Package 'qlcMatrix'

October 13, 2022

Type Package

Title Utility Sparse Matrix Functions for Quantitative Language Comparison

Version 0.9.7

Date 2018-04-19

Author Michael Cysouw

Maintainer Michael Cysouw <cysouw@mac.com>

Description Extension of the functionality of the Matrix package for using sparse matrices. Some of the functions are very general, while other are highly specific for special data format as used for quantitative language comparison (QLC).

License GPL-3

Encoding UTF-8

Depends Matrix (>= 1.1-0), R (>= 3.2), slam (>= 0.1-32), sparsesvd

Imports methods, docopt

Suggests MASS, knitr

VignetteBuilder knitr

RoxygenNote 6.0.1

NeedsCompilation no

Repository CRAN

Date/Publication 2018-04-20 07:03:04 UTC

R topics documented:

`qlcMatrix-package` *Utility sparse matrix functions for Quantitative Language Comparison (QLC)*

Description

This package contains various functions that extend the functionality of the `Matrix` package for using sparse matrices. Some of the functions are very general, while other are highly specific for special data format as used for quantitative language comparison.

Details

Package:	qlcMatrix
Type:	Package
Version:	0.9.7
Date:	2018-04-19
License:	GPL-3

This package contains various different kinds of function.

First, some general utility functions to deal with sparse matrices: (i) `rowMax` to compute and identify row-wise maxima and minima in sparse matrices, (ii) `rKhatriRao` to remove empty rows in a Khatri-Rao product (but still get the right rownames) and (iii) `rSparseMatrix` to produce random sparse

matrices. There are also some experimental basic methods for handling sparse arrays ("tensors"), most interestingly unfold.

Second, some general functions to compute associations between the columns of sparse matrices, with possibilities for extension for ad-hoc measures: cosSparse, corSparse, and assocSparse There are special versions of these for nominal data cosNominal, assocNominal.

Third, there are three central functions needed to efficiently turn data from quantitative language comparison into sparse matrices. These basic functions are then used by high-level function in this package. Although these functions might seem almost trivial, they form the basis for many highly complex computations. They are ttMatrix, pwMatrix and jMatrix.

Fourth, there are some high-level convenience function that take specific data formats from quantitative language comparison and turn them into set of sparse matrices for efficient computations. They might also be useful for other data types, but various details decisions are specifically tailored to the envisioned data types. These functions are splitTable splitStrings, splitWordlist, and splitText.

Finally, there are various shortcuts to directly compute similarity matrices from various kinds of data: sim.nominal, sim.words, sim.strings, sim.wordlist. These are specifically tailored towards specific kinds of data, though they might also be useful elsewhere. Also, the code is mostly easy wrappers around the split and cos/assoc functions, so it should not be difficult to adapt these functions to other needs.

Author(s)

Michael Cysouw <cysouw@mac.com>

References

Cysouw, Michael. 2014. *Matrix Algebra for Language Comparison*. Manuscript.

Mayer, Thomas and Michael Cysouw. 2012. Language comparison through sparse multilingual word alignment. *Proceedings of the EACL 2012 Joint Workshop of LINGVIS & UNCLH*, 54–62. Avignon: Association for Computational Linguistics.

Prokić, Jelena and Michael Cysouw. 2013. Combining regular sound correspondences and geographic spread. *Language Dynamics and Change* 3(2). 147–168.

Array *Sparse Arrays ("Tensors")*

Description

Convenient function linking sparse Arrays from the package spam to the sparse Matrices from the package Matrix.

Usage

```
Array(A)
sparseArray(i, v = NULL, ...)

as.Matrix(M)
```

Arguments

A	An array to be turned into a sparse Array using as.simple_sparse_array. Can also be a dataframe, but see Details below about the treatment of data frames here.
i	Integer matrix of array indices passed to simple_sparse_array.
v	vector of values passed to simple_sparse_array. If NULL (by default), all specified indices (i.e. all rows in i) are given the value 1.
M	Matrix of type simple_triple_matrix from the package spam to be turned into a TsparseMatrix from the packages Matrix.
...	Further arguments passed to simple_sparse_array.

Details

Array turns an array into a sparse Array. There is a special behavior when a dataframe is supplied. Such a dataframe is treated as 'long format', i.e. the columns of the dataframe are treated as dimensions of the Array, and all rows of the dataframe are interpreted as entries. The coordinates are given by the ordering of the levels in the dataframe, and the dimnames are given by the levels.

sparseArray constructs sparse Arrays from a matrix of indices and a vector of values. dim and dimnames can be added as in simple_sparse_array

as.Matrix turns a simple_triplet_matrix into a dgTMatrix.

Value

Sparse Arrays use the class "simple_sparse_array" from spam

Note

These functions are only an example of how spam can be linked to Matrix.

Author(s)

Michael Cysouw

Examples

```
x <- matrix(c(1, 0, 0, 2), nrow = 2)
s <- as.simple_triplet_matrix(x)
str(s)

as.Matrix(s)
str(as.Matrix(s))
```

assocSparse *Association between columns (sparse matrices)*

Description

This function offers an interface to various different measures of association between columns in sparse matrices (based on functions of 'observed' and 'expected' values). Currently, the following measures are available: pointwise mutual information (aka log-odds), a poisson-based measure and Pearson residuals. Further measures can easily be specifically defined by the user. The calculations are optimized to be able to deal with large sparse matrices. Note that these association values are really only (sensibly) defined for binary data.

Usage

```
assocSparse(X, Y = NULL, method = res, N = nrow(X), sparse = TRUE )
```

Arguments

X	a sparse matrix in a format of the Matrix package, typically a dgCMatrix with only zeros or ones. The association will be calculated between the columns of this matrix.
Y	a second matrix in a format of the Matrix package with the same number of rows as X. When Y=NULL, then the associations between the columns of X and itself will be taken. If Y is specified, the association between the columns of X and the columns of Y will be calculated.
method	The method to be used for the calculation. Currently res (residuals), poi (poisson), pmi (pointwise mutual information) and wpmi (weighted pointwise mutual information) are available, but further methods can be specified by the user. See details for more information.
N	Variable that is needed for the calculations of the expected values. Only in exceptional situations this should be different from the default value (i.e. the number of rows of the matrix).
sparse	By default, nothing is computed when the observed co-occurrence of two columns is zero. This keeps the computations and the resulting matrix nicely sparse. However, for some measures (specifically the Pearson residuals 'res') this leads to incorrect results. Mostly the error is negligible, but if the correct behavior is necessary, chose sparse = F. Note that the result will then be a full matrix, so this is not feasible for large datasets.

Details

Computations are based on a comparison of the observed interaction crossprod(X,Y) and the expected interaction. Expectation is in principle computed as tcrossprod(rowSums(abs(X)),rowSums(abs(Y)))/nrow(X), though in practice the code is more efficient than that.

Note that calculating the observed interaction as crossprod(X,Y) really only makes sense for binary data (i.e. matrices with only ones and zeros). Currently, all input is coerced to such data by

`as(X, "nMatrix")*1`, meaning that all values that are not one or zero are turned into one (including negative values!).

Any method can be defined as a function with two arguments, `o` and `e`, e.g. simply by specifying `method = function(o,e){o/e}`. See below for more examples.

The predefined functions are:

- `pmi`: pointwise mutual information, aka as log-odds in bioinformatics, defined as `pmi <- function(o,e) { log(o/e) }`.
- `wpmi`: weighted pointwise mutual information, defined as `wpmi <- function(o,e) { o * log(o/e) }`.
- `res`: Pearson residuals, defined as `res <- function(o,e) { (o-e) / sqrt(e) }`.
- `poi`: association assuming a poisson-distribution of the values, defined as `poi <- function(o,e) { sign(o-e) * (o * log(o/e) - (o-e)) }`. Seems to be very useful when the non-zero data is strongly skewed along the rows, i.e. some rows are much fuller than others. A short explanation of this method can be found in Prokić and Cysouw (2013).

Value

The result is a sparse matrix with the non-zero association values. Values range between -Inf and +Inf, with values close to zero indicating low association. The exact interpretation of the values depends on the method used.

When `Y = NULL`, then the result is a symmetric matrix, so a matrix of type `dsCMatrix` with size `ncol(X)` by `ncol{X}` is returned. When `X` and `Y` are both specified, a matrix of type `dgCMatrix` with size `ncol(X)` by `ncol{Y}` is returned.

Note

Care is taken in the implementation not to compute any association between columns that will end up with a value of zero anyway. However, very small association values will be computed. For further usage, these small values are often unnecessary, and can be removed for reasons of sparsity. Consider something like `X <- drop0(X, tol = value)` on the resulting `X` matrix (which removes all values between -value and +value). See examples below.

It is important to realize, that by default noting is computed when the observed co-occurrence is zero. However, this leads to wrong results with `method = res`, as `(o-e)/sqrt(e)` will be a negative value when `o = 0`. In most practically situations this error will be small and not important. However, when needed, the option `sparse = F` will give the correct results (though the resulting matrix will not be sparse anymore). Note that with all other methods implemented here, the default behavior leads to correct results (i.e. for `log(0)` nothing is calculated).

The current implementation will not lead to correct results with lots of missing data (that option is simply not yet implemented). See `cosMissing` for now.

Author(s)

Michael Cysouw

References

Prokić, Jelena & Michael Cysouw. 2013. Combining regular sound correspondences and geographic spread. *Language Dynamics and Change* 3(2). 147–168.

See Also

See assocCol and assocRow for this measure defined for nominal data. Also, see corSparse and cosSparse for other sparse association measures.

Examples

```
# ----- reasonably fast with large very sparse matrices -----

X <- rSparseMatrix(1e6, 1e6, 1e6, NULL)
system.time(M <- assocSparse(X, method = poi))
length(M@x) / prod(dim(M)) # only one in 1e6 cells non-zero

## Not run:

# ----- reaching limits of sparsity -----

# watch out:
# with slightly less sparse matrices the result will not be very sparse,
# so this will easily fill up your RAM during computation!

X <- rSparseMatrix(1e4, 1e4, 1e6, NULL)
system.time(M <- assocSparse(X, method = poi))
print(object.size(M), units = "auto") # about 350 Mb
length(M@x) / prod(dim(M)) # 30% filled

# most values are low, so it often makes sense
# to remove low values to keep results sparse

M <- drop0(M, tol = 2)
print(object.size(M), units = "auto") # reduces to 10 Mb
length(M@x) / prod(dim(M)) # down to less than 1% filled

## End(Not run)

# ----- defining new methods -----

# Using the following simple 'div' method is the same as
# using a cosine similarity with a 1-norm, up to a factor nrow(X)

div <- function(o,e) {o/e}
X <- rSparseMatrix(10, 10, 30, NULL)
all.equal(
assocSparse(X, method = div),
cosSparse(X, norm = norm1) * nrow(X)
)

# ----- comparing methods -----
```

```
# Compare various methods on random data
# ignore values on diagonal, because different methods differ strongly here
# Note the different behaviour of pointwise mutual information (and division)

X <- rSparseMatrix(1e2, 1e2, 1e3, NULL)

p <- assocSparse(X, method = poi); diag(p) <- 0
r <- assocSparse(X, method = res); diag(r) <- 0
m <- assocSparse(X, method = pmi); diag(m) <- 0
w <- assocSparse(X, method = wpmi); diag(w) <- 0
d <- assocSparse(X, method = div); diag(d) <- 0

pairs(~w@x+p@x+r@x+d@x+m@x,
  labels=c("weighted pointwise\nmutual information","poisson","residuals","division",
           "pointwise\nmutual\ninformation"), cex = 0.7)
```

`bibles` *A selection of bible-texts*

Description

A selection of six bible texts as prepared by the `paralleltext.info` project.

