logical to toggle verbose messages.

Details

This function makes use of heuristics to automatically update object slots, which have been borrowed from the BiocGenerics package, the function updateObjectFromSlots in particular.

NMFOffset-class *NMF Model - Nonnegative Matrix Factorization with Offset*

Description

This class implements the *Nonnegative Matrix Factorization with Offset* model, required by the NMF with Offset algorithm.

Usage

```
  ## S4 method for signature 'NMFOffset'
initialize(.Object, ..., offset)
```

Arguments

offset	optional numeric vector used to initialise slot 'offset'.
.Object	An object: see the Details section.
...	data to include in the new object. Named arguments correspond to slots in the class definition. Unnamed arguments must be objects from classes that this class extends.

Details

The NMF with Offset algorithm is defined by *Badea (2008)* as a modification of the euclidean based NMF algorithm from Lee2001 (see section Details and references below). It aims at obtaining 'cleaner' factor matrices, by the introduction of an offset matrix, explicitly modelling a feature specific baseline – constant across samples.

Methods

fitted signature(object = "NMFOffset"): Computes the target matrix estimate for an NMFOffset object.

The estimate is computed as:

$$WH + offset$$

offset signature(object = "NMFOffset"): The function offset returns the offset vector from an NMF model that has an offset, e.g. an NMFOffset model.

rnmf `signature(x = "NMFOffset", target = "numeric")`: Generates a random NMF model with offset, from class `NMFOffset`.

The offset values are drawn from a uniform distribution between 0 and the maximum entry of the basis and coefficient matrices, which are drawn by the next suitable `rnmf` method, which is the workhorse method `rnmf,NMF,numeric`.

show `signature(object = "NMFOffset")`: Show method for objects of class `NMFOffset`

Creating objects from the Class

Object of class `NMFOffset` can be created using the standard way with operator `new`

However, as for all NMF model classes – that extend class `NMF`, objects of class `NMFOffset` should be created using factory method `nmfModel` :

```
new('NMFOffset')
```

```
nmfModel(model='NMFOffset')
```

```
nmfModel(model='NMFOffset', W=w, offset=rep(1, nrow(w)))
```

See `nmfModel` for more details on how to use the factory method.

Initialize method

The initialize method for `NMFOffset` objects tries to correct the initial value passed for slot `offset`, so that it is consistent with the dimensions of the `NMF` model: it will pad the offset vector with NA values to get the length equal to the number of rows in the basis matrix.

References

Badea L (2008). "Extracting gene expression profiles common to colon and pancreatic adenocarcinoma using simultaneous nonnegative matrix factorization." _Pacific Symposium on Biocomputing. Pacific Symposium on Biocomputing_, *290*, pp. 267-78. ISSN 1793-5091, <URL: http://www.ncbi.nlm.nih.gov/pubmed/18229692>.

See Also

Other NMF-model: `NMFns-class`, `NMFstd-class`

Examples

```
# create a completely empty NMF object
new('NMFOffset')

# create a NMF object based on random (compatible) matrices
n <- 50; r <- 3; p <- 20
w <- rmatrix(n, r)
h <- rmatrix(r, p)
nmfModel(model='NMFOffset', W=w, H=h, offset=rep(0.5, nrow(w)))

# apply Nonsmooth NMF algorithm to a random target matrix
V <- rmatrix(n, p)
## Not run: nmf(V, r, 'offset')
```

```
# random NMF model with offset
rnmf(3, 10, 5, model='NMFOffset')
```

---

nmfReport *Run NMF Methods and Generate a Report*

Description

Generates an HTML report from running a set of method on a given target matrix, for a set of factorization ranks.

Usage

```
nmfReport(x, rank, method, colClass = NULL, ...,
  output = NULL, template = NULL)
```

Arguments

x	target matrix
rank	factorization rank
method	list of methods to apply
colClass	reference class to assess accuracy
...	extra paramters passed to nmf
output	output HTML file
template	template Rmd file

Details

The report is based on an .Rmd document 'report.Rmd' stored in the package installation subdirectory scripts/, and is compiled using **knitr**.

At the beginning of the document, a file named 'functions.R' is looked for in the current directory, and sourced if present. This enables the definition of custom NMF methods (see setNMFMethod) or setting global options.

Value

a list with the following elements:

fits	the fit(s) for each method and each value of the rank.
accuracy	a data.frame that contains the summary assessment measures, for each fit.

Examples

```
## Not run:

x <- rmatrix(20, 10)
gr <- gl(2, 5)
nmfReport(x, 2:4, method = list('br', 'lee'), colClass = gr, nrun = 5)


## End(Not run)
```

NMFSeed *NMFSeed is a constructor method that instantiate NMFSeed objects.*

Description

NMFSeed is a constructor method that instantiate NMFSeed objects.

NMF seeding methods are registered via the function setNMFSeed, which stores them as NMFSeed objects in a dedicated registry.

removeNMFSeed removes an NMF seeding method from the registry.

Usage

```
NMFSeed(key, method, ...)

setNMFSeed(..., overwrite = isLoadingNamespace(),
  verbose = TRUE)

removeNMFSeed(name, ...)
```

Arguments

key	access key as a single character string
method	specification of the seeding method, as a function that takes at least the following arguments: **object** uninitialised/empty NMF model, i.e. that it has 0 rows and columns, but has already the rank requested in the call to nmf or seed. **x** target matrix **...** extra arguments
...	arguments passed to NMFSeed and used to initialise slots in the NMFSeed object, or to pkgreg_remove.
name	name of the seeding method.
overwrite	logical that indicates if any existing NMF method with the same name should be overwritten (TRUE) or not (FALSE), in which case an error is thrown.
verbose	a logical that indicates if information about the registration should be printed (TRUE) or not (FALSE).

Methods

NMFSeed `signature(key = "character")`: Default method simply calls `new` with the same arguments.

NMFSeed `signature(key = "NMFSeed")`: Creates an `NMFSeed` based on a template object (Constructor-Copy), in particular it uses the **same** name.

`nmfSeed` *Seeding Strategies for NMF Algorithms*

Description

`nmfSeed` lists and retrieves NMF seeding methods.

`getNMFSeed` is an alias for `nmfSeed`.

`existsNMFSeed` tells if a given seeding method exists in the registry.

Usage

```
nmfSeed(name = NULL, ...)

getNMFSeed(name = NULL, ...)

existsNMFSeed(name, exact = TRUE)
```

Arguments

`name`	access key of a seeding method stored in registry. If missing, `nmfSeed` returns the list of all available seeding methods.
`...`	extra arguments used for internal calls
`exact`	a logical that indicates if the access key should be matched exactly or partially.

Details

Currently the internal registry contains the following seeding methods, which may be specified to the function `nmf` via its argument `seed` using their access keys:

random The entries of each factors are drawn from a uniform distribution over $[0, max(x)]$, where x is the target matrix.

nndsvd Nonnegative Double Singular Value Decomposition.

The basic algorithm contains no randomization and is based on two SVD processes, one approximating the data matrix, the other approximating positive sections of the resulting partial SVD factors utilising an algebraic property of unit rank matrices.

It is well suited to initialise NMF algorithms with sparse factors. Simple practical variants of the algorithm allows to generate dense factors.

Reference: *Boutsidis et al. (2008)*

ica Uses the result of an Independent Component Analysis (ICA) (from the `fastICA` package). Only the positive part of the result are used to initialise the factors.

none Fixed seed.

This method allows the user to manually provide initial values for both matrix factors.

References

Boutsidis C and Gallopoulos E (2008). "SVD based initialization: A head start for nonnegative matrix factorization." _Pattern Recognition_, *41*(4), pp. 1350-1362. ISSN 00313203, <URL: http://dx.doi.org/10.1016/j.patcog.2007.09.010>, <URL: http://linkinghub.elsevier.com/retrieve/pii/S0031320307004359>.

Examples

```
# list all registered seeding methods
nmfSeed()
# retrieve one of the methods
nmfSeed('ica')
```

NMFSeed-class *Base class that defines the interface for NMF seeding methods.*

Description

This class implements a simple wrapper strategy object that defines a unified interface to seeding methods, that are used to initialise NMF models before fitting them with any NMF algorithm.

Slots

name character string giving the name of the seeding strategy

method workhorse function that implements the seeding strategy. It must have signature `(object="NMF", x="matrix", ...)` and initialise the NMF model `object` with suitable values for fitting the target matrix `x`.

Methods

algorithm `signature(object = "NMFSeed")`: Returns the workhorse function of the seeding method described by `object`.

algorithm<- `signature(object = "NMFSeed", value = "function")`: Sets the workhorse function of the seeding method described by `object`.

NMFSeed `signature(key = "NMFSeed")`: Creates an `NMFSeed` based on a template object (Constructor-Copy), in particular it uses the **same** name.

show `signature(object = "NMFSeed")`: Show method for objects of class `NMFSeed`

`NMFstd-class` *NMF Model - Standard model*

Description

This class implements the standard model of Nonnegative Matrix Factorization. It provides a general structure and generic functions to manage factorizations that follow the standard NMF model, as defined by *Lee et al. (2001)*.

Details

Let V be a $n \times m$ non-negative matrix and r a positive integer. In its standard form (see references below), a NMF of V is commonly defined as a pair of matrices (W, H) such that:

$$V \equiv WH,$$

where:

- W and H are $n \times r$ and $r \times m$ matrices respectively with non-negative entries;
- $\equiv$ is to be understood with respect to some loss function. Common choices of loss functions are based on Frobenius norm or Kullback-Leibler divergence.

Integer r is called the *factorization rank*. Depending on the context of application of NMF, the columns of W and H are given different names:

columns of `W` basis vector, metagenes, factors, source, image basis

columns of `H` mixture coefficients, metagene sample expression profiles, weights

rows of `H` basis profiles, metagene expression profiles

NMF approaches have been successfully applied to several fields. The package NMF was implemented trying to use names as generic as possible for objects and methods.

The following terminology is used:

samples the columns of the target matrix V

features the rows of the target matrix V

basis matrix the first matrix factor W

basis vectors the columns of first matrix factor W

mixture matrix the second matrix factor H

mixtures coefficients the columns of second matrix factor H

However, because the package NMF was primarily implemented to work with gene expression microarray data, it also provides a layer to easily and intuitively work with objects from the Bioconductor base framework. See bioc-NMF for more details.

Slots

W A `matrix` that contains the basis matrix, i.e. the *first* matrix factor of the factorisation

H A `matrix` that contains the coefficient matrix, i.e. the *second* matrix factor of the factorisation

bterms a `data.frame` that contains the primary data that define fixed basis terms. See `bterms`.

ibterms integer vector that contains the indexes of the basis components that are fixed, i.e. for which only the coefficient are estimated.

IMPORTANT: This slot is set on construction of an NMF model via `nmfModel` and is not recommended to not be subsequently changed by the end-user.

cterms a `data.frame` that contains the primary data that define fixed coefficient terms. See `cterms`.

icterms integer vector that contains the indexes of the basis components that have fixed coefficients, i.e. for which only the basis vectors are estimated.

IMPORTANT: This slot is set on construction of an NMF model via `nmfModel` and is not recommended to not be subsequently changed by the end-user.

Methods

.basis `signature(object = "NMFstd")`: Get the basis matrix in standard NMF models

This function returns slot `W` of `object`.

.basis<- `signature(object = "NMFstd", value = "matrix")`: Set the basis matrix in standard NMF models

This function sets slot `W` of `object`.

bterms<- `signature(object = "NMFstd")`: Default method tries to coerce `value` into a `data.frame` with `as.data.frame`.

.coef `signature(object = "NMFstd")`: Get the mixture coefficient matrix in standard NMF models

This function returns slot `H` of `object`.

.coef<- `signature(object = "NMFstd", value = "matrix")`: Set the mixture coefficient matrix in standard NMF models

This function sets slot `H` of `object`.

cterms<- `signature(object = "NMFstd")`: Default method tries to coerce `value` into a `data.frame` with `as.data.frame`.

fitted `signature(object = "NMFstd")`: Compute the target matrix estimate in *standard NMF models*.

The estimate matrix is computed as the product of the two matrix slots `W` and `H`:

$$\hat{V} = WH$$

ibterms `signature(object = "NMFstd")`: Method for standard NMF models, which returns the integer vector that is stored in slot `ibterms` when a formula-based NMF model is instantiated.

icterms `signature(object = "NMFstd")`: Method for standard NMF models, which returns the integer vector that is stored in slot `icterms` when a formula-based NMF model is instantiated.

References

Lee DD and Seung H (2001). "Algorithms for non-negative matrix factorization." _Advances in neural information processing systems_. <URL: http://scholar.google.com/scholar?q=intitle:Algorithms+for+non-negative+matrix+factorization>.

See Also

Other NMF-model: `initialize,NMFOffset-method`, `NMFns-class`, `NMFOffset-class`

Examples

```
# create a completely empty NMFstd object
new('NMFstd')

# create a NMF object based on one random matrix: the missing matrix is deduced
# Note this only works when using factory method NMF
n <- 50; r <- 3;
w <- rmatrix(n, r)
nmfModel(W=w)

# create a NMF object based on random (compatible) matrices
p <- 20
h <- rmatrix(r, p)
nmfModel(W=w, H=h)

# create a NMF object based on incompatible matrices: generate an error
h <- rmatrix(r+1, p)
try( new('NMFstd', W=w, H=h) )
try( nmfModel(w, h) )

# Giving target dimensions to the factory method allow for coping with dimension
# incompatibilty (a warning is thrown in such case)
nmfModel(r, W=w, H=h)
```

NMFStop *Stopping Criteria for NMF Iterative Strategies*

Description

The function documented here implement stopping/convergence criteria commonly used in NMF algorithms.

`NMFStop` acts as a factory method that creates stopping criterion functions from different types of values, which are subsequently used by `NMFStrategyIterative` objects to determine when to stop their iterative process.

`nmf.stop.iteration` generates a function that implements the stopping criterion that limits the number of iterations to a maximum of `n`), i.e. that returns `TRUE` if `i>=n`, `FALSE` otherwise.

`nmf.stop.threshold` generates a function that implements the stopping criterion that stops when a given stationarity threshold is achieved by successive iterations. The returned function is identical to `nmf.stop.stationary`, but with the default threshold set to `threshold`.

More precisely, the objective function is computed over n successive iterations (specified in argument `check.niter`), every `check.interval` iterations. The criterion stops when the absolute difference between the maximum and the minimum objective values over these iterations is lower than a given threshold α (specified in `stationary.th`):

`nmf.stop.connectivity` implements the stopping criterion that is based on the stationarity of the connectivity matrix.