Usage

```
data(bibles)
```

Format

A list of five elements

`verses` a character vector with all 43904 verse numbers as occurring throughout all translations as collected in the paralleltext.info project. This vector is used to align all texts to each other. The verse-numbers are treated as characters so ordering and matching works as expected.

`eng` The English 'Darby' Bible translation from 1890 by John Nelson Darby.

deu The Bible in German. Schlachter Version von 1951. Genfer Bibelgesellschaft 1951.

tgl The New Testament in Tagalog. Philippine Bible Society 1996.

aak The New Testament in the Ankave language of Papua New Guinea. Wycliffe Bible Translators, Inc. 1990.

Details

For details on the formatting, see paralleltext.info. Basically, all verse-numbering is harmonized, the text is unicode normalized, translations that capture multiple verses are included in the first of those verses, with the others left empty. Empty verses are thus a sign of combined translations. Verses that are not translated simply do not occur in the original files. Most importantly, the text are tokenized as to wordform, i.e. all punctuation and other non-word-based symbols are separated by spaces. In this way, space can be used for a quick wordform-based tokenization. The addition of spaces has been manually corrected to achieve a high precision of language-specific wordform tokenization.

The Bible texts are provided as named vectors of strings, each containing one verse. The names of the vector are codes for the verses. See Mayer & Cysouw (2014) for more information about the verse IDs and other formatting issues.

Source

All data is taken from paralleltext.info. For further information and links to the original versions, please check the detailed meta-information for all texts as provided on the webpage.

References

Mayer, Thomas and Michael Cysouw. 2014. Creating a massively parallel Bible corpus. *Proceedings of LREC 2014*.

Examples

```
# ----- load data -----

data(bibles)

# ----- separate into sparse matrices -----

# use splitText to turn a bible into a sparse matrix of wordforms x verses
E <- splitText(bibles$eng, simplify = TRUE, lowercase = FALSE)

# all wordforms from the first verse
# (internally using pure Unicode collation, i.e. ordering is determined by Unicode numbering)
which(E[,1] > 0)

# ----- co-occurrence across text -----

# how often do 'father' and 'mother' co-occur in one verse?
# (ignore warnings of chisq.test, because we are not interested in p-values here)

( cooc <- table(E["father",] > 0, E["mother",] > 0) )
```

```
suppressWarnings( chisq.test(cooc)$residuals )

# the function 'sim.words' does such computations efficiently
# for all 15000 x 15000 pairs of words at the same time

## Not run:
system.time( sim <- sim.words(bibles$eng, lowercase = FALSE) )
sim["father", "mother"]

## End(Not run)
```

---

corSparse *Pearson correlation between columns (sparse matrices)*

Description

This function computes the product-moment correlation coefficients between the columns of sparse matrices. Performance-wise, this improves over the approach taken in the cor function. However, because the resulting matrix is not-sparse, this function still cannot be used with very large matrices.

Usage

```
corSparse(X, Y = NULL, cov = FALSE)
```

Arguments

X	a sparse matrix in a format of the Matrix package, typically dgCMatrix . The correlations will be calculated between the columns of this matrix.
Y	a second matrix in a format of the Matrix package. When Y = NULL, then the correlations between the columns of X and itself will be taken. If Y is specified, the association between the columns of X and the columns of Y will be calculated.
cov	when TRUE the covariance matrix is returned, instead of the default correlation matrix.

Details

To compute the covariance matrix, the code uses the principle that

$$E[(X - \mu(X))'(Y - \mu(Y))] = E[X'Y] - \mu(X')\mu(Y)$$

With sample correction n/(n-1) this leads to the covariance between X and Y as

$$(X'Y - n * \mu(X')\mu(Y))/(n-1)$$

The computation of the standard deviation (to turn covariance into correlation) is trivial in the case Y = NULL, as they are found on the diagonal of the covariance matrix. In the case Y != NULL uses the principle that

$$E[X - \mu(X)]^2 = E[X^2] - \mu(X)^2$$

With sample correction n/(n-1) this leads to

$$sd^2 = (X^2 - n * \mu(X)^2)/(n-1)$$

Value

The result is a regular square (non-sparse!) Matrix with the Pearson product-moment correlation coefficients between the columns of `X`.

When `Y` is specified, the result is a rectangular (non-sparse!) Matrix of size `nrow(X)` by `nrow(Y)` with the correlation coefficients between the columns of `X` and `Y`.

When `cov = T`, the result is a covariance matrix (i.e. a non-normalized correlation).

Note

Because of the 'centering' of the Pearson correlation, the resulting Matrix is completely filled. This implies that this approach is normally not feasible with resulting matrices with more than 1e8 cells or so (except in dedicated computational environments with lots of RAM). However, in most sparse data situations, the cosine similarity `cosSparse` will almost be identical to the Pearson correlation, so consider using that one instead. For a comparison, see examples below.

For further usage, the many small coefficients are often unnecessary anyway, and can be removed for reasons of sparsity. Consider something like `M <- drop0(M, tol = value)` on the resulting `M` matrix (which removes all values between -value and +value). See examples below.

Author(s)

Michael Cysouw

Slightly extended and optimized, based on the code from a discussion at stackoverflow.

See Also

`cor` in the base packages, `cosSparse`, `assocSparse` for other sparse association measures.

Examples

```
## Not run:

# reasonably fast (though not instantly!) with
# sparse matrices up to a resulting matrix size of 1e8 cells.
# However, the calculations and the resulting matrix take up lots of memory

X <- rSparseMatrix(1e4, 1e4, 1e5)
system.time(M <- corSparse(X))
print(object.size(M), units = "auto") # more than 750 Mb

# Most values are low, so it often makes sense
# to remove low values to keep results sparse

M <- drop0(M, tol = 0.4)
print(object.size(M), units = "auto") # normally reduces size by half or more
length(M@x) / prod(dim(M)) # down to less than 0.05% non-zero entries

## End(Not run)

# comparison with other methods
```

```
# corSparse is much faster than cor from the stats package
# but cosSparse is even quicker than both!

X <- rSparseMatrix(1e3, 1e3, 1e4)
X2 <- as.matrix(X)

# if there is a warning, try again with different random X
system.time(McorRegular <- cor(X2))
system.time(McorSparse <- corSparse(X))
system.time(McosSparse <- cosSparse(X))

# cor and corSparse give identical results

all.equal(McorSparse, McorRegular)

# corSparse and cosSparse are not identical, but close

McosSparse <- as.matrix(McosSparse)
dimnames(McosSparse) <- NULL
all.equal(McorSparse, McosSparse)

# Actually, cosSparse and corSparse are *almost* identical!

cor(as.dist(McorSparse), as.dist(McosSparse))

# Visually it looks completely identical
# Note: this takes some time to plot

## Not run:
plot(as.dist(McorSparse), as.dist(McosSparse))

## End(Not run)

# So: consider using cosSparse instead of cor or corSparse.
# With sparse matrices, this gives mostly the same results,
# but much larger matrices are possible
# and the computations are quicker and more sparse
```

`cosNominal` *Associations-measures for sparsely encoded nominal variables*

Description

Nominal variables can be encoded as a combination of a sparse incidence and index matrix. Various functions to compute variations of `assocSparse` and `cosSparse` for such data are described here.

Usage

```
cosCol(X, colGroupX, Y = NULL, colGroupY = NULL, norm = norm2 )
```

```
assocCol(X, colGroupX, Y = NULL, colGroupY = NULL, method = res, sparse = TRUE)

cosRow(X, rowGroup, Y = NULL, norm = norm2 , weight = NULL)
assocRow(X, rowGroup, Y = NULL, method = res)
```

Arguments

X,Y	sparse matrices in a format of the Matrix package, typically dgCMatrix . When Y = NULL, then the similarity between the columns of X and itself will be taken. If Y is specified, the similarity between the columns of X and the columns of Y will be calculated.
colGroupX, colGroupY	sparse matrices (typically pattern matrices) with the same number of columns as X and Y, respectively, indicating which columns belong to the same group. Each row of these matrices represents a group.
rowGroup	sparse matrix (typically pattern matrices) with the same number of rows as X (and Y when not NULL), indicating which rows belong to the same group. Each column of these matrices represents a group.
norm	norm to be used. See cosSparse for details.
weight	weighting of rows. See cosSparse for details. Note that row-weighting only makes sense with cosRow.
method	method to be used. See assocSparse for details.
sparse	All methods try to be as sparse as possible. Specifically, when there are no observed co-occurrence, then nothing is computed. This might lead to slight deviations in the results for some methods. Set sparse=F to force computation for all cells. This leads to non-sparse results, so use with caution with large datasets.

Details

The approaches assoc and cos are described in detail in assocSparse and cosSparse, respectively. Those methods are extended here in case either the columns (.col) or the rows (.row) form groups. Specifically, this occurs with sparse encoding of nominal variables (see splitTable). In such encoding, the different values of a nominal variable are encoded in separate columns. However, these columns cannot be treated independently, but have to be treated as groups.

The .col methods should be used when similarities between the different values of nominal variables are to be computed. The .row methods should be used when similarities between the observations of nominal variables are to be computed.

Note that the calculations of the assoc functions really only makes sense for binary data (i.e. matrices with only ones and zeros). Currently, all input is coerced to such data by as(X, "nMatrix")*1, meaning that all values that are not one or zero are turned into one (including negative values!).

Value

When Y = NULL, then all methods return symmetric similarity matrices in the form dsCMatrix, only specifying the upper triangle. The only exception is when sparse=T is chose, then the result will be in the form dsyMatrix.

When a second matrix Y is specified, the result will be of the kind `dgCMatrix` or `dgeMatrix`, respectively.

Note

Note that these methods automatically take missing data into account. They also work with large amount of missing data, but of course the validity of any similarity with much missing data is problematic.

Author(s)

Michael Cysouw

See Also

`sim.att`, `sim.obs` for convenient shortcuts around these methods.

Examples

```
# convenience functions are easiest to use
# first a simple example using the farms-dataset from MASS
library(MASS)

# to investigate the relation between the individual values
# This is similar to Multiple Correspondence Analysis (see mca in MASS)
f <- splitTable(farms)
s <- assocCol(f$OV,f$AV)
rownames(s) <- f$values
plot(hclust(as.dist(-s)))
```

`cosSparse` *Cosine similarity between columns (sparse matrices)*

Description

`cosSparse` computes the cosine similarity between the columns of sparse matrices. Different normalizations and weightings can be specified. Performance-wise, this strongly improves over the approach taken in the `corSparse` function, though the results are almost identical for large sparse matrices. `cosMissing` adds the possibility to deal with large amounts of missing data.

Usage

```
cosSparse(X, Y = NULL, norm = norm2, weight = NULL)
cosMissing(X, availX, Y = NULL, availY = NULL, norm = norm2 , weight = NULL)
```

Arguments

X	a sparse matrix in a format of the Matrix package, typically dgCMatrix . The similarity will be calculated between the columns of this matrix.
Y	a second matrix in a format of the Matrix package. When Y = NULL, then the similarity between the columns of X and itself will be taken. If Y is specified, the similarity between the columns of X and the columns of Y will be calculated.
availX, availY	sparse Matrices (typically pattern matrices) of the same size of X and Y, respectively, indicating the available information for each matrix.
norm	The function to be used for the normalization of the columns. Currently norm2 (euclidean norm) and norm1 (manhattan norm) are available, but further methods can be easily specified by the user. See details for more information.
weight	The function to be used for the weighting of the rows. Currently idf (inverse document frequency) and isqrt (inverse square root) are available, but further methods can be easily specified by the user. See details for more information.

Details

This measure is called a 'cosine' similarity as it computes the cosine of the angle between high-dimensional vectors. It can also be considered a Pearson correlation without centering. Because centering removes sparsity, and because centering has almost no influence for highly sparse matrices, this cosine similarity performs much better that the Pearson correlation, both related for speed and memory consumption.

The variant cosMissing can be used when the available information itself is also sparse. In such a situation, a zero in the data matrix X, Y can mean either 'zero value' or 'missing data'. To deal with the missing data, matrices indicating the available data can be specified. Note that this really only makes sense when the available data is sparse itself. When, say, 90% of the data is available, the availX matrix becomes very large, and the results does not differ strongly from the regular cosSparse, i.e. ignoring the missing data.

Different normalizations of the columns and weightings of the rows can be specified.

The predefined normalizations are defined as a function of the matrix x and a 'summation function' s (to be specified as a sparse matrix or a vector). This slight complexity is needed to be able to deal with missing data. With complete data, then `s = rep(1,nrow(X))`, leads to `crossprod(X,s) == colSums(X)`.

- norm2: euclidean norm. The default setting, and the same normalization as used in the Pearson correlation. It is defined as
  `norm2 <- function(x,s) { drop(crossprod(x^2,s)) ^ 0.5 }`.
- norm1: Manhattan, or taxi-cab norm, defined as
  `norm1 <- function(x,s) { abs(drop(crossprod(x,s))) }`.
- normL: normalized Laplacian norm, used in spectral clustering of a graph, defined as
  `normL <- function(x,s) { abs(drop(crossprod(x,s))) ^ 0.5 }`.

The predefined weightings are defined as a function of the frequency of a row (s) and the number of columns (N):

- `idf`: inverse document frequency, used typically in distributional semantics to down-weight high frequent rows. It is defined as
  `idf <- function(s,N) { log(N/(1+s)) }`.
- `isqrt`: inverse square root, an alternative to idf, defined as
  `isqrt <- function(s,N) { s^-0.5 }`.
- `none`: no weighting. This is only included for use inside later high-level functions (e.g. `sim.words`). Normally, `weight = NULL` gives identical results, but is slightly quicker.
  `none <- function(s,N) { s }`

Further norms of weighting functions can be defined at will.

Value

The result is a sparse matrix with the non-zero association values. Values range between -1 and +1, with values close to zero indicating low association.

When `Y = NULL`, then the result is a symmetric matrix, so a matrix of type `dsCMatrix` with size `ncol(X)` by `ncol{X}` is returned. When `X` and `Y` are both specified, a matrix of type `dgCMatrix` with size `ncol(X)` by `ncol{Y}` is returned.

Note

For large sparse matrices, consider this as an alternative to `cor`. See `corSparse` for a comparison of performance and results.

Author(s)

Michael Cysouw

See Also

`corSparse`, `assocSparse` for other sparse association measures. See also `cosRow`, `cosCol` for variants of cosSparse dealing with nominal data.

Examples

```
# functional upto limits of RAM of the hardware used
# consider removing small values of result to improve sparsity

## Not run:

X <- rSparseMatrix(1e6, 1e6, 1e7)
print(object.size(X), units = "auto") # 118 Mb
system.time(M <- cosSparse(X)) # might take half a minute, depending on hardware
print(object.size(M), units = "auto") # 587 Mb
M <- drop0(M, tol = 0.1) # remove small values
print(object.size(M), units = "auto") # reduced to 141 Mb

## End(Not run)

# Compare various weightings.
```

```
# with random data from a normal distribution there is almost no difference
#
# data from a normal distribution
# X <- rSparseMatrix(1e2, 1e2, 1e3)

# with heavily skewed data there is a strong difference!
X <- rSparseMatrix(1e2, 1e2, 1e3,
rand.x = function(n){round(rpois(1e3, 10), 2)})

w0 <- cosSparse(X, norm = norm2, weight = NULL)@x
wi <- cosSparse(X, norm = norm2, weight = idf)@x
ws <- cosSparse(X, norm = norm2, weight = isqrt)@x

pairs(~ w0 + wi + ws,
  labels=c("no weighting","inverse\ndocument\nfrequency","inverse\nsquare root"))
```

`dimRed` *Dimensionality Reduction for sparse matrices, based on Cholesky decomposition*

Description

To inspect the structure of a large sparse matrix, it is often highly useful to reduce the matrix to a few major dimensions (cf. multidimensional scaling). This functions implements a rough approach to provide a few major dimensions. The function provides a simple wrapper around `Cholesky` and `sparsesvd`.

Usage

```
dimRed(sim, k = 2, method = "svd")
```

Arguments

`sim`	Sparse, symmetric, positive-definite matrix (typically a similarity matrix produces by `sim` or `assoc` functions)
`k`	Number of dimensions to be returned, defaults to two.
`method`	Method used for the decomposition. Currently implemted are `svd` and `cholesky`.

Details

Based on the Cholesky decomposition, the Matrix `sim` is decomposed into:

$$LDL'$$

The D Matrix is a diagonal matrix, the values of which are returned here as `$D`. Only the first few columns of the L Matrix are returned (possibly after permutation, see the details at `Cholesky`).

Based on the svd decomposition, the Matrix `sim` is decomposed into:

$$UDV$$

The U Matrix and the values from D are returned.

Value

A list of two elements is returned:

L	: a sparse matrix of type dgCMatrix with k columns
D	: the diagional values from the Cholesky decomposition, or the eigenvalues from the svd decomposition

Author(s)

Michael Cysouw <cysouw@mac.com>

See Also

See Also as Cholesky and sparsesvd

Examples

```
# some random points in two dimensions
coor <- cbind(sample(1:30), sample(1:30))

# using cmdscale() to reconstruct the coordinates from a distance matrix
d <- dist(coor)
mds <- cmdscale(d)

# using dimRed() on a similarity matrix.
# Note that normL works much better than other norms in this 2-dimensional case
s <- cosSparse(t(coor), norm = normL)
red <- as.matrix(dimRed(s)$L)

# show the different point clouds

par(mfrow = c(1,3))

  plot(coor, type = "n", axes = FALSE, xlab = "", ylab = "")
  text(coor, labels = 1:30)
  title("Original coordinates")

  plot(mds, type = "n", axes = FALSE, xlab = "", ylab = "")
  text(mds, labels = 1:30)
  title("MDS from euclidean distances")

  plot(red, type = "n", axes = FALSE, xlab = "", ylab = "")
  text(red, labels = 1:30)
  title("dimRed from cosSparse similarity")

par(mfrow = c(1,1))

# ======

# example, using the iris data
data(iris)
X <- t(as.matrix(iris[,1:4]))
```

```
cols <- rainbow(3)[iris$Species]

s <- cosSparse(X, norm = norm1)
d <- dist(t(X), method = "manhattan")

svd <- as.matrix(dimRed(s, method = "svd")$L)
chol <- as.matrix(dimRed(s, method = "cholesky")$L)
mds <- cmdscale(d)

par(mfrow = c(1,3))
  plot(mds, col = cols, main = "cmdscale\nfrom euclidean distances")
  plot(svd, col = cols, main = "dimRed with svd\nfrom cosSparse with norm1")
  plot(chol, col = cols, main = "dimRed with cholesky\nfrom cosSparse with norm1")
par(mfrow = c(1,1))
```

`distSparse` *Sparse distance matrix calculations*

Description

Sparse alternative to base `dist` function. WARNING: the result is not a distance metric, see details! Also: distances are calculated between columns (not between rows, as in the base `dist` function).

Usage

```
distSparse(M, method = "euclidean", diag = FALSE)
```

Arguments

`M`	a sparse matrix in a format of the `Matrix` package, typically `dMatrix`. Any other matrices will be converted to such a sparse Matrix. The correlations will be calculated between the columns of this matrix (different from the base `dist` function!)
`method`	method to calculate distances. Currently only `"euclidean"` is supported.
`diag`	should the diagonal be included in the results?

Details

A sparse distance matrix is a slightly awkward concept, because distances of zero are rare in most data. Further, it is mostly the small distances that are of interest, and not the large distanes (which are mostly also less trustwhorthy). Note that for random data, this assumption is not necessarily true.

To obtain sparse results, the current implementation takes a special approach. First, only those distances will be calculated for which there is at least some non-zero data for both columns. The assumption is taken that those distances will be uninteresting (and relatively large anyway).

Second, to differentiate the non-calculated distances from real zero distances, the distances are converted into similarities by substracting them from the maximum. In this way, all non-calculated distances are zero, and the real zeros have value `max(M)`.

Euclidean distances are calculated using the following trick:

$$colSums(M^2) + rowSums(M^2) - 2 * M'M$$

Value

A symmetric matrix of type `dsCMatrix`, consisting of similarity(!) values instead of distances (viz. `max(dist)-dist`).

Note

Please note:

- The values in the result are not distances, but similarities computed as `max(dist)-dist`.
- Non-calculated values are zero.