Usage

```
NMFStop(s, check = TRUE)

nmf.stop.iteration(n)

nmf.stop.threshold(threshold)

nmf.stop.stationary(object, i, y, x,
  stationary.th = .Machine$double.eps,
  check.interval = 5 * check.niter, check.niter = 10L,
  ...)

nmf.stop.connectivity(object, i, y, x, stopconv = 40,
  check.interval = 10, ...)
```

Arguments

`s`	specification of the stopping criterion. See section *Details* for the supported formats and how they are processed.
`check`	logical that indicates if the validity of the stopping criterion function should be checked before returning it.
`n`	maximum number of iteration to perform.
`threshold`	default stationarity threshold
`object`	an NMF strategy object
`i`	the current iteration
`y`	the target matrix
`x`	the current NMF model
`stationary.th`	maximum absolute value of the gradient, for the objective function to be considered stationary.
`check.interval`	interval (in number of iterations) on which the stopping criterion is computed.
`check.niter`	number of successive iteration used to compute the stationnary criterion.
`...`	extra arguments passed to the function `objective`, which computes the objective value between `x` and `y`.
`stopconv`	number of iterations intervals over which the connectivity matrix must not change for stationarity to be achieved.

Details

`NMFStop` can take the following values:

function is returned unchanged, except when it has no arguments, in which case it assumed to be a generator, which is immediately called and should return a function that implements the actual stopping criterion;

integer the value is used to create a stopping criterion that stops at that exact number of iterations via `nmf.stop.iteration`;

numeric the value is used to create a stopping criterion that stops when at that stationary threshold via `nmf.stop.threshold`;

character must be a single string which must be an access key for registered criteria (currently available: "connectivity" and "stationary"), or the name of a function in the global environment or the namespace of the loading package.

$$\left| \frac{\max_{i-N_s+1 \leq k \leq i} D_k - \min_{i-N_s+1 \leq k \leq i} D_k}{n} \right| \leq \alpha,$$

Value

a function that can be passed to argument `.stop` of function `nmf`, which is typically used when the algorith is implemented as an iterative strategy.

a function that can be used as a stopping criterion for NMF algorithms defined as `NMFStrategyIterative` objects. That is a function with arguments `(strategy, i, target, data, ...)` that returns `TRUE` if the stopping criterion is satisfied – which in turn stops the iterative process, and `FALSE` otherwise.

`NMFStrategy` *Factory Method for NMFStrategy Objects*

Description

Creates NMFStrategy objects that wraps implementation of NMF algorithms into a unified interface.

Usage

```
  NMFStrategy(name, method, ...)

  ## S4 method for signature 'NMFStrategy,matrix,NMFfit'
run(object, y, x,
    ...)

  ## S4 method for signature 'NMFStrategy,matrix,NMF'
run(object, y, x, ...)

  ## S4 method for signature 'NMFStrategyFunction,matrix,NMFfit'
```

```
run(object,
    y, x, ...)

  ## S4 method for signature 'NMFStrategyIterative,matrix,NMFfit'
run(object,
    y, x, .stop = NULL,
    maxIter = nmf.getOption("maxIter") %||% 2000, ...)

  ## S4 method for signature 'NMFStrategyIterativeX,matrix,NMFfit'
run(object,
    y, x, maxIter, ...)

  ## S4 method for signature 'NMFStrategyOctave,matrix,NMFfit'
run(object,
    y, x, ...)
```

Arguments

`name`	name/key of an NMF algorithm.
`method`	definition of the algorithm
`...`	extra arguments passed to `new`.
`.stop`	specification of a stopping criterion, that is used instead of the one associated to the NMF algorithm. It may be specified as: • the access key of a registered stopping criterion; • a single integer that specifies the exact number of iterations to perform, which will be honoured unless a lower value is explicitly passed in argument `maxIter`. • a single numeric value that specifies the stationnarity threshold for the objective function, used in with `nmf.stop.stationary`; • a function with signature `(object="NMFStrategy", i="integer", y="matrix", x="NMF", ...)`, where `object` is the `NMFStrategy` object that describes the algorithm being run, `i` is the current iteration, `y` is the target matrix and `x` is the current value of the NMF model.
`maxIter`	maximum number of iterations to perform.
`object`	an object computed using some algorithm, or that describes an algorithm itself.
`y`	data object, e.g. a target matrix
`x`	a model object used as a starting point by the algorithm, e.g. a non-empty NMF model.

Methods

NMFStrategy `signature(name = "character", method = "function")`: Creates an `NMFStrategyFunction` object that wraps the function `method` into a unified interface.

`method` must be a function with signature `(y="matrix", x="NMFfit", ...)`, and return an object of class `NMFfit`.

NMFStrategy `signature(name = "character", method = "NMFStrategy")`: Creates an `NMFStrategy` object based on a template object (Constructor-Copy).

NMFStrategy `signature(name = "NMFStrategy", method = "missing")`: Creates an `NMFStrategy` based on a template object (Constructor-Copy), in particular it uses the **same** name.

NMFStrategy `signature(name = "missing", method = "character")`: Creates an `NMFStrategy` based on a registered NMF algorithm that is used as a template (Constructor-Copy), in particular it uses the **same** name.

It is a shortcut for `NMFStrategy(nmfAlgorithm(method, exact=TRUE), ...)`.

NMFStrategy `signature(name = "NULL", method = "NMFStrategy")`: Creates an `NMFStrategy` based on a template object (Constructor-Copy) but using a randomly generated name.

NMFStrategy `signature(name = "character", method = "character")`: Creates an `NMFStrategy` based on a registered NMF algorithm that is used as a template.

NMFStrategy `signature(name = "NULL", method = "character")`: Creates an `NMFStrategy` based on a registered NMF algorithm (Constructor-Copy) using a randomly generated name.

It is a shortcut for `NMFStrategy(NULL, nmfAlgorithm(method), ...)`.

NMFStrategy `signature(name = "character", method = "missing")`: Creates an NMFStrategy, determining its type from the extra arguments passed in `...`: if there is an argument named `Update` then an `NMFStrategyIterative` is created, or if there is an argument named `algorithm` then an `NMFStrategyFunction` is created. Calls other than these generates an error.

run `signature(object = "NMFStrategy", y = "matrix", x = "NMFfit")`: Pure virtual method defined for all NMF algorithms to ensure that a method `run` is defined by sub-classes of `NMFStrategy`.

It throws an error if called directly.

run `signature(object = "NMFStrategy", y = "matrix", x = "NMF")`: Method to run an NMF algorithm directly starting from a given NMF model.

run `signature(object = "NMFStrategyFunction", y = "matrix", x = "NMFfit")`: Runs the NMF algorithms implemented by the single R function – and stored in slot `'algorithm'` of `object`, on the data object `y`, using `x` as starting point. It is equivalent to calling `object@algorithm(y, x, ...)`.

This method is usually not called directly, but only via the function `nmf`, which takes care of many other details such as seeding the computation, handling RNG settings, or setting up parallelisation.

run `signature(object = "NMFStrategyIterative", y = "matrix", x = "NMFfit")`: Runs an NMF iterative algorithm on a target matrix `y`.

run `signature(object = "NMFStrategyOctave", y = "matrix", x = "NMFfit")`: Runs the NMF algorithms implemented by the Octave/Matlab function associated with the strategy – and stored in slot `'algorithm'` of `object`.

This method is usually not called directly, but only via the function `nmf`, which takes care of many other details such as seeding the computation, handling RNG settings, or setting up parallel computations.

NMFStrategyFunction-class

Interface for Single Function NMF Strategies

Description

This class implements the virtual interface NMFStrategy for NMF algorithms that are implemented by a single workhorse R function.

Slots

algorithm a function that implements an NMF algorithm. It must have signature `(y='matrix', x='NMFfit')`, where `y` is the target matrix to approximate and `x` is the NMF model assumed to be seeded with an appropriate initial value – as it is done internally by function nmf.

Note that argument names currently do not matter, but it is recommended to name them as specified above.

Methods

algorithm `signature(object = "NMFStrategyFunction")`: Returns the single R function that implements the NMF algorithm – as stored in slot `algorithm`.

algorithm<- `signature(object = "NMFStrategyFunction", value = "function")`: Sets the function that implements the NMF algorithm, stored in slot `algorithm`.

run `signature(object = "NMFStrategyFunction", y = "matrix", x = "NMFfit")`: Runs the NMF algorithms implemented by the single R function – and stored in slot `'algorithm'` of `object`, on the data object `y`, using `x` as starting point. It is equivalent to calling `object@algorithm(y, x, ...)`.

This method is usually not called directly, but only via the function nmf, which takes care of many other details such as seeding the computation, handling RNG settings, or setting up parallelisation.

NMFStrategyIterative-class

Interface for Algorithms: Implementation for Iterative NMF Algorithms

Description

This class provides a specific implementation for the generic function `run` – concretising the virtual interface class NMFStrategy, for NMF algorithms that conform to the following iterative schema (starred numbers indicate mandatory steps):

- 1. Initialisation
- 2*. Update the model at each iteration

- 3. Stop if some criterion is satisfied
- 4. Wrap up

This schema could possibly apply to all NMF algorithms, since these are essentially optimisation algorithms, almost all of which use iterative methods to approximate a solution of the optimisation problem. The main advantage is that it allows to implement updates and stopping criterion separately, and combine them in different ways. In particular, many NMF algorithms are based on multiplicative updates, following the approach from *Lee et al. (2001)*, which are specially suitable to be cast into this simple schema.

Slots

onInit optional function that performs some initialisation or pre-processing on the model, before starting the iteration loop.

Update mandatory function that implement the update step, which computes new values for the model, based on its previous value. It is called at each iteration, until the stopping criterion is met or the maximum number of iteration is achieved.

Stop optional function that implements the stopping criterion. It is called **before** each Update step. If not provided, the iterations are stopped after a fixed number of updates.

onReturn optional function that wraps up the result into an NMF object. It is called just before returning the

Methods

run `signature(object = "NMFStrategyIterative", y = "matrix", x = "NMFfit")`: Runs an NMF iterative algorithm on a target matrix `y`.

show `signature(object = "NMFStrategyIterative")`: Show method for objects of class `NMFStrategyIterative`

References

Lee DD and Seung H (2001). "Algorithms for non-negative matrix factorization." _Advances in neural information processing systems_. <URL: http://scholar.google.com/scholar?q=intitle:Algorithms+for+non-negative+matrix+factorization>.

`nmf_update.brunet_R` *NMF Algorithm/Updates for Kullback-Leibler Divergence*

Description

The built-in NMF algorithms described here minimise the Kullback-Leibler divergence (KL) between an NMF model and a target matrix. They use the updates for the basis and coefficient matrices (W and H) defined by *Brunet et al. (2004)*, which are essentially those from *Lee et al. (2001)*, with an stabilisation step that shift up all entries from zero every 10 iterations, to a very small positive value.

`nmf_update.brunet` implements in C++ an optimised version of the single update step.

Algorithms 'brunet' and '.R#brunet' provide the complete NMF algorithm from *Brunet et al. (2004)*, using the C++-optimised and pure R updates `nmf_update.brunet` and `nmf_update.brunet_R` respectively.

Algorithm 'KL' provides an NMF algorithm based on the C++-optimised version of the updates from *Brunet et al. (2004)*, which uses the stationarity of the objective value as a stopping criterion `nmf.stop.stationary`, instead of the stationarity of the connectivity matrix `nmf.stop.connectivity` as used by 'brunet'.

Usage

```
nmf_update.brunet_R(i, v, x, eps = .Machine$double.eps,
  ...)

nmf_update.brunet(i, v, x, copy = FALSE,
  eps = .Machine$double.eps, ...)

nmfAlgorithm.brunet_R(..., .stop = NULL,
  maxIter = nmf.getOption("maxIter") %||% 2000,
  eps = .Machine$double.eps, stopconv = 40,
  check.interval = 10)

nmfAlgorithm.brunet(..., .stop = NULL,
  maxIter = nmf.getOption("maxIter") %||% 2000,
  copy = FALSE, eps = .Machine$double.eps, stopconv = 40,
  check.interval = 10)

nmfAlgorithm.KL(..., .stop = NULL,
  maxIter = nmf.getOption("maxIter") %||% 2000,
  copy = FALSE, eps = .Machine$double.eps,
  stationary.th = .Machine$double.eps,
  check.interval = 5 * check.niter, check.niter = 10L)
```

Arguments

`i`	current iteration number.
`v`	target matrix.
`x`	current NMF model, as an `NMF` object.
`eps`	small numeric value used to ensure numeric stability, by shifting up entries from zero to this fixed value.
`...`	extra arguments. These are generally not used and present only to allow other arguments from the main call to be passed to the initialisation and stopping criterion functions (slots `onInit` and `Stop` respectively).
`copy`	logical that indicates if the update should be made on the original matrix directly (`FALSE`) or on a copy (`TRUE` - default). With `copy=FALSE` the memory footprint is very small, and some speed-up may be achieved in the case of big matrices. However, greater care should be taken due the side effect. We recommend that only experienced users use `copy=TRUE`.

`.stop`	specification of a stopping criterion, that is used instead of the one associated to the NMF algorithm. It may be specified as: • the access key of a registered stopping criterion; • a single integer that specifies the exact number of iterations to perform, which will be honoured unless a lower value is explicitly passed in argument `maxIter`. • a single numeric value that specifies the stationnarity threshold for the objective function, used in with `nmf.stop.stationary`; • a function with signature `(object="NMFStrategy", i="integer", y="matrix", x="NMF", ...)`, where `object` is the `NMFStrategy` object that describes the algorithm being run, `i` is the current iteration, `y` is the target matrix and `x` is the current value of the NMF model.
`maxIter`	maximum number of iterations to perform.
`stopconv`	number of iterations intervals over which the connectivity matrix must not change for stationarity to be achieved.
`check.interval`	interval (in number of iterations) on which the stopping criterion is computed.
`stationary.th`	maximum absolute value of the gradient, for the objective function to be considered stationary.
`check.niter`	number of successive iteration used to compute the stationnary criterion.

Details

`nmf_update.brunet_R` implements in pure R a single update step, i.e. it updates both matrices.

Author(s)

Original implementation in MATLAB: Jean-Philippe Brunet <brunet@broad.mit.edu>

Port to R and optimisation in C++: Renaud Gaujoux

Source

Original license terms:

This software and its documentation are copyright 2004 by the Broad Institute/Massachusetts Institute of Technology. All rights are reserved. This software is supplied without any warranty or guaranteed support whatsoever. Neither the Broad Institute nor MIT can not be responsible for its use, misuse, or functionality.

References

Brunet J, Tamayo P, Golub TR and Mesirov JP (2004). "Metagenes and molecular pattern discovery using matrix factorization." _Proceedings of the National Academy of Sciences of the United States of America_, *101*(12), pp. 4164-9. ISSN 0027-8424, <URL: http://dx.doi.org/10.1073/pnas.0308531101>, <URL: http://www.ncbi.nlm.nih.gov/pubmed/15016911>.

Lee DD and Seung H (2001). "Algorithms for non-negative matrix factorization." _Advances in neural information processing systems_. <URL: http://scholar.google.com/scholar?q=intitle:Algorithms+for+non-negative+matrix+factorization>.

nmf_update.euclidean.h *NMF Multiplicative Updates for Euclidean Distance*

Description

Multiplicative updates from *Lee et al. (2001)* for standard Nonnegative Matrix Factorization models $V \approx WH$, where the distance between the target matrix and its NMF estimate is measured by the – euclidean – Frobenius norm.