Author(s)

Michael Cysouw <cysouw@mac.com

See Also

See Also as `dist`.

Examples

```
# to be done
```

`huber` *Comparative vocabulary for indigenous languages of Colombia (Huber & Reed 1992)*

Description

Data from Huber & Reed (1992), containing a comparative vocabulary (a 'wordlist') for 69 indigenous languages from Colombia.

Usage

```
data(huber)
```

Format

A data frame with 27521 observations on the following 4 variables.

`CONCEPT` a factor with 366 levels, indicating the comparative concepts

`COUNTERPART` a character vector listing the actual wordforms as described in Huber & Reed 1992

`DOCULECT` a factor with 71 levels, indicating the languages from which the wordforms are taken ('documented lects', abbreviated as 'doculect'). These are 69 indigenous languages from Colombia, and English and Spanish.

`TOKENS` a tokenized version of the counterparts: spaces are added between graphemic units (i.e. groups of unicode characters that are functioning as a single unit in the orthography)

Details

The editors have attempted to use a harmonized orthography throughout all languages, approximately based on IPA, though there are still many language-specific idiosyncrasies included. However, the translations into English and Spanish are written in their regular orthography, and not in the IPA-dialect as used for the other languages. In general, the 'translations' into English and Spanish are simply lowercase versions of the concept-names, included here to more flexibly identify the meaning of words in the Colombian languages. In many cases these translations are somewhat clunky (e.g. 'spring of water'), and are missing the proper orthography details (e.g. 'Adams apple').

The book was digitized in the QuantHistLing project and provided here as an example of dealing efficiently with reasonably large data. Care has been taken to faithfully represent the original transcription from the printed version.

Source

Huber, Randall Q. & Robert B. Reed. 1992. *Vocabulario Comparativo: Palabras Selectas de Lenguas Indigenas de Colombia.* Bogota: Instituto Linguistico de Verano. available online at http://www.sil.org/americas/COLOMBIA/show_work.asp?id=928474518886. Copyright 2014 SIL International.

Examples

```
data(huber)
head(huber)
```

`jMatrix` *Harmonize ('join') sparse matrices*

Description

A utility function to make sparse matrices conformable semantically. Not only are the dimensions made conformable in size, but also the content of the dimensions. Formulated differently, this function harmonizes two matrices on a dimensions that have the same entities, but in a different order (and possibly with different subsets). Given two matrices with such (partly overlapping) dimensions, two new matrices are generated to reorder the original matrices via a matrix product to make them conformable. In an abstract sense, this is similar to an SQL 'inner join' operation.

Usage

```
jMatrix(rownamesX, rownamesY, collation.locale = "C")

jcrossprod(X, Y, rownamesX = rownames(X), rownamesY = rownames(Y))
tjcrossprod(X, Y, colnamesX = colnames(X), colnamesY = colnames(Y))
```

Arguments

`rownamesX, rownamesY`	rownames to be joined from two matrices.
`X, Y`	sparse matrices to be made (semantically) conformable.
`colnamesX, colnamesY`	colnames to be joined from two matrices.
`collation.locale`	locale to be used for ordering of the joined dimension. Defaults to pure numerical unicode ordering "C". See `ttMatrix` for details.

Details

Given a sparse matrix X with rownames rX and a sparse matrix Y with rownames rY, the function `jMatrix` produces joined rownames rXY with all unique entries in `c(rX, rY)`, reordered according to the specified locale, if necessary.

Further, two sparse matrices M1 and M2 are returned to link X and Y to the new joined dimension rXY. Specifically, X2 = M1 %*% X and Y2 = M2 %*% Y will have conformable rXY rows, so crossprod(X2, Y2) can be computed. Note that the result will be empty when there is no overlap between the rownames of X and Y.

The function `jcrossprod` is a shortcut to compute the above crossproduct immediately, using `jMatrix` internally to harmonize the rows. Similarly, `tjcrossprod` computes the tcrossprod, harmonizing the *columns* of two matrices using `jMatrix`.

Value

`jMatrix` returns a list of three elements (for naming, see Details above):

`M1`	sparse pattern matrix of type `ngCMatrix` with dimensions `c(length(rXY),length(rX))`
`M2`	sparse pattern matrix of type `ngCMatrix` with dimensions `c(length(rXY),length(rY))`
`rownames`	unique joined row names rXY

`jcrossprod` and `tjcrossprod` return a sparse Matrix of type `ngCMatrix` when both X and Y are pattern matrices. Otherwise they return a sparse Matrix of type `dgCMatrix`.

Note

Actually, it is unimportant whether the inputs to `jMatrix` are row or column names. However, care has to be taken to use the resulting matrices in the right transposition. To make this function easier to explain, I consistently talk only about row names above.

Author(s)

Michael Cysouw

Examples

```
# example about INNER JOIN from wikipedia
# http://en.wikipedia.org/wiki/Sql_join#Inner_join
# this might look complex, but it is maximally efficient on large sparse matrices

# Employee table as sparse Matrix
Employee.LastName <- c("Rafferty","Jones","Heisenberg","Robinson","Smith","John")
Employee.DepartmentID <- c(31,33,33,34,34,NA)
E.LN <- ttMatrix(Employee.LastName, simplify = TRUE)
E.DID <- ttMatrix(Employee.DepartmentID, simplify = TRUE)

( Employees <- tcrossprod(E.LN, E.DID) )

# Department table as sparse Matrix
Department.DepartmentID <- c(31,33,34,35)
Department.DepartmentName <- c("Sales","Engineering","Clerical","Marketing")
D.DID <- ttMatrix(Department.DepartmentID, simplify = TRUE)
D.DN <- ttMatrix(Department.DepartmentName, simplify = TRUE)

( Departments <- tcrossprod(D.DN, D.DID) )

# INNER JOIN on DepartmentID (i.e. on the columns of these two matrices)
# result is a sparse matrix linking Employee.LastName to Department.DepartmentName,
# internally having used the DepartmentID for the linking

( JOIN <- tjcrossprod(Employees, Departments) )

# Note that in this example it is much easier to directly use jMatrix on the DepartmentIDs
# instead of first making sparse matrices from the data
# and then using tjcrossprod on the matrices to get the INNER JOIN
# (only the ordering is different in this direct approach)

J <- jMatrix(Employee.DepartmentID, Department.DepartmentID)
JOIN <- crossprod(J$M1, J$M2)
rownames(JOIN) <- Employee.LastName
colnames(JOIN) <- Department.DepartmentName
JOIN
```

---

pwMatrix *Construct 'part-whole' (pw) Matrices from tokenized strings*

Description

A part-whole Matrix is a sparse matrix representation of a vector of strings ('wholes') split into smaller parts by a specified separator. It basically summarizes which strings consist of which parts. By itself, this is not a very interesting transformation, but it allows for quite fancy computations by simple matrix manipulations.

Usage

```
pwMatrix(strings, sep = "", gap.length = 0, gap.symbol = "\u2043", simplify = FALSE)
```

Arguments

strings	a vector (or list) of strings to be separated into parts
sep	The separator to be used. Defaults to space sep = " ". If separation in individual characters is needed, use sep = "". There is no fancy parsing of strings implemented (e.g. to catch complex unicode combined characters), that has to be done externally. The preferred route is to prepare the separation of the strings by using spaces, and then call this function.
gap.length	This adds the specified number of gap symbols between each pair of strings. This is only important for generating higher ngram-statistics later on, when no ordering of the strings is implied. For example, when the strings are alphabetically ordered words, any bigram-statistics should not count the bigrams consisting of the last character of the a word with the first character of the next word.
gap.symbol	The gap symbol to insert (see gap.length above). It defaults to U+2043 HYPHEN BULLET on the assumption that this character will not often be included in data.
simplify	by default, the row and column names are not included into the matrix to keep the matrix as lean as possible. The row names ('parts') are returned separately. Using simplify = T the row and column names will be added into the matrix. Note that the column names are simply the vector that went into the function.

Details

Internally, this is basically using strsplit and some cosmetic changes, returning a sparse matrix.

Value

By default (when simplify = F) the output is a list with two elements, containing:

M	a sparse pattern Matrix of type ngCMatrix with all input strings as columns, and all separated elements as rows.
rownames	all different characters from the strings in order (i.e. all individual tokens of the original strings).

When simplify = T, then only the matrix M with row and column names is returned.

Author(s)

Michael Cysouw

See Also

Used in splitStrings and splitWordlist

Examples

```
# By itself, this functions does nothing really interesting
example <- c("this","is","an","example")
pw <- pwMatrix(example)
pw

# However, making a type-token Matrix (with ttMatrix) of the rownames
# and then taking a matrix product, results in frequencies of each element in the strings
tt <- ttMatrix(pw$rownames)
distr <- (tt$M*1) %*% (pw$M*1)
rownames(distr) <- tt$rownames
colnames(distr) <- example
distr

# Use banded sparse matrix with superdiagonal ('shift matrix') to get co-occurrence counts
# of adjacent characters. Rows list first character, columns adjacent character.
# Non-zero entries list number of co-occurrences
S <- bandSparse( n = ncol(tt$M), k = 1) * 1
TT <- tt$M * 1
( C <- TT %*% S %*% t(TT) )

# show the non-zero entries as triplets:
s <- summary(C)
first <- tt$rownames[s[,1]]
second <- tt$rownames[s[,2]]
freq <- s[,3]
data.frame(first,second,freq)
```

---

rKhatriRao *'reduced' Khatri-Rao product (sparse matrices)*

Description

This function performs a Khatri-Rao product ('column-wise Kronecker product', see KhatriRao for more info) on two sparse matrices. However, the result of such a product on sparse matrices normally results in very many empty rows. This function removes those empty rows, and, most importantly, it produces row names only for the remaining rows. For large sparse matrices this is *much* more efficient than first producing all rownames, and then removing the one with the empty rows.

Usage

```
rKhatriRao(X, Y,
rownamesX = rownames(X), rownamesY = rownames(Y),
simplify = FALSE, binder = ":", FUN = "*")
```

Arguments

X,Y	matrices of with the same number of columns.
rownamesX, rownamesY	row names of matrices X and Y. These can be specified separately, but they default to the row names of the matrices.
simplify	by default, the names of rows and columns are not included into the matrix to keep the matrix as lean as possible: the row names are returned separately. Using include.dimnames=T adds the row names into the matrix. The column names are directly taken from X.
binder	symbol to include between the row names of X and Y for the resulting matrix
FUN	function to be used in the KhatriRao product, passed internally to the workhorse KhatriRao

Details

Up to 1e6 row names to be produced goes reasonably quick with the basic KhatriRao function. However, larger amounts of pasting of row names becomes very slow, and the row names take an enormous amount of RAM. This function solves that problem by only producing row names for the non-empty rows.

Value

By default, the result is a list of two items:

M	resulting sparse product matrix with empty rows removed
rownames	a vector with the resulting row names for the non-empty rows

When simplify=T, then the matrix is return with the row names included.

Note

This function allows for the row names of the input matrices to be added separately, and the resulting row names are returned separately by default. This might seem a bit unusual, given the nice way how R integrates row names into matrices. However, it turns out often to be easier to store row- and column names separately to efficiently work with large sparse matrices.

Author(s)

Michael Cysouw

See Also

KhatriRao

Examples

```
# two sparse matrices with row names

X <- rSparseMatrix(1e4, 1e3, 1e4)
Y <- rSparseMatrix(1e4, 1e3, 1e4)

rownames(X) <- 1:nrow(X)
rownames(Y) <- 1:nrow(Y)

# the basic KhatriRao product is very fast
# but almost all rows are empty

system.time(M <- KhatriRao(X, Y))
## Not run:
sum(rowSums(M)==0)/nrow(M) # 99.9% empty rows

## End(Not run)

# To produce all row names takes a long time
# with 1e8 row names it took half an hour on my laptop
# so: don't try the following, except on a very large machine!
## Not run:
system.time(M <- KhatriRao(X, Y, make.dimnames = TRUE))

## End(Not run)

# Using the current special version works just fine and is reasonably quick
system.time(M <- rKhatriRao(X, Y))
```

`rowMax` *Row and column extremes (sparse matrices)*

Description

Compute maxima and minima for all rows or columns of sparse matrices. Optionally also return which elements are the maxima/minima per row/column.

Usage

```
rowMax(X, which = FALSE, ignore.zero = TRUE)
colMax(X, which = FALSE, ignore.zero = TRUE)

rowMin(X, which = FALSE, ignore.zero = TRUE)
colMin(X, which = FALSE, ignore.zero = TRUE)
```

Arguments

X	a sparse matrix in a format of the Matrix package, typically dgCMatrix . The maxima or minima will be calculated for each row or column of this matrix.
which	optionally return a sparse matrix of the same dimensions as X marking the positions of the columns- or row-wise maxima or minima.
ignore.zero	By default, only the non-zero elements are included in the computations. However, when ignore.zero = F then zeros are also considered. This basically means that for all maxima below zero, the maximum will be set to zero. Likewise, for all minima above zero, the minimum will be set to zero.

Details

The basic workhorse of these functions is the function rollup from the package slam.

Value

By default, these functions returns a sparseVector with the non-zero maxima or minima. Use additionally as.vector to turn this into a regular vector.

When which = T, the result is a list of two items:

max/min	the same sparse vector as described above.
which	a sparse pattern matrix of the kind ngCMatrix indicating the position of the extrema. Note that an extreme might occur more than once per row/column. In that case multiple entries in the row/column are indicated.

Author(s)

Michael Cysouw

Examples

```
# rowMax(X, ignore.zero = FALSE) is the same as apply(X, 1, max)
# however, with large sparse matrices, the 'apply' approach will start eating away at memory
# and things become slower.
X <- rSparseMatrix(1e3, 1e3, 1e2)
system.time(m1 <- rowMax(X, ignore.zero = FALSE))
system.time(m2 <- apply(X, 1, max)) # slower
all.equal(as.vector(m1), m2) # but same result

# to see the effect even stronger, try something larger
# depending on the amount of available memory, the 'apply' approach will give an error
# "problem too large"
## Not run:
X <- rSparseMatrix(1e6, 1e6, 1e6)
system.time(m1 <- rowMax(X, ignore.zero = FALSE))
system.time(m2 <- apply(X, 1, max))

## End(Not run)
```

```
# speed depends most strongly on the number of entries in the matrix
# also some performance loss with size of matrix
# up to 1e5 entries is still reasonably fast

X <- rSparseMatrix(1e7, 1e7, 1e5)
system.time(m <- rowMax(X))

## Not run:
X <- rSparseMatrix(1e7, 1e7, 1e7)
system.time(M <- rowMax(X)) # about ten times as slow

## End(Not run)

# apply is not feasably on such large matrices
# Error: problem too large...
## Not run:
m <- apply(X, 1, max)

## End(Not run)
```

`rSparseMatrix` *Construct a random sparse matrix*

Description

This convenience function constructs a random sparse matrix of specified size, with specified sparsity. This is mainly useful for testing speed and memory load of sparse matrix manipulations

Usage

```
rSparseMatrix(nrow, ncol, nnz,
rand.x = function(n) round(rnorm(nnz), 2), ...)
```

Arguments

`nrow`	number of rows of the resulting matrix.
`ncol`	number of columns of the resulting matrix.
`nnz`	number of entries of the resulting matrix.
`rand.x`	randomization used for the construction of the entries. if `NULL` then a pattern matrix is constructed (random entries without values).
`...`	Other arguments passed to `sparseMatrix` internally.

Details

The sparsity of the resulting matrix (i.e. the fraction of non-zero entries to all entries) is $\frac{nnz}{nrow*ncol}$.

Value

Returns a sparse matrix of the type `dgCMatrix`. Defaults to random numeric entries with two decimal digits, generated randomly from a normal distribution with `mean = 0` and `sd = 1`.

When `rand.x = NULL` then the result is a patter**n** matrix of type `ngCMatrix`.

Author(s)

Martin Maechler with slight tweaks by Michael Cysouw

See Also

For random permutation matrices, see `pMatrix-class`. Specifically note the construction option `(p10 <- as(sample(10),"pMatrix"))`.

Examples

```
# example with reasonably large (100.000 by 100.000) but sparse matrix
# (only one in 10.000 entries is non-zero). On average 10 entries per column.
X <- rSparseMatrix(1e5, 1e5, 1e6)
print(object.size(X), units = "auto")

# speed of cosine similarity
system.time(M <- cosSparse(X))

# reduce memory footprint by removing low values
print(object.size(M), units = "auto")
M <- drop0(M, tol = 0.1)
print(object.size(M), units = "auto")
```

`sim.nominal` *Similarity-measures for nominal variables*

Description

Nominal variables can be encoded as a combination of a sparse incidence and index matrix, as discussed at `splitTable`. The present two functions are easy-to-use shortcuts to use those sparse matrices to computes pairwise similarities, either between observations (`sim.obs`) or attributes (`sim.att`).

Usage

```
sim.att(D, method = "chuprov", sparse = TRUE, ...)
sim.obs(D, method = "hamming", sparse = TRUE, ...)
```

Arguments

D	Dataframe with nominal attributes ('variables') as columns and observations as rows.
method	method to be used for similarity computation. See Details below.
sparse	All methods try to be as sparse as possible. Specifically, when there are no observed co-occurrence, then nothing is computed. This might lead to slight deviations in the results for some methods. Set sparse=F to force computation for all cells. This leads to non-sparse results, so use with caution with large datasets.
...	Arguments passed internally to splitTable, especially useful for multi-valued cells, using the option split. Note that method = hamming will give unexpected results for the comparison of cells that both are multi-valued. Consider using method = weighted instead.

Details

The function sim.att and sim.obs are convenience wrappers around the basic cosRow, cosCol and assocRow, assocCol functions. The sim functions take a dataframe as input, internally calling splitTable to turn the dataframe into sparse matrices, and then applying sparse matrix algebra to efficiently compute similarities. Currently only a few exemplary methods are encoded.

sim.att computes similarities between the different nominal variables. The method chuprov computes Chuprov's T (very similar to Cramer's V, but easier to compute efficiently). The method g computes the G-test from Sokal and Rohlf (1982), also known as Dunning's G from Dunning (1993). This G is closely related to Mutual Information (G = 2*N*MI, with N being the sample size). The method mutual returns the mutual information, and the method variation returns the so-called 'variation of information' (join information - mutual information). Note that the this last one is a metric, not a similarity. All these methods can be abbreviated, e.g use "c", "g", "m", and "v".

sim.obs computes similarities between the different observation for the nominal variables. The method hamming computes the relative Hamming similarity, i.e. the number of similarities devided by the number of comparisons made (Goebl 1984 calls this the 'Relativer Identitaetswert'). The method weighted uses an inverse square root weighting on all similarities, i.e. rare similarities count more. This is very similar to Goebl's 'Gewichteter Identitaetswert', though note that his definition is slightly different from the one used here. Further, all methods as defined for assocSparse can be used here, i.e. res, pmi, wpmi, poi, and new methods can be defined according to the explanations as assocSparse.

Value

All methods return symmetric similarity matrices in the form dsCMatrix, only specifying the upper triangle. The only exception is when sparse=T is chose, then the result will be in the form dsyMatrix.

Note

Note that these methods automatically take missing data into account. They also work with large amount of missing data, but of course the validity of any similarity with much missing data is problematic.

The `sim.att` and `sim.obs` methods by default use sparse computations, which leads (among other effects) to errors on the diagonal. The main diagonal should be one everywhere by definition, but this will only be the case with the option `sparse = F`. The deviations with `sparse = T` should be minimal in the non-diagonal entries, but computations should be faster, and the results often take up less space.

Author(s)

Michael Cysouw

References

Goebl, Hans. 1984. *Dialektometrische Studien: anhand italoromanischer, raetoromanischer und galloromanischer Sprachmaterialien aus AIS und AFL.* (Beihefte zur Zeitschrift fuer Romanische Philologie). Tuebingen: Niemeyer.

Dunning, Ted. 1993. Accurate methods for the statistics of surprise and coincidence. *Computational linguistics* 19(1). 61-74.