`nmf_update.euclidean.w` and `nmf_update.euclidean.h` compute the updated basis and coefficient matrices respectively. They use a *C++* implementation which is optimised for speed and memory usage.

`nmf_update.euclidean.w_R` and `nmf_update.euclidean.h_R` implement the same updates in *plain R*.

Usage

```
nmf_update.euclidean.h(v, w, h, eps = 10^-9,
  nbterms = 0L, ncterms = 0L, copy = TRUE)

nmf_update.euclidean.h_R(v, w, h, wh = NULL, eps = 10^-9)

nmf_update.euclidean.w(v, w, h, eps = 10^-9,
  nbterms = 0L, ncterms = 0L, weight = NULL, copy = TRUE)

nmf_update.euclidean.w_R(v, w, h, wh = NULL, eps = 10^-9)
```

Arguments

`eps`	small numeric value used to ensure numeric stability, by shifting up entries from zero to this fixed value.
`wh`	already computed NMF estimate used to compute the denominator term.
`weight`	numeric vector of sample weights, e.g., used to normalise samples coming from multiple datasets. It must be of the same length as the number of samples/columns in `v` – and `h`.
`v`	target matrix
`w`	current basis matrix
`h`	current coefficient matrix
`nbterms`	number of fixed basis terms
`ncterms`	number of fixed coefficient terms
`copy`	logical that indicates if the update should be made on the original matrix directly (`FALSE`) or on a copy (`TRUE` - default). With `copy=FALSE` the memory footprint is very small, and some speed-up may be achieved in the case of big matrices. However, greater care should be taken due the side effect. We recommend that only experienced users use `copy=TRUE`.

Details

The coefficient matrix (H) is updated as follows:

$$H_{kj} \leftarrow \frac{\max(H_{kj} W^T V)_{kj}, \varepsilon)}{(W^T W H)_{kj} + \varepsilon}$$

These updates are used by the built-in NMF algorithms `Frobenius` and `lee`.

The basis matrix (W) is updated as follows:

$$W_i k \leftarrow \frac{\max(W_i k (V H^T)_i k, \varepsilon)}{(W H H^T)_i k + \varepsilon}$$

Value

a matrix of the same dimension as the input matrix to update (i.e. `w` or `h`). If `copy=FALSE`, the returned matrix uses the same memory as the input object.

Author(s)

Update definitions by *Lee2001*.

C++ optimised implementation by Renaud Gaujoux.

References

Lee DD and Seung H (2001). "Algorithms for non-negative matrix factorization." _Advances in neural information processing systems_. <URL: http://scholar.google.com/scholar?q=intitle:Algorithms+for+non-negative+matrix+factorization>.

`nmf_update.euclidean_offset.h`

NMF Multiplicative Update for NMF with Offset Models

Description

These update rules proposed by *Badea (2008)* are modified version of the updates from *Lee et al. (2001)*, that include an offset/intercept vector, which models a common baseline for each feature accross all samples:

$$V \approx W H + I$$

`nmf_update.euclidean_offset.h` and `nmf_update.euclidean_offset.w` compute the updated NMFOffset model, using the optimized *C++* implementations.

`nmf_update.offset_R` implements a complete single update step, using plain R updates.

`nmf_update.offset` implements a complete single update step, using C++-optimised updates.

Algorithms 'offset' and '.R#offset' provide the complete NMF-with-offset algorithm from *Badea (2008)*, using the C++-optimised and pure R updates `nmf_update.offset` and `nmf_update.offset_R` respectively.

Usage

```
nmf_update.euclidean_offset.h(v, w, h, offset,
  eps = 10^-9, copy = TRUE)

nmf_update.euclidean_offset.w(v, w, h, offset,
  eps = 10^-9, copy = TRUE)

nmf_update.offset_R(i, v, x, eps = 10^-9, ...)

nmf_update.offset(i, v, x, copy = FALSE, eps = 10^-9,
  ...)

nmfAlgorithm.offset_R(..., .stop = NULL,
  maxIter = nmf.getOption("maxIter") %||% 2000,
  eps = 10^-9, stopconv = 40, check.interval = 10)

nmfAlgorithm.offset(..., .stop = NULL,
  maxIter = nmf.getOption("maxIter") %||% 2000,
  copy = FALSE, eps = 10^-9, stopconv = 40,
  check.interval = 10)
```

Arguments

offset	current value of the offset/intercept vector. It must be of length equal to the number of rows in the target matrix.
v	target matrix.
eps	small numeric value used to ensure numeric stability, by shifting up entries from zero to this fixed value.
copy	logical that indicates if the update should be made on the original matrix directly (FALSE) or on a copy (TRUE - default). With copy=FALSE the memory footprint is very small, and some speed-up may be achieved in the case of big matrices. However, greater care should be taken due the side effect. We recommend that only experienced users use copy=TRUE.
i	current iteration number.
x	current NMF model, as an NMF object.
...	extra arguments. These are generally not used and present only to allow other arguments from the main call to be passed to the initialisation and stopping criterion functions (slots onInit and Stop respectively).
.stop	specification of a stopping criterion, that is used instead of the one associated to the NMF algorithm. It may be specified as: • the access key of a registered stopping criterion; • a single integer that specifies the exact number of iterations to perform, which will be honoured unless a lower value is explicitly passed in argument maxIter. • a single numeric value that specifies the stationnarity threshold for the objective function, used in with nmf.stop.stationary;

- a function with signature `(object="NMFStrategy", i="integer", y="matrix", x="NMF", ...)`, where `object` is the `NMFStrategy` object that describes the algorithm being run, `i` is the current iteration, `y` is the target matrix and `x` is the current value of the NMF model.

`maxIter`	maximum number of iterations to perform.
`stopconv`	number of iterations intervals over which the connectivity matrix must not change for stationarity to be achieved.
`check.interval`	interval (in number of iterations) on which the stopping criterion is computed.
`w`	current basis matrix
`h`	current coefficient matrix

Details

The associated model is defined as an `NMFOffset` object. The details of the multiplicative updates can be found in *Badea (2008)*. Note that the updates are the ones defined for a single datasets, not the simultaneous NMF model, which is fit by algorithm 'siNMF' from formula-based NMF models.

Value

an `NMFOffset` model object.

Author(s)

Original update definition: Liviu Badea

Port to R and optimisation in C++: Renaud Gaujoux

References

Badea L (2008). "Extracting gene expression profiles common to colon and pancreatic adenocarcinoma using simultaneous nonnegative matrix factorization." _Pacific Symposium on Biocomputing. Pacific Symposium on Biocomputing_, *290*, pp. 267-78. ISSN 1793-5091, <URL: http://www.ncbi.nlm.nih.gov/pubmed/18229692>.

Lee DD and Seung H (2001). "Algorithms for non-negative matrix factorization." _Advances in neural information processing systems_. <URL: http://scholar.google.com/scholar?q=intitle:Algorithms+for+non-negative+matrix+factorization>.

`nmf_update.KL.h` *NMF Multiplicative Updates for Kullback-Leibler Divergence*

Description

Multiplicative updates from *Lee et al. (2001)* for standard Nonnegative Matrix Factorization models $V \approx WH$, where the distance between the target matrix and its NMF estimate is measured by the Kullback-Leibler divergence.

`nmf_update.KL.w` and `nmf_update.KL.h` compute the updated basis and coefficient matrices respectively. They use a *C++* implementation which is optimised for speed and memory usage.

`nmf_update.KL.w_R` and `nmf_update.KL.h_R` implement the same updates in *plain R*.

Usage

```
nmf_update.KL.h(v, w, h, nbterms = 0L, ncterms = 0L,
  copy = TRUE)

nmf_update.KL.h_R(v, w, h, wh = NULL)

nmf_update.KL.w(v, w, h, nbterms = 0L, ncterms = 0L,
  copy = TRUE)

nmf_update.KL.w_R(v, w, h, wh = NULL)
```

Arguments

v	target matrix
w	current basis matrix
h	current coefficient matrix
nbterms	number of fixed basis terms
ncterms	number of fixed coefficient terms
copy	logical that indicates if the update should be made on the original matrix directly (FALSE) or on a copy (TRUE - default). With copy=FALSE the memory footprint is very small, and some speed-up may be achieved in the case of big matrices. However, greater care should be taken due the side effect. We recommend that only experienced users use copy=TRUE.
wh	already computed NMF estimate used to compute the denominator term.

Details

The coefficient matrix (H) is updated as follows:

$$H_{kj} \leftarrow H_{kj} \frac{\left(sum_i \frac{W_{ik} V_{ij}}{(WH)_{ij}} \right)}{sum_i W_{ik}}.$$

These updates are used in built-in NMF algorithms KL and brunet.

The basis matrix (W) is updated as follows:

$$W_{ik} \leftarrow W_{ik} \frac{sum_j \left[\frac{H_{kj} A_{ij}}{(WH)_{ij}} \right]}{sum_j H_{kj}}$$

Value

a matrix of the same dimension as the input matrix to update (i.e. w or h). If copy=FALSE, the returned matrix uses the same memory as the input object.

Author(s)

Update definitions by *Lee2001*.

C++ optimised implementation by Renaud Gaujoux.

References

Lee DD and Seung H (2001). "Algorithms for non-negative matrix factorization." _Advances in neural information processing systems_. <URL: http://scholar.google.com/scholar?q=intitle:Algorithms+for+non-negative+matrix+factorization>.

`nmf_update.lee_R` *NMF Algorithm/Updates for Frobenius Norm*

Description

The built-in NMF algorithms described here minimise the Frobenius norm (Euclidean distance) between an NMF model and a target matrix. They use the updates for the basis and coefficient matrices (W and H) defined by *Lee et al. (2001)*.

`nmf_update.lee` implements in C++ an optimised version of the single update step.

Algorithms 'lee' and '.R#lee' provide the complete NMF algorithm from *Lee et al. (2001)*, using the C++-optimised and pure R updates `nmf_update.lee` and `nmf_update.lee_R` respectively.

Algorithm 'Frobenius' provides an NMF algorithm based on the C++-optimised version of the updates from *Lee et al. (2001)*, which uses the stationarity of the objective value as a stopping criterion `nmf.stop.stationary`, instead of the stationarity of the connectivity matrix `nmf.stop.connectivity` as used by 'lee'.

Usage

```
nmf_update.lee_R(i, v, x, rescale = TRUE, eps = 10^-9,
  ...)

nmf_update.lee(i, v, x, rescale = TRUE, copy = FALSE,
  eps = 10^-9, weight = NULL, ...)

nmfAlgorithm.lee_R(..., .stop = NULL,
  maxIter = nmf.getOption("maxIter") %||% 2000,
  rescale = TRUE, eps = 10^-9, stopconv = 40,
  check.interval = 10)

nmfAlgorithm.lee(..., .stop = NULL,
  maxIter = nmf.getOption("maxIter") %||% 2000,
  rescale = TRUE, copy = FALSE, eps = 10^-9,
  weight = NULL, stopconv = 40, check.interval = 10)

nmfAlgorithm.Frobenius(..., .stop = NULL,
  maxIter = nmf.getOption("maxIter") %||% 2000,
  rescale = TRUE, copy = FALSE, eps = 10^-9,
  weight = NULL, stationary.th = .Machine$double.eps,
  check.interval = 5 * check.niter, check.niter = 10L)
```

Arguments

rescale	logical that indicates if the basis matrix W should be rescaled so that its columns sum up to one.
i	current iteration number.
v	target matrix.
x	current NMF model, as an NMF object.
eps	small numeric value used to ensure numeric stability, by shifting up entries from zero to this fixed value.
...	extra arguments. These are generally not used and present only to allow other arguments from the main call to be passed to the initialisation and stopping criterion functions (slots onInit and Stop respectively).
copy	logical that indicates if the update should be made on the original matrix directly (FALSE) or on a copy (TRUE - default). With copy=FALSE the memory footprint is very small, and some speed-up may be achieved in the case of big matrices. However, greater care should be taken due the side effect. We recommend that only experienced users use copy=TRUE.
.stop	specification of a stopping criterion, that is used instead of the one associated to the NMF algorithm. It may be specified as: • the access key of a registered stopping criterion; • a single integer that specifies the exact number of iterations to perform, which will be honoured unless a lower value is explicitly passed in argument maxIter. • a single numeric value that specifies the stationnarity threshold for the objective function, used in with nmf.stop.stationary; • a function with signature (object="NMFStrategy", i="integer", y="matrix", x="NMF", ...), where object is the NMFStrategy object that describes the algorithm being run, i is the current iteration, y is the target matrix and x is the current value of the NMF model.
maxIter	maximum number of iterations to perform.
stopconv	number of iterations intervals over which the connectivity matrix must not change for stationarity to be achieved.
check.interval	interval (in number of iterations) on which the stopping criterion is computed.
stationary.th	maximum absolute value of the gradient, for the objective function to be considered stationary.
check.niter	number of successive iteration used to compute the stationnary criterion.
weight	numeric vector of sample weights, e.g., used to normalise samples coming from multiple datasets. It must be of the same length as the number of samples/columns in v – and h.

Details

nmf_update.lee_R implements in pure R a single update step, i.e. it updates both matrices.

Author(s)

Original update definition: D D Lee and HS Seung

Port to R and optimisation in C++: Renaud Gaujoux

References

Lee DD and Seung H (2001). "Algorithms for non-negative matrix factorization." _Advances in neural information processing systems_. <URL: http://scholar.google.com/scholar?q=intitle:Algorithms+for+non-negative+matrix+factorization>.

---

nmf_update.lsnmf *Multiplicative Updates for LS-NMF*

Description

Implementation of the updates for the LS-NMF algorithm from *Wang et al. (2006)*.

`wrss` implements the objective function used by the LS-NMF algorithm.

Usage

```
nmf_update.lsnmf(i, X, object, weight, eps = 10^-9, ...)

wrss(object, X, weight)

nmfAlgorithm.lsNMF(..., .stop = NULL,
  maxIter = nmf.getOption("maxIter") %||% 2000, weight,
  eps = 10^-9, stationary.th = .Machine$double.eps,
  check.interval = 5 * check.niter, check.niter = 10L)
```

Arguments

`i`	current iteration
`X`	target matrix
`object`	current NMF model
`weight`	value for Σ, i.e. the weights that are applied to each entry in `X` by `X * weight` (= entry wise product). Weights are usually specified as a matrix of the same dimension as `X` (e.g. uncertainty estimates for each measurement), but may also be passed as a vector, in which case the standard rules for entry wise product between matrices and vectors apply (e.g. recylcing elements).
`eps`	small number passed to the standard euclidean-based NMF updates (see `nmf_update.euclidean`).
`...`	extra arguments (not used)
`.stop`	specification of a stopping criterion, that is used instead of the one associated to the NMF algorithm. It may be specified as: • the access key of a registered stopping criterion;

- a single integer that specifies the exact number of iterations to perform, which will be honoured unless a lower value is explicitly passed in argument `maxIter`.
- a single numeric value that specifies the stationnarity threshold for the objective function, used in with `nmf.stop.stationary`;
- a function with signature `(object="NMFStrategy", i="integer", y="matrix", x="NMF", ...)`, where `object` is the `NMFStrategy` object that describes the algorithm being run, `i` is the current iteration, `y` is the target matrix and `x` is the current value of the NMF model.