Examples

```
# first a simple example using the farms-dataset from MASS
library(MASS)

# similarities between farms
s <- sim.obs(farms)
plot(hclust(as.dist(1-s), method = "ward.D"))

# similarities between attributes (`variables`)
s <- sim.att(farms)
plot(hclust(as.dist(1-s), method = "ward.D"))

# use the split option for multi-valued cells
farms2 <- as.matrix(farms)
farms2[1,1] <- "M1,M5"

s <- sim.obs(farms2, split = ",")
plot(hclust(as.dist(1-s), method = "ward.D"))

## Not run:
# a larger example with lots of missing data: the WALS-data as included here
# computations go reasonably quick
# (on 2566 observations and 131 attributes with 630 different values in total)
data(wals)
system.time(s <- sim.att(wals$data))
rownames(s) <- colnames(wals$data)
plot(hclust(as.dist(1-s), method = "ward.D"), cex = 0.5)

# Note that using sparse=T speeds up computations because it
# ignores zero co-occurrences
# This leads to small errors in the computation of Chuprov's T
system.time( # faster
```

```
chup.sparse <- sim.att(wals$data,  method = "chuprov", sparse = TRUE)
)
system.time( # slower
chup.full <- sim.att(wals$data, method = "chuprov", sparse = FALSE)
)

# The sparse approach is almost identical to the full approach.
# sparse sligtly underestimates the real values for Chuprov's T
plot(as.dist(chup.sparse), as.dist(chup.full))

# some more similarities on the attributes
g <- sim.att(wals$data, method = "g") # Dunning's G
m <- sim.att(wals$data, method = "mutual") # Mutual Information
v <- sim.att(wals$data, method = "variation") # Variation of Information

# Note the strong differences between these approaches
pairs(~ as.dist(chup.sparse) + as.dist(m) + as.dist(g) + as.dist(v),
labels=c("Chuprov's T","Mutual Information","G-statistic","Variation of Information"))

# Relative Hamming similarity on all observations (languages) in WALS
# time is not a problem, but the data is so sparse
# that for many language-pairs there is no shared data
system.time( s <- sim.obs(wals$data))

# select only the 168 language with more than 80 datapoints
sel <- wals$data[apply(wals$data,1,function(x){sum(!is.na(x))})>80,]

# compare different similarities
w <- sim.obs(sel, "weighted")
h <- sim.obs(sel, "hamming")
r <- sim.obs(sel, "res")
p <- sim.obs(sel, "poi")
m <- sim.obs(sel, "wpmi")
i <- sim.obs(sel, "pmi")

pairs(~ as.dist(w) + as.dist(h) + as.dist(r) + as.dist(p) + as.dist(m) + as.dist(i),
labels = c("weighted","hamming","residuals","poisson","weighted PMI","PMI"))

## End(Not run)
```

`sim.strings` *String similarity by cosine similarity between bigram vectors*

Description

Efficient computation of pairwise string similarities using a cosine similarity on bigram vectors.

Usage

```
sim.strings(strings1, strings2 = NULL, sep = "", boundary = TRUE, ...)
```

Arguments

strings1, strings2	Vector with strings to be compared, will be treated as.character. When only strings1 is provided, all pairwise similarities between its elements are computed. When two different input vectors are provided, the pairwise similarities between all elements from the first and the second vector are computed.
sep	Separator used to split the strings into parts. This will be passed to strsplit internally, so there is no fine-grained control possible on the splitting. If it is important to get the splitting exactly right, consider pre-processing the splitting by inserting a special symbol on the split-positions, and then choosing here to split by this special symbol.
boundary	In the default setting boundary = T, a special symbol is added to the front and to the end of each string, adding special bigrams for the initial and the final character. With words from real languages (which are mostly not very long) this has a strong impact.
...	Further arguments passed to splitStrings.

Details

The strings are converted into sparse matrices by splitStrings, and then assocSparse computes a cosine similarity on the bigram vectors. Only the option of bigrams is currently used, because for long lists of real words from a real language this seems to be an optimal tradeoff between speed and useful similarity.

Value

When either length(strings1) == 1 or length(strings2) == 1, the result will be a normal vector with similarities between 0 and 1.

When both the input vectors are longer than 1, then the result will be a sparse matrix with similarities. When only strings1 is provided, then the result is of type dsCMatrix. When two input vectors are provided, the result is of type dgCMatrix.

Note

The overhead of converting the strings into sparse matrices makes this function not optimal for small datasets. For large datasets the time of the conversion is negligible compared to the actual similarity computation, and then this approach becomes very worthwhile, because fast, and based on sparse matrix computation, that can be sped up by multicore processing in the future.

The result of sim.strings(a) and sim.strings(a,a) is identical, but the first version is more efficient, both as to processing time, as well as to the size of the resulting objects.

Note

There is a bash-executable simstrings distributed with this package (based on the docopt package) that let you use this function directly in a bash-terminal. The easiest way to use this executable is to softlink the executable to some directory in your bash PATH, for example /usr/local/bin or simply ~/bin. To softlink the function sim.strings to this directory, use something like the following in your bash terminal:

```
ln -is `Rscript -e 'cat(system.file("exec/simstrings", package="qlcMatrix"))'` ~/bin
```

From within R your can also use the following (again, optionally changing the linked-to directory from ~/bin to anything more suitable on your system):

```
file.symlink(system.file("exec/simstrings", package="qlcMatrix"), "~/bin")
```

Author(s)

Michael Cysouw

See Also

splitStrings, cosSparse on which this function is based. Compare with adist from the utils package. On large datasets, sim.strings seems to be about a factor 30 quicker. The package stringdist offers many more string comparison methods.

Examples

```
# ----- simple example -----

example <- c("still","till","stable","stale","tale","tall","ill","all")
( sim <- round( sim.strings(example), digits = 3) )

# show similarity in non-metric MDS
library(MASS)
mds <- isoMDS( as.dist(1-sim) )$points
plot(mds, type = "n", ann = FALSE, axes = FALSE)
text(mds, labels = example, cex = .7)

## Not run:

# ----- large example -----

# This similarity is meant to be used for large lists of wordforms.
# for example, all 15526 wordforms from the English Dalby Bible
# takes just a few seconds for the more than 1e8 pairwise comparisons
data(bibles)
words <- splitText(bibles$eng)$wordforms
system.time( sim <- sim.strings(words) )

# see most similar words
rownames(sim) <- colnames(sim) <- words
sort(sim["walk",], decreasing = TRUE)[1:10]

# just compare all words to "walk". This is the same as above, but less comparisons
# note that the overhead for the sparse conversion and matching of matrices is large
# this one is faster than doing all comparisons, but only be a factor 10
system.time( sim <- sim.strings(words, "walk"))
names(sim) <- words
sort(sim, decreasing = TRUE)[1:10]

# ----- comparison with Levinshtein -----
```

```
# don't try this with 'adist' from the utils package, it will take long!
# for a comparison, only take 2000 randomly selected strings: about a factor 30 slower
w <- sample(words, 2000)
system.time( sim1 <- sim.strings(w) )
system.time( sim2 <- adist(w) )

# compare the current approach with relative levenshtein similarity
# = number of matches / ( number of edits + number of matches)
# for reasons of speed, just take 1000 random words from the english bible
w <- sample(words, 1000)
sim1 <- sim.strings(w)
tmp <- adist(w, counts = TRUE)
sim2 <- 1- ( tmp / nchar(attr(tmp, "trafos")) )

# plotting relation between the two 'heatmap-style'
# not identical, but usefully similar
image( log(table(
round(as.dist(sim1) / 3, digits = 2) * 3,
round(as.dist(sim2) / 3, digits = 2) * 3 )),
xlab = "bigram similarity", ylab = "relative Levenshtein similarity")




## End(Not run)
```

`sim.wordlist` *Similarity matrices from wordlists*

Description

A few different approaches are implemented here to compute similarities from wordlists. `sim.lang` computes similarities between languages, assuming a harmonized orthography (i.e. symbols can be equated across languages). `sim.con` computes similarities between concepts, using only language-internal similarities. `sim.graph` computes similarities between graphemes (i.e. language-specific symbols) between languages, as a crude approximation of regular sound correspondences.

WARNING: All these methods are really very crude! If they seem to give expected results, then this should be a lesson to rethink more complex methods proposed in the literature. However, in most cases the methods implemented here should be taken as a proof-of-concept, showing that such high-level similarities can be computed efficiently for large datasets. For actual research, I strongly urge anybody to adapt the current methods, and fine-tune them as needed.

Usage

```
sim.lang(wordlist,
```

```
doculects = "DOCULECT", concepts = "CONCEPT", counterparts = "COUNTERPART",
method = "parallel", assoc.method =  res, weight = NULL, sep = "")

sim.con(wordlist,
doculects = "DOCULECT", concepts = "CONCEPT", counterparts = "COUNTERPART",
method = "bigrams", assoc.method = res, weight = NULL, sep = "")

sim.graph(wordlist,
doculects = "DOCULECT", concepts = "CONCEPT", counterparts = "TOKENS",
method = "cooccurrence", assoc.method = poi, weight = NULL, sep = " ")
```

Arguments

wordlist	Dataframe or matrix containing the wordlist data. Should have at least columns corresponding to languages (DOCULECT), meanings (CONCEPT) and translations (COUNTERPART).
doculects, concepts, counterparts	The name (or number) of the column of wordlist in which the respective information is to be found. The defaults are set to coincide with the naming of the example dataset included in this package. See huber.
method	Specific approach for the computation of the similarities. See Details below.
assoc.method, weight	Measures to be used internally (passed on to assocSparse or cosSparse). See Details below.
sep	Separator to be used to split strings. See link{splitStrings} for details.

Details

The following methods are currently implemented (all methods can be abbreviated):

For sim.lang:

- global: Global bigram similarity, i.e. ignoring the separation into concepts, and simply taking the bigram vector of all words per language. Probably best combined with weight = idf.
- parallel: By default, computes a parallel bigram similarity, i.e. splitting the bigram vectors per language and per concepts, and then simply making one long vector per language from all individual concept-bigram vectors. This approach seems to be very similar (if not slightly better) than the widespread 'average Levenshtein' distance.

For sim.con:

- colexification: Simply count the number of languages in which two concepts have at least one complete identical translations. No normalization is attempted, and assoc.method and weight are ignored (internally this just uses tcrossprod on the CW (concepts x words) sparse matrix). Because no splitting of strings is necessary, this method is very quick.
- global: Global bigram similarity, i.e. ignoring the separation into languages, and simply taking the bigram vector of all words per concept. Probably best combined with weight = idf.

- `bigrams`: By default, compute the similarity between concepts by comparing bigraphs, i.e. language-specific bigrams. In that way, cross-linguistically recurrent partial similarities are uncovered. It is very interesting to compare this measure with `colexification` above.

For `sim.graph`:

- `cooccurrence`: Currently the only method implemented. Computes the co-occurrence statistics for all pair of graphemes (e.g. between symbol x from language L1 and symbol y from language L2). See Prokic & Cysouw (2013) for an example using this approach.

All these methods (except for `sim.con(method = "colexification")`) use either `assocSparse` or `cosSparse` for the computation of the similarities. For the different measures available, see the documentation there. Currently implemented are `res`, `poi`, `pmi`, `wpmi` for `assocSparse` and `idf`, `isqrt`, `none` for `cosWeight`. It is actually very easy to define your own measure.

When `weight = NULL`, then `assocSparse` is used with the internal method as specified in `assoc.method`. When `weight` is specified, then `cosSparse` is used with an Euclidean norm and the weighting as specified in `weight`. When `weight` is specified, and specification of `assoc.method` is ignored.

Value

A sparse similarity matrix of class `dsCMatrix`. The magnitude of the actual values in the matrices depend strongly on the methods chosen.

With `sim.graph` a list of two matrices is returned.

`GG`	The grapheme by grapheme similarity matrix of class `dsCMatrix`
`GD`	A pattern matrix of class indicating which grapheme belongs to which language.