`maxIter`	maximum number of iterations to perform.
`stationary.th`	maximum absolute value of the gradient, for the objective function to be considered stationary.
`check.interval`	interval (in number of iterations) on which the stopping criterion is computed.
`check.niter`	number of successive iteration used to compute the stationnary criterion.

Value

updated object `object`

References

Wang G, Kossenkov AV and Ochs MF (2006). "LS-NMF: a modified non-negative matrix factorization algorithm utilizing uncertainty estimates." _BMC bioinformatics_, *7*, pp. 175. ISSN 1471-2105, <URL: http://dx.doi.org/10.1186/1471-2105-7-175>, <URL: http://www.ncbi.nlm.nih.gov/pubmed/16569230>.

`nmf_update.ns` *NMF Multiplicative Update for Nonsmooth Nonnegative Matrix Factorization (nsNMF).*

Description

These update rules, defined for the `NMFns` model $V \approx WSH$ from *Pascual-Montano et al. (2006)*, that introduces an intermediate smoothing matrix to enhance sparsity of the factors.

`nmf_update.ns` computes the updated nsNMF model. It uses the optimized *C++* implementations `nmf_update.KL.w` and `nmf_update.KL.h` to update W and H respectively.

`nmf_update.ns_R` implements the same updates in *plain R*.

Algorithms 'nsNMF' and '.R#nsNMF' provide the complete NMF algorithm from *Pascual-Montano et al. (2006)*, using the C++-optimised and plain R updates `nmf_update.brunet` and `nmf_update.brunet_R` respectively. The stopping criterion is based on the stationarity of the connectivity matrix.

Usage

```
nmf_update.ns(i, v, x, copy = FALSE, ...)

nmf_update.ns_R(i, v, x, ...)

nmfAlgorithm.nsNMF_R(..., .stop = NULL,
  maxIter = nmf.getOption("maxIter") %||% 2000,
  stopconv = 40, check.interval = 10)

nmfAlgorithm.nsNMF(..., .stop = NULL,
  maxIter = nmf.getOption("maxIter") %||% 2000,
  copy = FALSE, stopconv = 40, check.interval = 10)
```

Arguments

i	current iteration number.
v	target matrix.
x	current NMF model, as an NMF object.
copy	logical that indicates if the update should be made on the original matrix directly (FALSE) or on a copy (TRUE - default). With copy=FALSE the memory footprint is very small, and some speed-up may be achieved in the case of big matrices. However, greater care should be taken due the side effect. We recommend that only experienced users use copy=TRUE.
...	extra arguments. These are generally not used and present only to allow other arguments from the main call to be passed to the initialisation and stopping criterion functions (slots onInit and Stop respectively).
.stop	specification of a stopping criterion, that is used instead of the one associated to the NMF algorithm. It may be specified as: • the access key of a registered stopping criterion; • a single integer that specifies the exact number of iterations to perform, which will be honoured unless a lower value is explicitly passed in argument maxIter. • a single numeric value that specifies the stationnarity threshold for the objective function, used in with nmf.stop.stationary; • a function with signature (object="NMFStrategy", i="integer", y="matrix", x="NMF", ...), where object is the NMFStrategy object that describes the algorithm being run, i is the current iteration, y is the target matrix and x is the current value of the NMF model.
maxIter	maximum number of iterations to perform.
stopconv	number of iterations intervals over which the connectivity matrix must not change for stationarity to be achieved.
check.interval	interval (in number of iterations) on which the stopping criterion is computed.

Details

The multiplicative updates are based on the updates proposed by *Brunet et al. (2004)*, except that the NMF estimate WH is replaced by WSH and W (resp. H) is replaced by WS (resp. SH) in the update of H (resp. W).

See `nmf_update.KL` for more details on the update formula.

Value

an `NMFns` model object.

References

Pascual-Montano A, Carazo JM, Kochi K, Lehmann D and Pascual-marqui RD (2006). "Nonsmooth nonnegative matrix factorization (nsNMF)." _IEEE Trans. Pattern Anal. Mach. Intell_, *28*, pp. 403-415.

Brunet J, Tamayo P, Golub TR and Mesirov JP (2004). "Metagenes and molecular pattern discovery using matrix factorization." _Proceedings of the National Academy of Sciences of the United States of America_, *101*(12), pp. 4164-9. ISSN 0027-8424, <URL: http://dx.doi.org/10.1073/pnas.0308531101>, <URL: http://www.ncbi.nlm.nih.gov/pubmed/15016911>.

`nneg` *Transforming from Mixed-sign to Nonnegative Data*

Description

`nneg` is a generic function to transform a data objects that contains negative values into a similar object that only contains values that are nonnegative or greater than a given threshold.

`posneg` is a shortcut for `nneg(..., method='posneg')`, to split mixed-sign data into its positive and negative part. See description for method `"posneg"`, in `nneg`.

`rposneg` performs the "reverse" transformation of the `posneg` function.

Usage

```
  nneg(object, ...)

  ## S4 method for signature 'matrix'
nneg(object,
    method = c("pmax", "posneg", "absolute", "min"),
    threshold = 0, shift = TRUE)

  posneg(...)

  rposneg(object, ...)

  ## S4 method for signature 'matrix'
rposneg(object, unstack = TRUE)
```

Arguments

object	The data object to transform
...	extra arguments to allow extension or passed down to nneg,matrix or rposneg,matrix in subsequent calls.
method	Name of the transformation method to use, that is partially matched against the following possible methods: **pmax** Each entry is constrained to be above threshold threshold. **posneg** The matrix is split into its "positive" and "negative" parts, with the entries of each part constrained to be above threshold threshold. The result consists in these two parts stacked in rows (i.e. rbind-ed) into a single matrix, which has double the number of rows of the input matrix object. **absolute** The absolute value of each entry is constrained to be above threshold threshold. **min** Global shift by adding the minimum entry to each entry, only if it is negative, and then apply threshold.
threshold	Nonnegative lower threshold value (single numeric). See argument shit for details on how the threshold is used and affects the result.
shift	a logical indicating whether the entries below the threshold value threshold should be forced (shifted) to 0 (default) or to the threshold value itself. In other words, if shift=TRUE (default) all entries in the result matrix are either 0 or strictly greater than threshold. They are all greater or equal than threshold otherwise.
unstack	Logical indicating whether the positive and negative parts should be unstacked and combined into a matrix as pos - neg, which contains half the number of rows of object (default), or left stacked as [pos; -neg].

Value

an object of the same class as argument object.

an object of the same type of object

Methods

nneg signature(object = "matrix"): Transforms a mixed-sign matrix into a nonnegative matrix, optionally apply a lower threshold. This is the workhorse method, that is eventually called by all other methods defined in the NMF package.

nneg signature(object = "NMF"): Apply nneg to the basis matrix of an NMF object (i.e. basis(object)). All extra arguments in ... are passed to the method nneg,matrix.

rposneg signature(object = "NMF"): Apply rposneg to the basis matrix of an NMF object.

See Also

pmax

Other transforms: t.NMF

Examples

```
#----------
# nneg,matrix-method
#----------
# random mixed sign data (normal distribution)
set.seed(1)
x <- rmatrix(5,5, rnorm, mean=0, sd=5)
x

# pmax (default)
nneg(x)
# using a threshold
nneg(x, threshold=2)
# without shifting the entries lower than threshold
nneg(x, threshold=2, shift=FALSE)

# posneg: split positive and negative part
nneg(x, method='posneg')
nneg(x, method='pos', threshold=2)

# absolute
nneg(x, method='absolute')
nneg(x, method='abs', threshold=2)

# min
nneg(x, method='min')
nneg(x, method='min', threshold=2)

#----------
# nneg,NMF-method
#----------
# random
M <- nmfModel(x, rmatrix(ncol(x), 3))
nnM <- nneg(M)
basis(nnM)
# mixture coefficients are not affected
identical( coef(M), coef(nnM) )

#----------
# posneg
#----------
# shortcut for the "posneg" transformation
posneg(x)
posneg(x, 2)

#----------
# rposneg,matrix-method
#----------
# random mixed sign data (normal distribution)
set.seed(1)
x <- rmatrix(5,5, rnorm, mean=0, sd=5)
```

```
x

# posneg-transform: split positive and negative part
y <- posneg(x)
dim(y)
# posneg-reverse
z <- rposneg(y)
identical(x, z)
rposneg(y, unstack=FALSE)

# But posneg-transformation with a non zero threshold is not reversible
y1 <- posneg(x, 1)
identical(rposneg(y1), x)

#----------
# rposneg,NMF-method
#----------
# random mixed signed NMF model
M <- nmfModel(rmatrix(10, 3, rnorm), rmatrix(3, 4))
# split positive and negative part
nnM <- posneg(M)
M2 <- rposneg(nnM)
identical(M, M2)
```

`objective,NMFfit-method`

Returns the objective function associated with the algorithm that computed the fitted NMF model `object`*, or the objective value with respect to a given target matrix* `y` *if it is supplied.*

Description

Returns the objective function associated with the algorithm that computed the fitted NMF model `object`, or the objective value with respect to a given target matrix `y` if it is supplied.

Usage

```
  ## S4 method for signature 'NMFfit'
objective(object, y)
```

Arguments

`y`	optional target matrix used to compute the objective value.
`object`	an object computed using some algorithm, or that describes an algorithm itself.

offset,NMFfit-method *Returns the offset from the fitted model.*

Description

Returns the offset from the fitted model.

Usage

```
  ## S4 method for signature 'NMFfit'
offset(object)
```

Arguments

object	An offset to be included in a model frame

offset,NMFOffset-method *Offsets in NMF Models with Offset*

Description

The function offset returns the offset vector from an NMF model that has an offset, e.g. an NMFOffset model.

Usage

```
  ## S4 method for signature 'NMFOffset'
offset(object)
```

Arguments

object	an instance of class NMFOffset.

`options-NMF` *NMF Package Specific Options*

Description

NMF Package Specific Options

`nmf.options` sets/get single or multiple options, that are specific to the NMF package. It behaves in the same way as `options`.

`nmf.getOption` returns the value of a single option, that is specific to the NMF package. It behaves in the same way as `getOption`.

`nmf.resetOptions` reset all NMF specific options to their default values.

`nmf.printOptions` prints all NMF specific options along with their default values, in a relatively compact way.

Usage

```
nmf.options(...)

nmf.getOption(x, default = NULL)

nmf.resetOptions(..., ALL = FALSE)

nmf.printOptions()
```

Arguments

`...`	option specifications. For `nmf.options` this can be named arguments or a single unnamed argument that is a named list (see `options`. For `nmf.resetOptions`, this must be the names of the options to reset.
`ALL`	logical that indicates if options that are not part of the default set of options should be removed.
`x`	a character string holding an option name.
`default`	if the specified option is not set in the options list, this value is returned. This facilitates retrieving an option and checking whether it is set and setting it separately if not.

Available options

cores Default number of cores to use to perform parallel NMF computations. Note that this option is effectively used only if the global option `'cores'` is not set. Moreover, the number of cores can also be set at runtime, in the call to `nmf`, via arguments `.pbackend` or `.options` (see `nmf` for more details).

default.algorithm Default NMF algorithm used by the `nmf` function when argument `method` is missing. The value should the key of one of the registered NMF algorithms or a valid specification of an NMF algorithm. See `?nmfAlgorithm`.

default.seed Default seeding method used by the `nmf` function when argument `seed` is missing. The value should the key of one of the registered seeding methods or a vallid specification of a seeding method. See `?nmfSeed`.

track Toggle default residual tracking. When `TRUE`, the `nmf` function compute and store the residual track in the result – if not otherwise specified in argument `.options`. Note that tracking may significantly slow down the computations.

track.interval Number of iterations between two points in the residual track. This option is relevant only when residual tracking is enabled. See `?nmf`.

error.track this is a symbolic link to option `track` for backward compatibility.

pbackend Default loop/parallel foreach backend used by the `nmf` function when argument `.pbackend` is missing. Currently the following values are supported: `'par'` for multicore, `'seq'` for sequential, `NA` for standard `sapply` (i.e. do not use a foreach loop), `NULL` for using the currently registered foreach backend.

parallel.backend this is a symbolic link to option `pbackend` for backward compatibility.

gc Interval/frequency (in number of runs) at which garbage collection is performed.

verbose Default level of verbosity.

debug Toogles debug mode. In this mode the console output may be very – very – messy, and is aimed at debugging only.

maxIter Default maximum number of iteration to use (default NULL). This option is for internal/technical usage only, to globally speed up examples or tests of NMF algorithms. To be used with care at one's own risk... It is documented here so that advanced users are aware of its existence, and can avoid possible conflict with their own custom options.

Examples

```
# show all NMF specific options
nmf.printOptions()

# get some options
nmf.getOption('verbose')
nmf.getOption('pbackend')
# set new values
nmf.options(verbose=TRUE)
nmf.options(pbackend='mc', default.algorithm='lee')
nmf.printOptions()

# reset to default
nmf.resetOptions()
nmf.printOptions()
```

parallel-NMF *Utilities for Parallel Computations*

Description

Utilities for Parallel Computations

`ts_eval` generates a thread safe version of `eval`. It uses boost mutexes provided by the *synchronicity* package. The generated function has arguments `expr` and `envir`, which are passed to `eval`.

`ts_tempfile` generates a *unique* temporary filename that includes the name of the host machine and/or the caller's process id, so that it is thread safe.

`hostfile` generates a temporary filename composed with the name of the host machine and/or the current process id.

`gVariable` generates a function that access a global static variable, possibly in shared memory (only for numeric matrix-coercible data in this case). It is used primarily in parallel computations, to preserve data accross computations that are performed by the same process.

Usage

```
ts_eval(mutex = synchronicity::boost.mutex(),
  verbose = FALSE)

ts_tempfile(pattern = "file", ..., host = TRUE,
  pid = TRUE)

hostfile(pattern = "file", tmpdir = tempdir(),
  fileext = "", host = TRUE, pid = TRUE)

gVariable(init, shared = FALSE)
```

Arguments

`mutex`	a mutex or a mutex descriptor. If missing, a new mutex is created via the function *boost.mutex* from the *synchronicity* package.
`verbose`	a logical that indicates if messages should be printed when locking and unlocking the mutex.
`...`	extra arguments passed to `tempfile`.
`host`	logical that indicates if the host machine name should be appear in the filename.
`pid`	logical that indicates if the current process id be appear in the filename.
`init`	initial value
`shared`	a logical that indicates if the variable should be stored in shared memory or in a local environment.
`pattern`	a non-empty character vector giving the initial part of the name.
`tmpdir`	a non-empty character vector giving the directory name
`fileext`	a non-empty character vector giving the file extension

plot,NMFfit,missing-method

Plots the residual track computed at regular interval during the fit of the NMF model x.

Description

Plots the residual track computed at regular interval during the fit of the NMF model x.

Usage

```
  ## S4 method for signature 'NMFfit,missing'
plot(x, y, skip = -1, ...)
```

Arguments

skip	an integer that indicates the number of points to skip/remove from the beginning of the curve. If skip=1L (default) only the initial residual – that is computed before any iteration, is skipped, if present in the track (it associated with iteration 0).
x	the coordinates of points in the plot. Alternatively, a single plotting structure, function or *any* R *object with a* plot *method* can be provided.
y	the y coordinates of points in the plot, *optional* if x is an appropriate structure.
...	Arguments to be passed to methods, such as graphical parameters (see par). Many methods will accept the following arguments: type what type of plot should be drawn. Possible types are • "p" for **p**oints, • "l" for **l**ines, • "b" for **b**oth, • "c" for the lines part alone of "b", • "o" for both '**o**verplotted', • "h" for '**h**istogram' like (or 'high-density') vertical lines, • "s" for stair **s**teps, • "S" for other **s**teps, see 'Details' below, • "n" for no plotting. All other types give a warning or an error; using, e.g., type = "punkte" being equivalent to type = "p" for S compatibility. Note that some methods, e.g. plot.factor, do not accept this. main an overall title for the plot: see title. sub a sub title for the plot: see title. xlab a title for the x axis: see title. ylab a title for the y axis: see title. asp the y/x aspect ratio, see plot.window.

predict *Clustering and Prediction*

Description

The methods predict for NMF models return the cluster membership of each sample or each feature. Currently the classification/prediction of new data is not implemented.

Usage

```
  predict(object, ...)

  ## S4 method for signature 'NMF'
predict(object,
    what = c("columns", "rows", "samples", "features"),
    prob = FALSE, dmatrix = FALSE)

  ## S4 method for signature 'NMFfitX'
predict(object,
    what = c("columns", "rows", "samples", "features", "consensus", "chc"),
    dmatrix = FALSE, ...)
```

Arguments

object	an NMF model
what	a character string that indicates the type of cluster membership should be returned: 'columns' or 'rows' for clustering the colmuns or the rows of the target matrix respectively. The values 'samples' and 'features' are aliases for 'colmuns' and 'rows' respectively.
prob	logical that indicates if the relative contributions of/to the dominant basis component should be computed and returned. See *Details*.
dmatrix	logical that indicates if a dissimiliarity matrix should be attached to the result. This is notably used internally when computing NMF clustering silhouettes.
...	additional arguments affecting the predictions produced.

Details

The cluster membership is computed as the index of the dominant basis component for each sample (what='samples' or 'columns') or each feature (what='features' or 'rows'), based on their corresponding entries in the coefficient matrix or basis matrix respectively.

For example, if what='samples', then the dominant basis component is computed for each column of the coefficient matrix as the row index of the maximum within the column.

If argument prob=FALSE (default), the result is a factor. Otherwise a list with two elements is returned: element predict contains the cluster membership index (as a factor) and element prob contains the relative contribution of the dominant component to each sample (resp. the relative contribution of each feature to the dominant basis component):

- Samples:

$$p_j = x_{k_0} / \sum_k x_k$$

, for each sample $1 \leq j \leq p$, where x_k is the contribution of the k-th basis component to j-th sample (i.e. `H[k ,j]`), and x_{k_0} is the maximum of these contributions.

- Features:

$$p_i = y_{k_0} / \sum_k y_k$$

, for each feature $1 \leq i \leq p$, where y_k is the contribution of the k-th basis component to i-th feature (i.e. `W[i, k]`), and y_{k_0} is the maximum of these contributions.

Methods

predict `signature(object = "NMF")`: Default method for NMF models

predict `signature(object = "NMFfitX")`: Returns the cluster membership index from an NMF model fitted with multiple runs.

Besides the type of clustering available for any NMF models (`'columns'`, `'rows'`, `'samples'`, `'features'`), this method can return the cluster membership index based on the consensus matrix, computed from the multiple NMF runs.

Argument `what` accepts the following extra types:

`'chc'` returns the cluster membership based on the hierarchical clustering of the consensus matrix, as performed by `consensushc`.

`'consensus'` same as `'chc'` but the levels of the membership index are re-labeled to match the order of the clusters as they would be displayed on the associated dendrogram, as re-ordered on the default annotation track in consensus heatmap produced by `consensusmap`.

References

Brunet J, Tamayo P, Golub TR and Mesirov JP (2004). "Metagenes and molecular pattern discovery using matrix factorization." _Proceedings of the National Academy of Sciences of the United States of America_, *101*(12), pp. 4164-9. ISSN 0027-8424, <URL: http://dx.doi.org/10.1073/pnas.0308531101>, <URL: http://www.ncbi.nlm.nih.gov/pubmed/15016911>.

Pascual-Montano A, Carazo JM, Kochi K, Lehmann D and Pascual-marqui RD (2006). "Nonsmooth nonnegative matrix factorization (nsNMF)." _IEEE Trans. Pattern Anal. Mach. Intell._, *28*, pp. 403-415.

Examples

```
# random target matrix
v <- rmatrix(20, 10)
# fit an NMF model
x <- nmf(v, 5)

# predicted column and row clusters
predict(x)
predict(x, 'rows')
```

```
# with relative contributions of each basis component
predict(x, prob=TRUE)
predict(x, 'rows', prob=TRUE)
```

`profplot` *Plotting Expression Profiles*

Description

Plotting Expression Profiles

When using NMF for clustering in particular, one looks for strong associations between the basis and a priori known groups of samples. Plotting the profiles may highlight such patterns.

Usage

```
profplot(x, ...)

## Default S3 method:
profplot(x, y,
   scale = c("none", "max", "c1"), match.names = TRUE,
   legend = TRUE, confint = TRUE, Colv, labels,
   annotation, ..., add = FALSE)
```

Arguments

`x`	a matrix or an NMF object from which is extracted the mixture coefficient matrix. It is extracted from the best fit if `x` is the results from multiple NMF runs.
`y`	a matrix or an NMF object from which is extracted the mixture coefficient matrix. It is extracted from the best fit if `y` is the results from multiple NMF runs.
`scale`	specifies how the data should be scaled before plotting. If `'none'` or `NA`, then no scaling is applied and the "raw" data is plotted. If `TRUE` or `'max'` then each row of both matrices are normalised with their respective maximum values. If `'c1'`, then each column of both matrix is scaled into proportions (i.e. to sum up to one). Default is `'none'`.
`match.names`	a logical that indicates if the profiles in `y` should be subset and/or re-ordered to match the profile names in `x` (i.e. the rownames). This is attempted only when both `x` and `y` have names.
`legend`	a logical that specifies whether drawing the legend or not, or coordinates specifications passed to argument `x` of `legend`, that specifies the position of the legend.
`confint`	logical that indicates if confidence intervals for the R-squared should be shown in legend.
`Colv`	specifies the way the columns of `x` are ordered before plotting. It is used only when `y` is missing. It can be: • a single numeric value, specifying the index of a row of `x`, that is used to order the columns by `x[, order(x[abs(Colv),])]`. Decreasing order is specified with a negative index.

- an integer vector directly specifying the order itself, in which case the columns are ordered by `x[, Colv]`
- a factor used to order the columns by `x[, order(Colv)]` and as argument `annotation` if this latter is missing or not `NA`.
- any other object with a suitable `order` method. The columns are by `x[, order(Colv)]`

`labels` a character vector containing labels for each sample (i.e. each column of `x`). These are used for labelling the x-axis.

`annotation` a factor annotating each sample (i.e. each column of `x`). If not missing, a coloured raw is plotted under the x-axis and annotates each sample accordingly. If argument `Colv` is a factor, then it is used to annotate the plot, unless `annotation=NA`.

`...` graphical parameters passed to `matplot` or `matpoints`.

`add` logical that indicates if the plot should be added as points to a previous plot

Details

The function can also be used to compare the profiles from two NMF models or mixture coefficient matrices. In this case, it draws a scatter plot of the paired profiles.

See Also

`profcor`

Examples

```
# create a random target matrix
v <- rmatrix(30, 10)

# fit a single NMF model
res <- nmf(v, 3)
profplot(res)

# fit a multi-run NMF model
res2 <- nmf(v, 3, nrun=2)
# ordering according to first profile
profplot(res2, Colv=1) # increasing

# draw a profile correlation plot: this show how the basis components are
# returned in an unpredictable order
profplot(res, res2)

# looking at all the correlations allow to order the components in a "common" order
profcor(res, res2)
```

purity *Purity and Entropy of a Clustering*

Description

The functions purity and entropy respectively compute the purity and the entropy of a clustering given *a priori* known classes.

The purity and entropy measure the ability of a clustering method, to recover known classes (e.g. one knows the true class labels of each sample), that are applicable even when the number of cluster is different from the number of known classes. *Kim et al. (2007)* used these measures to evaluate the performance of their alternate least-squares NMF algorithm.

Usage

```
  purity(x, y, ...)

  entropy(x, y, ...)

  ## S4 method for signature 'NMFfitXn,ANY'
purity(x, y, method = "best",
    ...)

  ## S4 method for signature 'NMFfitXn,ANY'
entropy(x, y, method = "best",
    ...)
```

Arguments

x	an object that can be interpreted as a factor or can generate such an object, e.g. via a suitable method predict, which gives the cluster membership for each sample.
y	a factor or an object coerced into a factor that gives the true class labels for each sample. It may be missing if x is a contingency table.
...	extra arguments to allow extension, and usually passed to the next method.
method	a character string that specifies how the value is computed. It may be either 'best' or 'mean' to compute the best or mean purity respectively.

Details

Suppose we are given l categories, while the clustering method generates k clusters.

The purity of the clustering with respect to the known categories is given by:

$$Purity = \frac{1}{n} \sum_{q=1}^{k} \max_{1 \leq j \leq l} n_q^j$$

,

where:

- n is the total number of samples;
- n_q^j is the number of samples in cluster q that belongs to original class j ($1 \leq j \leq l$).

The purity is therefore a real number in $[0, 1]$. The larger the purity, the better the clustering performance.

The entropy of the clustering with respect to the known categories is given by:

$$Entropy = -\frac{1}{n \log_2 l} \sum_{q=1}^{k} \sum_{j=1}^{l} n_q^j \log_2 \frac{n_q^j}{n_q}$$

,

where:

- n is the total number of samples;
- n is the total number of samples in cluster q ($1 \leq q \leq k$);
- n_q^j is the number of samples in cluster q that belongs to original class j ($1 \leq j \leq l$).

The smaller the entropy, the better the clustering performance.

Value

a single numeric value

the entropy (i.e. a single numeric value)

Methods

entropy `signature(x = "table", y = "missing")`: Computes the purity directly from the contingency table `x`.

This is the workhorse method that is eventually called by all other methods.

entropy `signature(x = "factor", y = "ANY")`: Computes the purity on the contingency table of `x` and `y`, that is coerced into a factor if necessary.

entropy `signature(x = "ANY", y = "ANY")`: Default method that should work for results of clustering algorithms, that have a suitable `predict` method that returns the cluster membership vector: the purity is computed between `x` and `predict{y}`

entropy `signature(x = "NMFfitXn", y = "ANY")`: Computes the best or mean entropy across all NMF fits stored in `x`.

purity `signature(x = "table", y = "missing")`: Computes the purity directly from the contingency table `x`

purity `signature(x = "factor", y = "ANY")`: Computes the purity on the contingency table of `x` and `y`, that is coerced into a factor if necessary.

purity `signature(x = "ANY", y = "ANY")`: Default method that should work for results of clustering algorithms, that have a suitable `predict` method that returns the cluster membership vector: the purity is computed between `x` and `predict{y}`

purity `signature(x = "NMFfitXn", y = "ANY")`: Computes the best or mean purity across all NMF fits stored in `x`.

References

Kim H and Park H (2007). "Sparse non-negative matrix factorizations via alternating non-negativity-constrained least squares for microarray data analysis." _Bioinformatics (Oxford, England)_, *23*(12), pp. 1495-502. ISSN 1460-2059, <URL: http://dx.doi.org/10.1093/bioinformatics/btm134>, <URL: http://www.ncbi.nlm.nih.gov/pubmed/17483501>.

See Also

Other assess: sparseness

Examples

```
# generate a synthetic dataset with known classes: 50 features, 18 samples (5+5+8)
n <- 50; counts <- c(5, 5, 8);
V <- syntheticNMF(n, counts)
cl <- unlist(mapply(rep, 1:3, counts))

# perform default NMF with rank=2
x2 <- nmf(V, 2)
purity(x2, cl)
entropy(x2, cl)
# perform default NMF with rank=2
x3 <- nmf(V, 3)
purity(x3, cl)
entropy(x3, cl)
```

---

randomize *Randomizing Data*

Description

randomize permutates independently the entries in each column of a matrix-like object, to produce random data that can be used in permutation tests or bootstrap analysis.

Usage

```
randomize(x, ...)
```

Arguments

x	data to be permutated. It must be an object suitable to be passed to the function apply.
...	extra arguments passed to the function sample.

Details

In the context of NMF, it may be used to generate random data, whose factorization serves as a reference for selecting a factorization rank, that does not overfit the data.

Value

a matrix

Examples

```
x <- matrix(1:32, 4, 8)
randomize(x)
randomize(x)
```

`residuals` *Residuals in NMF Models*

Description

The package NMF defines methods for the function `residuals` that returns the final residuals of an NMF fit or the track of the residuals along the fit process, computed according to the objective function associated with the algorithm that fitted the model.

`residuals<-` sets the value of the last residuals, or, optionally, of the complete residual track.

Tells if an `NMFfit` object contains a recorded residual track.

`trackError` adds a residual value to the track of residuals.

Usage

```
  residuals(object, ...)

  ## S4 method for signature 'NMFfit'
residuals(object, track = FALSE,
    niter = NULL, ...)

  residuals(object, ...)<-value

  ## S4 replacement method for signature 'NMFfit'
residuals(object, ..., niter = NULL,
    track = FALSE)<-value

  hasTrack(object, niter = NULL)

  trackError(object, value, niter, force = FALSE)
```

Arguments

`object`	an `NMFfit` object as fitted by function `nmf`, in single run mode.
`...`	extra parameters (not used)
`track`	a logical that indicates if the complete track of residuals should be returned (if it has been computed during the fit), or only the last value.

niter	specifies the iteration number for which one wants to get/set/test a residual value. This argument is used only if not NULL
value	residual value
force	logical that indicates if the value should be added to the track even if there already is a value for this iteration number or if the iteration does not conform to the tracking interval nmf.getOption('track.interval').

Details

When called with track=TRUE, the whole residuals track is returned, if available. Note that method nmf does not compute the residuals track, unless explicitly required.

It is a S4 methods defined for the associated generic functions from package stats (See residuals).

Value

residuals returns a single numeric value if track=FALSE or a numeric vector containing the residual values at some iterations. The names correspond to the iterations at which the residuals were computed.