Author(s)

Michael Cysouw

References

Prokic, Jelena and Michael Cysouw. 2013. Combining regular sound correspondences and geographic spread. *Language Dynamics and Change* 3(2). 147–168.

See Also

Based on `splitWordlist` for the underlying conversion of the wordlist into sparse matrices. The actual similarities are mostly computed using `assocSparse` or `cosSparse`.

Examples

```
# ----- load data -----

# an example wordlist, see help(huber) for details
data(huber)

# ----- similarity between languages -----

# most time is spend splitting the strings
```

```
# the rest does not really influence the time needed
system.time( sim <- sim.lang(huber, method = "p") )

# a simple distance-based UPGMA tree
plot(hclust(as.dist(-sim), method = "average"), cex = .7)

## Not run:

# ----- similarity between concepts -----

# similarity based on bigrams
system.time( simB <- sim.con(huber, method = "b") )
# similarity based on colexification. much easier to calculate
system.time( simC <- sim.con(huber, method = "c") )

# As an example, look at all adjectival concepts
adj <- c(1,5,13,14,28,35,40,48,67,89,105,106,120,131,137,146,148,
171,179,183,188,193,195,206,222,234,259,262,275,279,292,
294,300,309,341,353,355,359)

# show them as trees
par(mfrow = c(1,2))
plot(hclust(as.dist(-simB[adj,adj]), method = "ward"),
cex = .5, main = "bigrams")
plot(hclust(as.dist(-simC[adj,adj]), method = "ward"),
cex = .5, main = "colexification")
par(mfrow = c(1,1))

# ----- similarity between graphemes -----

# this is a very crude approach towards regular sound correspondences
# when the languages are not too distantly related, it works rather nicely
# can be used as a quick first guess of correspondences for input in more advanced methods

# all 2080 graphemes in the data by all 2080 graphemes, from all languages
system.time( X <- sim.graph(huber) )

# throw away the low values
# select just one pair of languages for a quick visualisation
X$GG <- drop0(X$GG, tol = 1)
colnames(X$GG) <- rownames(X$GG)
correspondences <- X$GG[X$GD[,"bora"],X$GD[,"muinane"]]
heatmap(as.matrix(correspondences))

## End(Not run)
```

---

sim.words — *Similarity-measures for words between two languages, based on co-occurrences in parallel text*

Description

Based on co-occurrences in a parallel text, this convenience function (a wrapper around various other functions from this package) efficiently computes something close to translational equivalence.

Usage

```
sim.words(text1, text2 = NULL, method = res, weight = NULL,
lowercase = TRUE, best = FALSE, tol = 0)
```

Arguments

text1, text2	Vectors of strings representing sentences. The names of the vectors should contain IDs that identify the parallelism between the two texts. If there are no specific names, the function assumes that the two vectors are perfectly parallel. Within the strings, wordforms are simply separated based on spaces (i.e. everything between two spaces is a wordform). For more details about the format-assumptions, see splitText, which is used internally here.
method	Method to be used as a co-occurrence statistic. See assocSparse for a detailed presentation of the available methods. It is possible to define your own statistic, when it can be formulated as a function of observed and expected frequencies.
weight	When weight is specified, the function cosSparse is used for the co-occurrence statistics (with a Euclidean normalization, i.e. norm2). The specified weight function will be used, currently idf, sqrt, and none are available. For more details, and for instructions how to formulate your own weight function, see the discussion at cosSparse. When weight is specified, any specification of method is ignored.
lowercase	Should all words be turned into lowercase? See splitText for discussion how this is implemented.
best	When best = T, an additional sparse matrix is returned with a (simplistic) attempt to find the best translational equivalents between the texts.
tol	Tolerance: remove all values between -tol and +tol in the result. Low values can mostly be ignored for co-occurrence statistics without any loss of information. However, what is considered 'low' depends on the methods used to calculate the statistics. See discussion below.

Details

Care is taken in this function to match multiple verses that are translated into one verse, see bibles for a survey of the encoding assumptions taken here.

The parameter method can take anything that is also available for assocSparse. Similarities are computed using that function.

When weight is specified, the similarities are computed using cosSparse with default setting of norm = norm2. All available weights can also be used here.

The option best = T uses rowMax and colMax. This approach to get the 'best' translation is really crude, but it works reasonably well with one-to-one and many-to-one situations. This option takes

rather a lot more time to finish, as row-wise maxima for matrices is not trivial to optimize. Consider raising `tol`, as this removes low values that won't be important for the maxima anyway. See examples below.

Guidelines for the value of `tol` are difficult to give, as it depends on the method used, but also on the distribution of the data (i.e. the number of sentences, and the frequency distribution of the words in the text). Some suggestions:

- when `weight` is specified, results range between -1 and +1. Then `tol = 0.1` should never lead to problems, but often even `tol = 0.3` or higher will lead to identical results.
- when `weight` is not specified (i.e. `assocSparse` will be used), then results range between `-inf` and `+inf`, so the tolerance is more problematic. In general, `tol = 2` seems to be unproblematic. Higher tolerance, e.g. `tol = 10` can be used to find the 'obvious' translations, but you will loose some of the more incidental co-occurrences.

Value

When `best = F`, a single sparse matrix is returned of type `dgCMatrix` with the values of the statistic chosen. All unique wordforms of text1 are included as row names, and those from text2 as column names.

When `best = T`, a list of two sparse matrices is returned:

`sim`	the same matrix as above
`best`	a sparse pattern matrix of type `ngCMatrix` with the same dimensions as the previous matrix. Only the 'best' translations between the two languages are marked

Author(s)

Michael Cysouw

References

Mayer, Thomas and Michael Cysouw. 2012. Language comparison through sparse multilingual word alignment. *Proceedings of the EACL 2012 Joint Workshop of LINGVIS & UNCLH*, 54–62. Avignon: Association for Computational Linguistics.

See Also

`splitText`, `assocSparse` and `cosSparse` are the central parts of this function. Also check `rowMax`, which is used to extract the 'best' translations.

Examples

```
data(bibles)

# ----- small example of co-occurrences -----

# as an example, just take partially overlapping parts of two bibles
# sim.words uses the names to get the paralellism right, so this works
eng <- bibles$eng[1:5000]
```

```
deu <- bibles$deu[2000:7000]
sim <- sim.words(eng, deu, method = res)

# but the statistics are not perfect (because too little data)
# sorted co-occurrences for the english word "your" in German:
sort(sim["your",], decreasing = TRUE)[1:10]

# ----- complete example of co-occurrences -----

## Not run:
# running the complete bibles takes a bit more time (but still manageable)
system.time(sim <- sim.words(bibles$eng, bibles$deu, method = res))

# results are much better
# sorted co-occurrences for the english word "your" in German:
sort(sim["your",], decreasing = TRUE)[1:10]

# ----- look for 'best' translations -----

# note that selecting the 'best' takes even more time
system.time(sim2 <- sim.words(bibles$eng, bibles$deu, method = res, best = TRUE))

# best co-occurrences for the English word "your"
which(sim2$best["your",])

# but can be made faster by removing low values
# (though the boundary in \code{tol =  5} depends on the method used
system.time(sim3 <- sim.words(bibles$eng, bibles$deu, best = TRUE, method = res, tol = 5))

# note that the decision on the 'best' remains the same here
all.equal(sim2$best, sim3$best)

# ----- computations also work with other languages -----

# All works completely language-independent
# translations for 'we' in Tagalog:
sim <- sim.words(bibles$eng, bibles$tgl, best = TRUE, weight = idf, tol = 0.1)
which(sim$best["we",])

## End(Not run)
```

---

splitStrings *Construct unigram and bigram matrices from a vector of strings*

Description

A (possibly large) vector of strings is separated into sparse pattern matrices, which allows for efficient computation on the strings.

Usage

```
splitStrings(strings, sep = "", bigrams = TRUE, boundary = TRUE,
bigram.binder = "", gap.symbol = "\u2043", left.boundary = "#",
right.boundary = "#", simplify = FALSE)
```

Arguments

strings	Vector of strings to be separated into sparse matrices
sep	Separator used to split the strings into parts. This will be passed to strsplit internally, so there is no fine-grained control possible over the splitting. If it is important to get the splitting exactly right, consider pre-processing the splitting by inserting a special symbol on the split-positions, and then choosing to split by this specific symbol.
bigrams	By default, both unigrams and bigrams are computer. If bigrams are not needed, setting bigrams = F will save on resources.
boundary	Should a start symbol and a stop symbol be added to each string? This will only be used for the determination of bigrams, and will be ignored if bigrams = F.
bigram.binder	Only when bigrams = T. What symbol(s) should occur between the two parts of the bigram?
gap.symbol	Only when bigram = T. What symbol should be included to separate the strings? It defaults to U+2043 HYPHEN BULLET on the assumption that this character will not often be included in data. See pwMatrix for some more explanation about the necessity of this gap symbol.
left.boundary, right.boundary	Symbols to be used as boundaries, only used when boundary = T.
simplify	By default, various vectors and matrices are returned. However, when simplify = T, only a single sparse matrix is returned. See Value.

Value

By default, the output is a list of six elements:

segments	A vector with all splitted parts (i.e. all tokens) in order of occurrence, separated between the original strings with gap symbols.
unigrams	A vector with all unique parts occuring in the segments.
bigrams	Only present when bigrams = T. A vector with all unique bigrams.
SW	A sparse pattern matrix of class ngCMatrix specifying the distribution of segments (S) over the original strings (W, think 'words'). This matrix is only interesting in combination with the following matrices.
US	A sparse pattern matrix of class ngCMatrix specifying the distribution of the unique unigrams (U) over the tokenized segments (S).
BS	Only present when bigrams = T. A sparse pattern matrix of class ngCMatrix specifying the distribution of the unique bigrams (B) over the tokenized segments (S)

When simplify = T the output is a single sparse matrix of class dgCMatrix. This is basically BS %8% SW (when bigrams = T) or US %*% SW (when bigrams = F) with rows and column names added into the matrix.

Note

Because of some internal idiosyncrasies, the ordering of the bigrams is first by second element, and then by first element. This might change in future versions.

Author(s)

Michael Cysouw

See Also

sim.strings is a convenience function to quickly compute pairwise strings similarities, based on splitStrings.

Examples

```
# a simple example to see the function at work
example <- c("this","is","an","example")
splitStrings(example)
splitStrings(example, simplify = TRUE)

## Not run:
# a bit larger, but still quick and efficient
# taking 15526 wordforms from the English Dalby Bible and splitting them into bigrams
data(bibles)
words <- splitText(bibles$eng)$wordforms
system.time( S <- splitStrings(words, simplify = TRUE) )

# and then taking the cosine similarity between the bigram-vectors for all word pairs
system.time( sim <- cosSparse(S) )

# most similar words to "father"
sort(sim["father",], decreasing = TRUE)[1:20]

## End(Not run)
```

---

splitTable *Construct sparse matrices from a nominal matrix/dataframe*

Description

This function splits a matrix or dataframe into two sparse matrices: an incidence and an index matrix. The incidence matrix links the observations (rows) to all possible values that occur in the original matrix. The index matrix links the values to the attributes (columns). This encoding allows for highly efficient calculations on nominal data.