Methods

residuals signature(object = "NMFfit"): Returns the residuals – track – between the target matrix and the NMF fit object.

residuals signature(object = "NMFfitX"): Returns the residuals achieved by the best fit object, i.e. the lowest residual approximation error achieved across all NMF runs.

Note

Stricly speaking, the method residuals,NMFfit does not fulfill its contract as defined by the package stats, but rather acts as function deviance. The might be changed in a later release to make it behave as it should.

See Also

Other stats: deviance, deviance,NMF-method, nmfDistance

---

rmatrix *Generating Random Matrices*

Description

The S4 generic rmatrix generates a random matrix from a given object. Methods are provided to generate matrices with entries drawn from any given random distribution function, e.g. runif or rnorm.

Usage

```
  rmatrix(x, ...)

  ## S4 method for signature 'numeric'
rmatrix(x, y = NULL, dist = runif,
    byrow = FALSE, dimnames = NULL, ...)
```

Arguments

x	object from which to generate a random matrix
y	optional specification of number of columns
dist	a random distribution function or a numeric seed (see details of method rmatrix,numeric)
byrow	a logical passed in the internal call to the function matrix
dimnames	NULL or a list passed in the internal call to the function matrix
...	extra arguments passed to the distribution function dist.

Methods

rmatrix signature(x = "numeric"): Generates a random matrix of given dimensions, whose entries are drawn using the distribution function dist.

This is the workhorse method that is eventually called by all other methods. It returns a matrix with:

- x rows and y columns if y is not missing and not NULL;
- dimension x[1] x x[2] if x has at least two elements;
- dimension x (i.e. a square matrix) otherwise.

The default is to draw its entries from the standard uniform distribution using the base function runif, but any other function that generates random numeric vectors of a given length may be specified in argument dist. All arguments in ... are passed to the function specified in dist.

The only requirement is that the function in dist is of the following form:

'function(n, ...){ # return vector of length n ... }'

This is the case of all base random draw function such as rnorm, rgamma, etc...

rmatrix signature(x = "ANY"): Default method which calls rmatrix,vector on the dimensions of x that is assumed to be returned by a suitable dim method: it is equivalent to rmatrix(dim(x), y=NULL, ...).

rmatrix signature(x = "NMF"): Returns the target matrix estimate of the NMF model x, perturbated by adding a random matrix generated using the default method of rmatrix: it is a equivalent to fitted(x) + rmatrix(fitted(x), ...).

This method can be used to generate random target matrices that depart from a known NMF model to a controlled extend. This is useful to test the robustness of NMF algorithms to the presence of certain types of noise in the data.

Examples

```
#----------
# rmatrix,numeric-method
#----------
## Generate a random matrix of a given size
rmatrix(5, 3)


## Generate a random matrix of the same dimension of a template matrix
a <- matrix(1, 3, 4)
rmatrix(a)


## Specificy the distribution to use

# the default is uniform
a <- rmatrix(1000, 50)
## Not run:  hist(a)

# use normal ditribution
a <- rmatrix(1000, 50, rnorm)
## Not run:  hist(a)

# extra arguments can be passed to the random variate generation function
a <- rmatrix(1000, 50, rnorm, mean=2, sd=0.5)
## Not run:  hist(a)

#----------
# rmatrix,ANY-method
#----------
# random matrix of the same dimension as another matrix
x <- matrix(3,4)
dim(rmatrix(x))

#----------
# rmatrix,NMF-method
#----------
# generate noisy fitted target from an NMF model (the true model)
gr <- as.numeric(mapply(rep, 1:3, 3))
h <- outer(1:3, gr, '==') + 0
x <- rnmf(10, H=h)
y <- rmatrix(x)
## Not run:
# show heatmap of the noisy target matrix: block patterns should be clear
aheatmap(y)

## End(Not run)


# test NMF algorithm on noisy data
# add some noise to the true model (drawn from uniform [0,1])
```

```
res <- nmf(rmatrix(x), 3)
summary(res)

# add more noise to the true model (drawn from uniform [0,10])
res <- nmf(rmatrix(x, max=10), 3)
summary(res)
```

---

rnmf *Generating Random NMF Models*

Description

Generates NMF models with random values drawn from a uniform distribution. It returns an NMF model with basis and mixture coefficient matrices filled with random values. The main purpose of the function rnmf is to provide a common interface to generate random seeds used by the nmf function.

Usage

```
  rnmf(x, target, ...)

  ## S4 method for signature 'NMF,numeric'
rnmf(x, target, ncol = NULL,
    keep.names = TRUE, dist = runif)

  ## S4 method for signature 'ANY,matrix'
rnmf(x, target, ...,
    dist = list(max = max(max(target, na.rm = TRUE), 1)),
    use.dimnames = TRUE)

  ## S4 method for signature 'numeric,missing'
rnmf(x, target, ..., W, H,
    dist = runif)

  ## S4 method for signature 'missing,missing'
rnmf(x, target, ..., W, H)

  ## S4 method for signature 'numeric,numeric'
rnmf(x, target, ncol = NULL,
    ..., dist = runif)

  ## S4 method for signature 'formula,ANY'
rnmf(x, target, ...,
    dist = runif)
```

Arguments

x	an object that determines the rank, dimension and/or class of the generated NMF model, e.g. a numeric value or an object that inherits from class NMF. See the description of the specific methods for more details on the supported types.
target	optional specification of target dimensions. See section *Methods* for how this parameter is used by the different methods.
...	extra arguments to allow extensions and passed to the next method eventually down to nmfModel, where they are used to initialise slots that are specific to the instantiating NMF model.
ncol	single numeric value that specifies the number of columns of the coefficient matrix. Only used when target is a single numeric value.
keep.names	a logical that indicates if the dimension names of the original NMF object x should be conserved (TRUE) or discarded (FALSE).
dist	specification of the random distribution to use to draw the entries of the basis and coefficient matrices. It may be specified as: • a function which must be a distribution function such as e.g. runif that is used to draw the entries of both the basis and coefficient matrices. It is passed in the dist argument of rmatrix. • a list of arguments that are passed internally to rmatrix, via do.call('rmatrix', dist). • a character string that is partially matched to 'basis' or 'coef', that specifies which matrix in should be drawn randomly, the other remaining as in x – unchanged. • a list with elements 'basis' and/or 'coef', which specify the dist argument separately for the basis and coefficient matrix respectively. These elements may be either a distribution function, or a list of arguments that are passed internally to rmatrix, via do.call('rmatrix', dist$basis) or do.call('rmatrix', dist$coef).
use.dimnames	a logical that indicates whether the dimnames of the target matrix should be set on the returned NMF model.
W	value for the basis matrix. data.frame objects are converted into matrices with as.matrix.
H	value for the mixture coefficient matrix data.frame objects are converted into matrices with as.matrix.

Details

If necessary, extensions of the standard NMF model or custom models must define a method "rnmf,<NMF.MODEL.CLASS>,numeric" for initialising their specific slots other than the basis and mixture coefficient matrices. In order to benefit from the complete built-in interface, the overloading methods should call the generic version using function callNextMethod, prior to set the values of the specific slots. See for example the method rnmf defined for NMFOffset models: showMethods(rnmf, class='NMFOffset', include=TRUE)).

For convenience, shortcut methods for working on data.frame objects directly are implemented. However, note that conversion of a data.frame into a matrix object may take some non-negligible

time, for large datasets. If using this method or other NMF-related methods several times, consider converting your data `data.frame` object into a matrix once for good, when first loaded.

Value

An NMF model, i.e. an object that inherits from class `NMF`.

Methods

rnmf `signature(x = "NMFOffset", target = "numeric")`: Generates a random NMF model with offset, from class `NMFOffset`.

The offset values are drawn from a uniform distribution between 0 and the maximum entry of the basis and coefficient matrices, which are drawn by the next suitable `rnmf` method, which is the workhorse method `rnmf,NMF,numeric`.

rnmf `signature(x = "NMF", target = "numeric")`: Generates a random NMF model of the same class and rank as another NMF model.

This is the workhorse method that is eventually called by all other methods. It generates an NMF model of the same class and rank as `x`, compatible with the dimensions specified in `target`, that can be a single or 2-length numeric vector, to specify a square or rectangular target matrix respectively.

The second dimension can also be passed via argument `ncol`, so that calling `rnmf(x, 20, 10, ...)` is equivalent to `rnmf(x, c(20, 10), ...)`, but easier to write.

The entries are uniformly drawn between `0` and `max` (optionally specified in `...`) that defaults to 1.

By default the dimnames of `x` are set on the returned NMF model. This behaviour is disabled with argument `keep.names=FALSE`. See `nmfModel`.

rnmf `signature(x = "ANY", target = "matrix")`: Generates a random NMF model compatible and consistent with a target matrix.

The entries are uniformly drawn between `0` and `max(target)`. It is more or less a shortcut for: '`rnmf(x, dim(target), max=max(target), ...)`'

It returns an NMF model of the same class as `x`.

rnmf `signature(x = "ANY", target = "data.frame")`: Shortcut for `rnmf(x, as.matrix(target))`.

rnmf `signature(x = "NMF", target = "missing")`: Generates a random NMF model of the same dimension as another NMF model.

It is a shortcut for `rnmf(x, nrow(x), ncol(x), ...)`, which returns a random NMF model of the same class and dimensions as `x`.

rnmf `signature(x = "numeric", target = "missing")`: Generates a random NMF model of a given rank, with known basis and/or coefficient matrices.

This methods allow to easily generate partially random NMF model, where one or both factors are known. Although the later case might seems strange, it makes sense for NMF models that have fit extra data, other than the basis and coefficient matrices, that are drawn by an `rnmf` method defined for their own class, which should internally call `rnmf,NMF,numeric` and let it draw the basis and coefficient matrices. (e.g. see `NMFOffset` and `rnmf,NMFOffset,numeric-method`).

Depending on whether arguments `W` and/or `H` are missing, this method interprets `x` differently:

- `W` provided, `H` missing: `x` is taken as the number of columns that must be drawn to build a random coefficient matrix (i.e. the number of columns in the target matrix).

- W is missing, H is provided: x is taken as the number of rows that must be drawn to build a random basis matrix (i.e. the number of rows in the target matrix).
- both W and H are provided: x is taken as the target rank of the model to generate.
- Having both W and H missing produces an error, as the dimension of the model cannot be determined in this case.

The matrices W and H are reduced if necessary and possible to be consistent with this value of the rank, by the internal call to nmfModel.

All arguments in ... are passed to the function nmfModel which is used to build an initial NMF model, that is in turn passed to rnmf,NMF,numeric with dist=list(coef=dist) or dist=list(basis=dist) when suitable. The type of NMF model to generate can therefore be specified in argument model (see nmfModel for other possible arguments).

The returned NMF model, has a basis matrix equal to W (if not missing) and a coefficient matrix equal to H (if not missing), or drawn according to the specification provided in argument dist (see method rnmf,NMF,numeric for details on the supported values for dist).

rnmf signature(x = "missing", target = "missing"): Generates a random NMF model with known basis and coefficient matrices.

This method is a shortcut for calling rnmf,numeric,missing with a suitable value for x (the rank), when both factors are known: rnmf(min(ncol(W), nrow(H)), ..., W=W, H=H).

Arguments W and H are required. Note that calling this method only makes sense for NMF models that contains data to fit other than the basis and coefficient matrices, e.g. NMFOffset.

rnmf signature(x = "numeric", target = "numeric"): Generates a random standard NMF model of given dimensions.

This is a shortcut for rnmf(nmfModel(x, target, ncol, ...)), dist=dist). It generates a standard NMF model compatible with the dimensions passed in target, that can be a single or 2-length numeric vector, to specify a square or rectangular target matrix respectively. See nmfModel.

rnmf signature(x = "formula", target = "ANY"): Generate a random formula-based NMF model, using the method nmfModel,formula,ANY-method.

See Also

rmatrix

Other NMF-interface: basis, .basis, .basis<-, basis<-, coef, .coef, .coef<-, coef<-, coefficients, .DollarNames,NMF-method, loadings,NMF-method, misc, NMF-class, $<-,NMF-method, $,NMF-method, nmfModel, nmfModels, scoef

Examples

```
#----------
# rnmf,NMFOffset,numeric-method
#----------
# random NMF model with offset
x <- rnmf(2, 3, model='NMFOffset')
x
offset(x)
# from a matrix
```

```
x <- rnmf(2, rmatrix(5,3, max=10), model='NMFOffset')
offset(x)

#----------
# rnmf,NMF,numeric-method
#----------
## random NMF of same class and rank as another model

x <- nmfModel(3, 10, 5)
x
rnmf(x, 20) # square
rnmf(x, 20, 13)
rnmf(x, c(20, 13))

# using another distribution
rnmf(x, 20, dist=rnorm)

# other than standard model
y <- rnmf(3, 50, 10, model='NMFns')
y



#----------
# rnmf,ANY,matrix-method
#----------
# random NMF compatible with a target matrix
x <- nmfModel(3, 10, 5)
y <- rmatrix(20, 13)
rnmf(x, y) # rank of x
rnmf(2, y) # rank 2

#----------
# rnmf,NMF,missing-method
#----------
## random NMF from another model

a <- nmfModel(3, 100, 20)
b <- rnmf(a)


#----------
# rnmf,numeric,missing-method
#----------
# random NMF model with known basis matrix
x <- rnmf(5, W=matrix(1:18, 6)) # 6 x 5 model with rank=3
basis(x) # fixed
coef(x) # random

# random NMF model with known coefficient matrix
x <- rnmf(5, H=matrix(1:18, 3)) # 5 x 6 model with rank=3
basis(x) # random
coef(x) # fixed
```

```
# random model other than standard NMF
x <- rnmf(5, H=matrix(1:18, 3), model='NMFOffset')
basis(x) # random
coef(x) # fixed
offset(x) # random

#----------
# rnmf,missing,missing-method
#----------
# random model other than standard NMF
x <- rnmf(W=matrix(1:18, 6), H=matrix(21:38, 3), model='NMFOffset')
basis(x) # fixed
coef(x) # fixed
offset(x) # random

#----------
# rnmf,numeric,numeric-method
#----------
## random standard NMF of given dimensions

# generate a random NMF model with rank 3 that fits a 100x20 matrix
rnmf(3, 100, 20)

# generate a random NMF model with rank 3 that fits a 100x100 matrix
rnmf(3, 100)
```

---

rss *Residual Sum of Squares and Explained Variance*

Description

rss and evar are S4 generic functions that respectively computes the Residual Sum of Squares (RSS) and explained variance achieved by a model.

The explained variance for a target V is computed as:

$$evar = 1 - \frac{RSS}{\sum_{i,j} v_{ij}^2}$$

,

Usage

```
  rss(object, ...)

  ## S4 method for signature 'matrix'
rss(object, target)

  evar(object, ...)
```

```
  ## S4 method for signature 'ANY'
evar(object, target, ...)
```

Arguments

object	an R object with a suitable fitted, rss or evar method.
...	extra arguments to allow extension, e.g. passed to rss in evar calls.
target	target matrix

Details

where RSS is the residual sum of squares.

The explained variance is usefull to compare the performance of different models and their ability to accurately reproduce the original target matrix. Note, however, that a possible caveat is that some models explicitly aim at minimizing the RSS (i.e. maximizing the explained variance), while others do not.

Value

a single numeric value

Methods

evar `signature(object = "ANY")`: Default method for `evar`.

It requires a suitable `rss` method to be defined for `object`, as it internally calls `rss(object, target, ...)`.

rss `signature(object = "matrix")`: Computes the RSS between a target matrix and its estimate `object`, which must be a matrix of the same dimensions as `target`.

The RSS between a target matrix V and its estimate v is computed as:

$$RSS = \sum_{i,j} (v_{ij} - V_{ij})^2$$

Internally, the computation is performed using an optimised C++ implementation, that is light in memory usage.

rss `signature(object = "ANY")`: Residual sum of square between a given target matrix and a model that has a suitable `fitted` method. It is equivalent to `rss(fitted(object), ...)`

In the context of NMF, *Hutchins et al. (2008)* used the variation of the RSS in combination with the algorithm from *Lee et al. (1999)* to estimate the correct number of basis vectors. The optimal rank is chosen where the graph of the RSS first shows an inflexion point, i.e. using a screeplot-type criterium. See section *Rank estimation* in `nmf`.

Note that this way of estimation may not be suitable for all models. Indeed, if the NMF optimisation problem is not based on the Frobenius norm, the RSS is not directly linked to the quality of approximation of the NMF model. However, it is often the case that it still decreases with the rank.

References

Hutchins LN, Murphy SM, Singh P and Graber JH (2008). "Position-dependent motif characterization using non-negative matrix factorization." _Bioinformatics (Oxford, England)_, *24*(23), pp. 2684-90. ISSN 1367-4811, <URL: http://dx.doi.org/10.1093/bioinformatics/btn526>, <URL: http://www.ncbi.nlm.nih.gov/pubmed/18852176>.

Lee DD and Seung HS (1999). "Learning the parts of objects by non-negative matrix factorization." _Nature_, *401*(6755), pp. 788-91. ISSN 0028-0836, <URL: http://dx.doi.org/10.1038/44565>, <URL: http://www.ncbi.nlm.nih.gov/pubmed/10548103>.

Examples

```
#----------
# rss,matrix-method
#----------
# RSS bewteeen random matrices
x <- rmatrix(20,10, max=50)
y <- rmatrix(20,10, max=50)
rss(x, y)
rss(x, x + rmatrix(x, max=0.1))

#----------
# rss,ANY-method
#----------
# RSS between an NMF model and a target matrix
x <- rmatrix(20, 10)
y <- rnmf(3, x) # random compatible model
rss(y, x)

# fit a model with nmf(): one should do better
y2 <- nmf(x, 3) # default minimizes the KL-divergence
rss(y2, x)
y2 <- nmf(x, 3, 'lee') # 'lee' minimizes the RSS
rss(y2, x)
```

`runtime,NMFList-method`

Returns the CPU time required to compute all NMF fits in the list. It returns `NULL` *if the list is empty. If no timing data are available, the sequential time is returned.*

Description

Returns the CPU time required to compute all NMF fits in the list. It returns `NULL` if the list is empty. If no timing data are available, the sequential time is returned.

Usage

```
  ## S4 method for signature 'NMFList'
runtime(object, all = FALSE)
```

Arguments

all	logical that indicates if the CPU time of each fit should be returned (TRUE) or only the total CPU time used to compute all the fits in object.
object	an object computed using some algorithm, or that describes an algorithm itself.

---

runtime.all,NMFfitXn-method

Returns the CPU time used to perform all the NMF fits stored in object.

Description

If no time data is available from in slot 'runtime.all' and argument null=TRUE, then the sequential time as computed by seqtime is returned, and a warning is thrown unless warning=FALSE.

Usage

```
  ## S4 method for signature 'NMFfitXn'
runtime.all(object, null = FALSE,
    warning = TRUE)
```

Arguments

null	a logical that indicates if the sequential time should be returned if no time data is available in slot 'runtime.all'.
warning	a logical that indicates if a warning should be thrown if the sequential time is returned instead of the real CPU time.
object	an object computed using some algorithm, or that describes an algorithm itself.

---

scale.NMF *Rescaling NMF Models*

Description

Rescales an NMF model keeping the fitted target matrix identical.

Usage

```
  ## S3 method for class 'NMF'
scale(x, center = c("basis", "coef"),
   scale = 1)
```

Arguments

x	an NMF object
center	either a numeric normalising vector δ, or either 'basis' or 'coef', which respectively correspond to using the column sums of the basis matrix or the inverse of the row sums of the coefficient matrix as a normalising vector. If numeric, center should be a single value or a vector of length the rank of the NMF model, i.e. the number of columns in the basis matrix.
scale	scaling coefficient applied to D, i.e. the value of α, or, if center='coef', the value of $1/\alpha$ (see section *Details*).

Details

Standard NMF models are identifiable modulo a scaling factor, meaning that the basis components and basis profiles can be rescaled without changing the fitted values:

$$X = W_1 H_1 = (W_1 D)(D^{-1} H_1) = W_2 H_2$$

with $D = \alpha diag(1/\delta_1, \ldots, 1\delta_r)$

The default call scale(object) rescales the basis NMF object so that each column of the basis matrix sums up to one.

Value

an NMF object

Examples

```
# random 3-rank 10x5 NMF model
x <- rnmf(3, 10, 5)

# rescale based on basis
colSums(basis(x))
colSums(basis(scale(x)))

rx <- scale(x, 'basis', 10)
colSums(basis(rx))
rowSums(coef(rx))

# rescale based on coef
rowSums(coef(x))
rowSums(coef(scale(x, 'coef')))
rx <- scale(x, 'coef', 10)
rowSums(coef(rx))
colSums(basis(rx))

# fitted target matrix is identical but the factors have been rescaled
rx <- scale(x, 'basis')
all.equal(fitted(x), fitted(rx))
all.equal(basis(x), basis(rx))
```

seed *Interface for NMF Seeding Methods*

Description

The function seed provides a single interface for calling all seeding methods used to initialise NMF computations. These methods at least set the basis and coefficient matrices of the initial object to valid nonnegative matrices. They will be used as a starting point by any NMF algorithm that accept initialisation.

IMPORTANT: this interface is still considered experimental and is subject to changes in future release.

Usage

```
  seed(x, model, method, ...)

  ## S4 method for signature 'matrix,NMF,NMFSeed'
seed(x, model, method,
    rng, ...)

  ## S4 method for signature 'ANY,ANY,function'
seed(x, model, method, name,
    ...)
```

Arguments

x	target matrix one wants to approximate with NMF
model	specification of the NMF model, e.g., the factorization rank.
method	specification of a seeding method. See each method for details on the supported formats.
...	extra to allow extensions and passed down to the actual seeding method.
rng	rng setting to use. If not missing the RNG settings are set and restored on exit using setRNG. All arguments in ... are passed to teh seeding strategy.
name	optional name of the seeding method for custom seeding strategies.

Value

an NMFfit object.

Methods

seed signature(x = "matrix", model = "NMF", method = "NMFSeed"): This is the workhorse method that seeds an NMF model object using a given seeding strategy defined by an NMFSeed object, to fit a given target matrix.

seed `signature(x = "ANY", model = "ANY", method = "function")`: Seeds an NMF model using a custom seeding strategy, defined by a function.

`method` must have signature `(x='NMFfit', y='matrix', ...)`, where `x` is the unseeded NMF model and `y` is the target matrix to fit. It must return an `NMF` object, that contains the seeded NMF model.

seed `signature(x = "ANY", model = "ANY", method = "missing")`: Seeds the model with the default seeding method given by `nmf.getOption('default.seed')`

seed `signature(x = "ANY", model = "ANY", method = "NULL")`: Use NMF method `'none'`.

seed `signature(x = "ANY", model = "ANY", method = "numeric")`: Use `method` to set the RNG with `setRNG` and use method "random" to seed the NMF model.

Note that in this case the RNG settings are not restored. This is due to some internal technical reasons, and might change in future releases.

seed `signature(x = "ANY", model = "ANY", method = "character")`: Use the registered seeding method whose access key is `method`.

seed `signature(x = "ANY", model = "list", method = "NMFSeed")`: Seed a model using the elements in `model` to instantiate it with `nmfModel`.

seed `signature(x = "ANY", model = "numeric", method = "NMFSeed")`: Seeds a standard NMF model (i.e. of class `NMFstd`) of rank `model`.

`setNMFMethod` *Registering NMF Algorithms*

Description

Adds a new algorithm to the registry of algorithms that perform Nonnegative Matrix Factorization.

`nmfRegisterAlgorithm` is an alias to `setNMFMethod` for backward compatibility.

Usage

```
setNMFMethod(name, method, ...,
  overwrite = isLoadingNamespace(), verbose = TRUE)

nmfRegisterAlgorithm(name, method, ...,
  overwrite = isLoadingNamespace(), verbose = TRUE)
```

Arguments

`...`	arguments passed to the factory function `NMFStrategy`, which instantiate the `NMFStrategy` object that is stored in registry.
`overwrite`	logical that indicates if any existing NMF method with the same name should be overwritten (`TRUE`) or not (`FALSE`), in which case an error is thrown.
`verbose`	a logical that indicates if information about the registration should be printed (`TRUE`) or not (`FALSE`).
`name`	name/key of an NMF algorithm.
`method`	definition of the algorithm

Examples

```
# define/regsiter a new -- dummy -- NMF algorithm with the minimum arguments
# y: target matrix
# x: initial NMF model (i.e. the seed)
# NB: this algorithm simply return the seed unchanged
setNMFMethod('mynmf', function(y, x, ...){ x })

# check algorithm on toy data
res <- nmfCheck('mynmf')
# the NMF seed is not changed
stopifnot( nmf.equal(res, nmfCheck('mynmf', seed=res)) )
```

---

setupBackend *Computational Setup Functions*

Description

Functions used internally to setup the computational environment.

`setupBackend` sets up a foreach backend given some specifications.

`setupSharedMemory` checks if one can use the packages *bigmemory* and *sychronicity* to speed-up parallel computations when not keeping all the fits. When both these packages are available, only one result per host is written on disk, with its achieved deviance stored in shared memory, that is accessible to all cores on a same host. It returns `TRUE` if both packages are available and NMF option `'shared'` is toggled on.

`setupTempDirectory` creates a temporary directory to store the best fits computed on each host. It ensures each worker process has access to it.

`setupLibPaths` add the path to the NMF package to each workers' libPaths.

`setupRNG` sets the RNG for use by the function nmf. It returns the old RNG as an rstream object or the result of set.seed if the RNG is not changed due to one of the following reason: - the settings are not compatible with rstream

Usage

```
setupBackend(spec, backend, optional = FALSE,
  verbose = FALSE)

setupSharedMemory(verbose)

setupTempDirectory(verbose)

setupLibPaths(pkg = "NMF", verbose = FALSE)

setupRNG(seed, n, verbose = FALSE)
```

Arguments

spec	target parallel specification: either TRUE or FALSE, or a single numeric value that specifies the number of cores to setup.
backend	value from argument .pbackend of nmf.
optional	a logical that indicates if the specification must be fully satisfied, throwing an error if it is not, or if one can switch back to sequential, only outputting a verbose message.
verbose	logical or integer level of verbosity for message outputs.
pkg	package name whose path should be exported the workers.
seed	initial RNG seed specification
n	number of RNG seeds to generate

Value

Returns FALSE if no foreach backend is to be used, NA if the currently registered backend is to be used, or, if this function call registered a new backend, the previously registered backend as a foreach object, so that it can be restored after the computation is over.

---

show,NMF-method *Show method for objects of class* NMF

Description

Show method for objects of class NMF

Usage

```
  ## S4 method for signature 'NMF'
show(object)
```

Arguments

object	Any R object

show,NMFfit-method *Show method for objects of class* NMFfit

Description

Show method for objects of class NMFfit

Usage

```
  ## S4 method for signature 'NMFfit'
show(object)
```

Arguments

object	Any R object

show,NMFfitX-method *Show method for objects of class* NMFfitX

Description

Show method for objects of class NMFfitX

Usage

```
  ## S4 method for signature 'NMFfitX'
show(object)
```

Arguments

object	Any R object

show,NMFfitX1-method *Show method for objects of class* NMFfitX1

Description

Show method for objects of class NMFfitX1

Usage

```
  ## S4 method for signature 'NMFfitX1'
show(object)
```

Arguments

object	Any R object

show,NMFfitXn-method *Show method for objects of class* NMFfitXn

Description

Show method for objects of class NMFfitXn

Usage

```
  ## S4 method for signature 'NMFfitXn'
show(object)
```

Arguments

object	Any R object

show,NMFList-method *Show method for objects of class* NMFList

Description

Show method for objects of class NMFList

Usage

```
  ## S4 method for signature 'NMFList'
show(object)
```

Arguments

object	Any R object

show,NMFns-method *Show method for objects of class* NMFns

Description

Show method for objects of class NMFns

Usage

```
  ## S4 method for signature 'NMFns'
show(object)
```

Arguments

object	Any R object

show,NMFOffset-method *Show method for objects of class* NMFOffset

Description

Show method for objects of class NMFOffset

Usage

```
  ## S4 method for signature 'NMFOffset'
show(object)
```

Arguments

object	Any R object

show,NMFSeed-method *Show method for objects of class* NMFSeed

Description

Show method for objects of class NMFSeed

Usage

```
  ## S4 method for signature 'NMFSeed'
show(object)
```

Arguments

object	Any R object

`show,NMFStrategyIterative-method` *Show method for objects of class* `NMFStrategyIterative`

Description

Show method for objects of class `NMFStrategyIterative`

Usage

```
  ## S4 method for signature 'NMFStrategyIterative'
show(object)
```

Arguments

`object`	Any R object

`silhouette.NMF` *Silhouette of NMF Clustering*

Description

Silhouette of NMF Clustering

Usage

```
  ## S3 method for class 'NMF'
silhouette(x, what = NULL, order = NULL,
   ...)
```

Arguments

`x`	an NMF object, as returned by `nmf`.
`what`	defines the type of clustering the computed silhouettes are meant to assess: `'samples'` for the clustering of samples (i.e. the columns of the target matrix), `'features'` for the clustering of features (i.e. the rows of the target matrix), and `'chc'` for the consensus clustering of samples as defined by hierarchical clustering dendrogram, `'consensus'` for the consensus clustering of samples, with clustered ordered as in the **default** hierarchical clustering used by `consensusmap` when plotting the heatmap of the consensus matrix (for multi-run NMF fits). That is `dist = 1 - consensus(x)`, average linkage and reordering based on row means.
`order`	integer indexing vector that can be used to force the silhouette order.
`...`	extra arguments not used.

See Also

predict

Examples

```
x <- rmatrix(75, 15, dimnames = list(paste0('a', 1:75), letters[1:15]))
# NB: using low value for maxIter for the example purpose only
res <- nmf(x, 4, nrun = 3, maxIter = 20)

# sample clustering from best fit
plot(silhouette(res))

# average silhouette are computed in summary measures
summary(res)

# consensus silhouettes are ordered as on default consensusmap heatmap
## Not run:  op <- par(mfrow = c(1,2))
consensusmap(res)
si <- silhouette(res, what = 'consensus')
plot(si)
## Not run:  par(op)

# if the order is based on some custom numeric weights
## Not run:  op <- par(mfrow = c(1,2))
cm <- consensusmap(res, Rowv = runif(ncol(res)))
# NB: use reverse order because silhouettes are plotted top-down
si <- silhouette(res, what = 'consensus', order = rev(cm$rowInd))
plot(si)
## Not run:  par(op)

# do the reverse: order the heatmap as a set of silhouettes
si <- silhouette(res, what = 'features')
## Not run:  op <- par(mfrow = c(1,2))
basismap(res, Rowv = si)
plot(si)
## Not run:  par(op)
```

---

smoothing *Smoothing Matrix in Nonsmooth NMF Models*

Description

The function smoothing builds a smoothing matrix for using in Nonsmooth NMF models.

Usage

```
smoothing(x, theta = x@theta, ...)
```

Arguments

x	a object of class NMFns.
theta	the smoothing parameter (numeric) between 0 and 1.
...	extra arguments to allow extension (not used)

Details

For a r-rank NMF, the smoothing matrix of parameter θ is built as follows:

$$S = (1 - \theta)I + \frac{\theta}{r}11^T,$$

where I is the identity matrix and 1 is a vector of ones (cf. NMFns-class for more details).

Value

if x estimates a r-rank NMF, then the result is a $r \times r$ square matrix.

Examples

```
x <- nmfModel(3, model='NMFns')
smoothing(x)
smoothing(x, 0.1)
```

---

sparseness *Sparseness*

Description

Generic function that computes the *sparseness* of an object, as defined by *Hoyer (2004)*. The sparseness quantifies how much energy of a vector is packed into only few components.

Usage

```
sparseness(x, ...)
```

Arguments

x	an object whose sparseness is computed.
...	extra arguments to allow extension

Details

In *Hoyer (2004)*, the sparseness is defined for a real vector x as:

$$Sparseness(x) = \frac{\sqrt{n} - \frac{\sum |x_i|}{\sqrt{\sum x_i^2}}}{\sqrt{n} - 1}$$

, where n is the length of x.

The sparseness is a real number in $[0, 1]$. It is equal to 1 if and only if `x` contains a single nonzero component, and is equal to 0 if and only if all components of `x` are equal. It interpolates smoothly between these two extreme values. The closer to 1 is the sparseness the sparser is the vector.

The basic definition is for a `numeric` vector, and is extended for matrices as the mean sparseness of its column vectors.

Value

usually a single numeric value – in [0,1], or a numeric vector. See each method for more details.

Methods

sparseness `signature(x = "numeric")`: Base method that computes the sparseness of a numeric vector.

It returns a single numeric value, computed following the definition given in section *Description*.

sparseness `signature(x = "matrix")`: Computes the sparseness of a matrix as the mean sparseness of its column vectors. It returns a single numeric value.

sparseness `signature(x = "NMF")`: Compute the sparseness of an object of class `NMF`, as the sparseness of the basis and coefficient matrices computed separately.

It returns the two values in a numeric vector with names 'basis' and 'coef'.

References

Hoyer P (2004). "Non-negative matrix factorization with sparseness constraints." _The Journal of Machine Learning Research_, *5*, pp. 1457-1469. <URL: http://portal.acm.org/citation.cfm?id=1044709>.

See Also

Other assess: `entropy`, `purity`

staticVar *Get/Set a Static Variable in NMF Algorithms*

Description

This function is used in iterative NMF algorithms to manage variables stored in a local workspace, that are accessible to all functions that define the iterative schema described in NMFStrategyIterative.

It is specially useful for computing stopping criteria, which often require model data from different iterations.

Usage

```
staticVar(name, value, init = FALSE)
```

Arguments

name	Name of the static variable (as a single character string)
value	New value of the static variable
init	a logical used when a value is provided, that specifies if the variable should be set to the new value only if it does not exist yet (init=TRUE).

Value

The value of the static variable

summary *Assessing and Comparing NMF Models*

Description

The NMF package defines summary methods for different classes of objects, which helps assessing and comparing the quality of NMF models by computing a set of quantitative measures, e.g. with respect to their ability to recover known classes and/or the original target matrix.

The most useful methods are for classes NMF, NMFfit, NMFfitX and NMFList, which compute summary measures for, respectively, a single NMF model, a single fit, a multiple-run fit and a list of heterogenous fits performed with the function nmf.

Usage

```
summary(object, ...)

## S4 method for signature 'NMF'
summary(object, class, target)
```

Arguments

`object`	an NMF object. See available methods in section *Methods*.
`...`	extra arguments passed to the next `summary` method.
`class`	known classes/cluster of samples specified in one of the formats that is supported by the functions `entropy` and `purity`.
`target`	target matrix specified in one of the formats supported by the functions `rss` and `evar`

Details

Due to the somehow hierarchical structure of the classes mentionned in *Description*, their respective `summary` methods call each other in chain, each super-class adding some extra measures, only relevant for objects of a specific class.

Methods

summary `signature(object = "NMF")`: Computes summary measures for a single NMF model.

The following measures are computed:

sparseness Sparseness of the factorization computed by the function `sparseness`.

entropy Purity of the clustering, with respect to known classes, computed by the function `purity`.

entropy Entropy of the clustering, with respect to known classes, computed by the function `entropy`.

RSS Residual Sum of Squares computed by the function `rss`.

evar Explained variance computed by the function `evar`.

summary `signature(object = "NMFfit")`: Computes summary measures for a single fit from `nmf`.

This method adds the following measures to the measures computed by the method `summary,NMF`:

residuals Residual error as measured by the objective function associated to the algorithm used to fit the model.

niter Number of iterations performed to achieve convergence of the algorithm.

cpu Total CPU time required for the fit.

cpu.all Total CPU time required for the fit. For `NMFfit` objects, this element is always equal to the value in "cpu", but will be different for multiple-run fits.

nrun Number of runs performed to fit the model. This is always equal to 1 for `NMFfit` objects, but will vary for multiple-run fits.

summary `signature(object = "NMFfitX")`: Computes a set of measures to help evaluate the quality of the *best fit* of the set. The result is similar to the result from the `summary` method of `NMFfit` objects. See `NMF` for details on the computed measures. In addition, the cophenetic correlation (`cophcor`) and `dispersion` coefficients of the consensus matrix are returned, as well as the total CPU time (`runtime.all`).

Examples

```
#----------
# summary,NMF-method
#----------
# random NMF model
x <- rnmf(3, 20, 12)
summary(x)
summary(x, gl(3, 4))
summary(x, target=rmatrix(x))
summary(x, gl(3,4), target=rmatrix(x))

#----------
# summary,NMFfit-method
#----------
# generate a synthetic dataset with known classes: 50 features, 18 samples (5+5+8)
n <- 50; counts <- c(5, 5, 8);
V <- syntheticNMF(n, counts)
cl <- unlist(mapply(rep, 1:3, counts))

# perform default NMF with rank=2
x2 <- nmf(V, 2)
summary(x2, cl, V)
# perform default NMF with rank=2
x3 <- nmf(V, 3)
summary(x2, cl, V)
```

---

syntheticNMF *Simulating Datasets*

Description

The function `syntheticNMF` generates random target matrices that follow some defined NMF model, and may be used to test NMF algorithms. It is designed to designed to produce data with known or clear classes of samples.

Usage

```
syntheticNMF(n, r, p, offset = NULL, noise = TRUE,
  factors = FALSE, seed = NULL)
```

Arguments

`n`	number of rows of the target matrix.
`r`	specification of the factorization rank. It may be a single `numeric`, in which case argument `p` is required and `r` groups of samples are generated from a draw from a multinomial distribution with equal probabilities, that provides their sizes.

	It may also be a numerical vector, which contains the number of samples in each class (i.e integers). In this case argument p is discarded and forced to be the sum of r.
p	number of columns of the synthetic target matrix. Not used if parameter r is a vector (see description of argument r).
offset	specification of a common offset to be added to the synthetic target matrix, before noisification. Its may be a numeric vector of length n, or a single numeric value that is used as the standard deviation of a centred normal distribution from which the actual offset values are drawn.
noise	a logical that indicate if noise should be added to the matrix.
factors	a logical that indicates if the NMF factors should be return together with the matrix.
seed	a single numeric value used to seed the random number generator before generating the matrix. The state of the RNG is restored on exit.

Value

a matrix, or a list if argument factors=TRUE.

When factors=FALSE, the result is a matrix object, with the following attributes set:

coefficients the true underlying coefficient matrix (i.e. H);

basis the true underlying coefficient matrix (i.e. H);

offset the offset if any;

pData a list with one element 'Group' that contains a factor that indicates the true groups of samples, i.e. the most contributing basis component for each sample;

fData a list with one element 'Group' that contains a factor that indicates the true groups of features, i.e. the basis component to which each feature contributes the most.

Moreover, the result object is an ExposeAttribute object, which means that relevant attributes are accessible via $, e.g., res$coefficients. In particular, methods coef and basis will work as expected and return the true underlying coefficient and basis matrices respectively.

Examples

```
# generate a synthetic dataset with known classes: 50 features, 18 samples (5+5+8)
n <- 50
counts <- c(5, 5, 8)

# no noise
V <- syntheticNMF(n, counts, noise=FALSE)
## Not run: aheatmap(V)

# with noise
V <- syntheticNMF(n, counts)
## Not run: aheatmap(V)
```

t.NMF *Transformation NMF Model Objects*

Description

t transpose an NMF model, by transposing and swapping its basis and coefficient matrices: $t([W,H]) = [t(H), t(W)]$.

Usage

```
 ## S3 method for class 'NMF'
t(x)
```

Arguments

x	NMF model object.

Details

The function t is a generic defined in the **base** package. The method t.NMF defines the trasnformation for the general NMF interface. This method may need to be overloaded for NMF models, whose structure requires specific handling.

See Also

Other transforms: nneg, posneg, rposneg

Examples

```
x <- rnmf(3, 100, 20)
x
# transpose
y <- t(x)
y

# factors are swapped-transposed
stopifnot( identical(basis(y), t(coef(x))) )
stopifnot( identical(coef(y), t(basis(x))) )
```

utils-NMF *Utility Function in the NMF Package*

Description

Utility Function in the NMF Package

`str_args` formats the arguments of a function using `args`, but returns the output as a string.

Usage

```
str_args(x, exdent = 10L)
```

Arguments

`x`	a function
`exdent`	indentation for extra lines if the output takes more than one line.

Examples

```
args(library)
str_args(library)
```

[,NMF-method *Sub-setting NMF Objects*

Description

This method provides a convenient way of sub-setting objects of class `NMF`, using a matrix-like syntax.

It allows to consistently subset one or both matrix factors in the NMF model, as well as retrieving part of the basis components or part of the mixture coefficients with a reduced amount of code.

Usage

```
## S4 method for signature 'NMF'
x[i, j, ..., drop = FALSE]
```

Arguments

i	index used to subset on the **rows** of the basis matrix (i.e. the features). It can be a numeric, logical, or character vector (whose elements must match the row names of x). In the case of a logical vector the entries are recycled if necessary.
j	index used to subset on the **columns** of the mixture coefficient matrix (i.e. the samples). It can be a numeric, logical, or character vector (whose elements must match the column names of x). In the case of a logical vector the entries are recycled if necessary.
...	used to specify a third index to subset on the basis components, i.e. on both the columns and rows of the basis matrix and mixture coefficient respectively. It can be a numeric, logical, or character vector (whose elements must match the basis names of x). In the case of a logical vector the entries are recycled if necessary. Note that only the first extra subset index is used. A warning is thrown if more than one extra argument is passed in ....
drop	single logical value used to drop the NMF-class wrapping and only return subsets of one of the factor matrices (see *Details*)
x	object from which to extract element(s) or in which to replace element(s).

Details

The returned value depends on the number of subset index passed and the value of argument drop:

- No index as in x[] or x[,]: the value is the object x unchanged.
- One single index as in x[i]: the value is the complete NMF model composed of the selected basis components, subset by i, except if argument drop=TRUE, or if it is missing and i is of length 1. Then only the basis matrix is returned with dropped dimensions: x[i, drop=TRUE] <=> drop(basis(x)[, i]).

  This means for example that x[1L] is the first basis vector, and x[1:3, drop = TRUE] is the matrix composed of the 3 first basis vectors – in columns.

  Note that in version <= 0.18.3, the call x[i, drop = TRUE.or.FALSE] was equivalent to basis(x)[, i, drop=TRUE.or.FALSE].
- More than one index with drop=FALSE (default) as in x[i,j], x[i,], x[,j], x[i,j,k], x[i,,k], etc...: the value is a NMF object whose basis and/or mixture coefficient matrices have been subset accordingly. The third index k affects simultaneously the columns of the basis matrix AND the rows of the mixture coefficient matrix. In this case argument drop is not used.
- More than one index with drop=TRUE and i xor j missing: the value returned is the matrix that is the more affected by the subset index. That is that x[i, , drop=TRUE] and x[i, , k, drop=TRUE] return the basis matrix subset by [i,] and [i,k] respectively, while x[, j, drop=TRUE] and x[, j, k, drop=TRUE] return the mixture coefficient matrix subset by [,j] and [k,j] respectively.

Examples

```
# create a dummy NMF object that highlight the different way of subsetting
a <- nmfModel(W=outer(seq(1,5),10^(0:2)), H=outer(10^(0:2),seq(-1,-10)))
basisnames(a) <- paste('b', 1:nbasis(a), sep='')
rownames(a) <- paste('f', 1:nrow(a), sep='')
colnames(a) <- paste('s', 1:ncol(a), sep='')

# or alternatively:
# dimnames(a) <- list( features=paste('f', 1:nrow(a), sep='')
# , samples=paste('s', 1:ncol(a), sep='')
# , basis=paste('b', 1:nbasis(a)) )

# look at the resulting NMF object
a
basis(a)
coef(a)

# extract basis components
a[1]
a[1, drop=FALSE] # not dropping matrix dimension
a[2:3]

# subset on the features
a[1,]
a[2:4,]
# dropping the NMF-class wrapping => return subset basis matrix
a[2:4,, drop=TRUE]

# subset on the samples
a[,1]
a[,2:4]
# dropping the NMF-class wrapping => return subset coef matrix
a[,2:4, drop=TRUE]

# subset on the basis => subsets simultaneously basis and coef matrix
a[,,1]
a[,,2:3]
a[4:5,,2:3]
a[4:5,,2:3, drop=TRUE] # return subset basis matrix
a[,4:5,2:3, drop=TRUE] # return subset coef matrix

# 'drop' has no effect here
a[,,2:3, drop=TRUE]
```

Index