Usage

```
splitTable(data,
    attributes = colnames(data), observations = rownames(data),
name.binder = ":", split = NULL)
```

Arguments

data	a matrix (or data.frame) with observations as rows and nominal attributes as columns. Numerical values in the data will be interpreted as classes (i.e. as nominal data, aka categorical data).
attributes, observations	The row names and column names of the data will by default be extracted from the input matrix. However, in special situations they can be added separately. Note that names of the attributes ('column names') are needed for the production of unique value names. In case of absent column names, new column names of the form 'X1' are automatically generated.
name.binder	Character string to be added between attribute names and value names. Defaults to colon ':'.
split	Character string to split values in each cell of the table, e.g. a comma or semi-colon.

Value

A list containing the various row and column names, and the two sparse pattern matrices of format ngCMatrix:

attributes	vector of attribute names
values	vector of unique value names
observations	vector of observation names
OV	sparse pattern matrix with observations as rows (O) and values as columns (V)
AV	sparse pattern matrix with attributes as rows (A) and values as columns (V)

Note

Input of data as a matrix or as a data.frame might lead to different ordering of the values because collation differs per locale (see the discussion at ttMatrix, which does the heavy lifting here).

The term 'attribute' is used in instead of the more common term 'variable' to allow for the capital A to uniquely identify attributes and V to identify values.

Author(s)

Michael Cysouw

See Also

More methods to use such split tables can be found at sim.nominal.

Examples

```
# start with a simple example from the MASS library
# compare the original data with the encoding as sparse matrices
library(MASS)
farms
splitTable(farms)

# As a more involved example, consider the WALS data included in this package
# Transforming the reasonably large WALS data.frame \code{wals$data} is fast
# (2566 observations, 131 attributes, 630 unique values)
# The function `str' gives a useful summary of the result of the splitting
data(wals)
system.time(W <- splitTable(wals$data))
str(W)

# Some basic use examples on the complete WALS data.
# The OV-matrix can be used to quickly count the number of similarities
# between all pairs of observations. Note that with the large amount of missing values
# the resulting numbers are not really meaningfull. Some normalisation is necessary.
system.time( O <- tcrossprod(W$OV*1) )
O[1:10,1:10]

# The number of comparisons available for each pair of attributes
system.time( N <- crossprod(tcrossprod(W$OV*1, W$AV*1)) )
N[1:10,1:10]

# compute the number of available datapoints per observation (language) in WALS
# once the sparse matrices W are computed, such calculations are much quicker than 'apply'
system.time( avail1 <- rowSums(W$OV) )
system.time( avail2 <- apply(wals$data,1,function(x){sum(!is.na(x))}))
names(avail2) <- NULL
all.equal(avail1, avail2)

# Very unequal availability of data over languages in WALS
hist(avail1)
```

`splitText` *Construct sparse matrices from parallel texts*

Description

Convenience functions to read parallel texts and to split parallel texts into sparse matrices.

Usage

```
splitText(text, globalSentenceID = NULL, localSentenceID = names(text), sep = " ",
simplify = FALSE, lowercase = TRUE)

read.text(file)
```

Arguments

text	vector of strings, typically sentences with wordforms separated by space (see sep below). names of the vector elements are typically IDs to link across texts (cf. the format as used in bibles).
globalSentenceID	Vector of all IDs that might possibly occur in the parallel texts, used to parallelize the texts. Can for example be constructed by using union on the localSentenceIDs.
localSentenceID	Vector of the IDs for the actual sentences in the present text. Typically present as names of the text.
sep	Separator on which the sentences should be parsed into wordforms. The implementation is very simple here, there are no advanced options for guessing punctuation. The variation in punctuation across a wide variety of languages and scripts normally turns out to be too large to be easily automatically parsed. Any advanced parsing has to be done externally, and here simply the parsed symbol is used to actually split the text into parts. Typically, this parsing of sentences into wordforms will be performed using space sep = " ". See also bibles for some examples of such pre-parsing.
simplify	By default (when simplify = F), this function returns a list of objects that represent the encoding of the text into sparse matrices. With simplify = T this list is reduced to a single matrix (wordforms x globalSentenceID), with the actual wordforms as row names.
lowercase	By default, a mapping between the text and a lowercase version of the same text. In the default output (with simplify = F), this is a sparse matrix linking strings with mixed upper/lower case to string with only lower case. Note that case folding is locale-specific, but here a simple universal case-folding is used (as available through tolower).
file	file name (or full path) for a file to be read.

Details

The function splitText is actually just a nice examples of how pwMatrix, jMatrix, and ttMatrix can be used to work with parallel texts.

The function read.text is a convenience function to read parallel texts as prepared in the paralleltext.info project.

Value

When simplify = F, a list is returned with the following elements:

runningWords	single vector with complete text (ignoring original sentence breaks), separated into strings according to sep
wordforms	vector with all wordforms as attested in the text (according to the specified separator). Ordering of wordforms is done by ttMatrix, which by default uses the "C" collation locale.

lowercase	only returned when lowercase = T. Vector with all unique wordforms after conversion to lowercase.
RS	Sparse pattern matrix of class ngCMatrix with runningWords (R) as rows and sentence IDs (S) as columns. When globalSentenceID = NULL, then the sentences are the elements of the original text. Else, they are the specified globalSentenceIDs.
WR	Sparse pattern matrix of class ngCMatrix with wordforms (W) as rows and running words (R) as columns.
wW	only returned when lowercase = T. Sparse pattern matrix of class ngCMatrix linking between lowercased wordforms and original wordforms.

When simplify = T the result is a single sparse Matrix (of type dgCMatrix) linking wordforms (either with or without case) to sentences (either global or local). Note that the result with options (simplify = T, lowercase = F) will result in the sparse matrix as available at paralleltext.info (there the matrix is in .mtx format), with the wordforms included into the matrix as row names. However, note that the resulting matrix from the code here will include frequencies for words that occur more than once per sentence. These have been removed for the .mtx version available online.

Author(s)

Michael Cysouw

See Also

bibles for some texts that led to the development of this function. sim.words for a convenience function to easily extract possible translations equivalents through co-occurrence (using splitText for the data-preparation.)

Examples

```
# a trivial examples to see the results of this function:
text <- c("This is a sentence .","A sentence this is !","Is this a sentence ?")
splitText(text)
splitText(text, simplify = TRUE, lowercase = FALSE)

## Not run:
# reasonably quick with complete bibles (about 1-2 second per complete bible)
# texts with only New Testament is even quicker
data(bibles)
system.time(eng <- splitText(bibles$eng, bibles$verses))
system.time(deu <- splitText(bibles$deu, bibles$verses))

# Use example: Number of co-ocurrences between two bibles
# How often do words from the one language cooccur with words from the other language?
ENG <- (eng$wW * 1) %*% (eng$WR * 1) %*% (eng$RS * 1)
DEU <- (deu$wW * 1) %*% (deu$WR * 1) %*% (deu$RS * 1)
C <- tcrossprod(ENG,DEU)
rownames(C) <- eng$lowercase
colnames(C) <- deu$lowercase
C[ c("father","father's","son","son's"),
```

```
c("vater","vaters","sohn","sohne","sohnes","sohns")
]

# Pure counts are not very interesting. This is better:
R <- assocSparse(t(ENG), t(DEU))
rownames(R) <- eng$lowercase
colnames(R) <- deu$lowercase
R[ c("father","father's","son","son's"),
c("vater","vaters","sohn","sohne","sohnes","sohns")
]

# For example: best co-occurrences for the english word "mine"
sort(R["mine",], decreasing = TRUE)[1:10]

# To get a quick-and-dirty translation matrix:
# adding maxima from both sides work quite well
best <-  colMax(R, which = TRUE, ignore.zero = FALSE)$which
+ rowMax(R,which = TRUE, ignore.zero = FALSE)$which
best <- as(best, "nMatrix")

which(best["your",])
which(best["went",])

# all of the above is also performed by the function sim.words

# A final speed check:
# split all 4 texts, and simplify them into one matrix
# They have all the same columns, so they can be rbind
system.time(all <- sapply(bibles[-1], function(x){splitText(x, bibles$verses, simplify = TRUE)}))
all <- do.call(rbind, all)

# then try a single co-occerrence measure on all pairs from these 72K words
# (so we are doing about 2.6e9 comparisons here!)
system.time( S <- cosSparse(t(all)) )

# this goes extremely fast! As long as everything fits into RAM this works nicely.
# Note that S quickly gets large
print(object.size(S), units = "auto")

# but most of it can be thrown away, because it is too low anyway
# this leads to a factor 10 reduction in size:
S <- drop0(S, tol = 0.2)
print(object.size(S), units = "auto")

## End(Not run)
```

`splitWordlist` *Construct sparse matrices from comparative wordlists (aka 'Swadesh list')*

Description

A comparative wordlist (aka 'Swadesh list') is a collection of wordforms from different languages, which are translations of a selected set of meanings. This function dismantles this data structure into a set of sparse matrices.

Usage

```
splitWordlist(data,
doculects = "DOCULECT", concepts = "CONCEPT", counterparts = "COUNTERPART",
splitstrings = TRUE, sep =  "", bigram.binder = "", grapheme.binder = "_",
simplify = FALSE)
```

Arguments

`data`	A dataframe or matrix with each row describing a combination of language (DOCULECT), meaning (CONCEPT) and translation (COUNTERPART).
`doculects, concepts, counterparts`	The name (or number) of the column of `data` in which the respective information is to be found. The defaults are set to coincide with the naming of the example dataset included in this package: `huber`.
`splitstrings`	Should the counterparts be separated into unigrams and bigrams (using `splitStrings`)?
`sep`	Separator to be passed to `splitStrings` to specify where to split the strings. Only used when `splitstrings = T`, ignored otherwise.
`bigram.binder`	Separator to be passed to `splitStrings` to be inserted between the parts of the bigrams
`grapheme.binder`	Separator to be used to separate a grapheme from the language name. Graphemes are language-specific symbols (i.e. the 'a' in the one language is not assumed to be the same as the 'a' from another language).
`simplify`	Should the output be reduced to the most important matrices only, with the row and columns names included in the matrices? Defaults to `simplify = F`, separating everything into different object. See Value below for details on the format of the results.

Details

The meanings that are selected for a wordlist are called CONCEPTS here, and the translations into the various languages COUNTERPARTS (following Poornima & Good 2010). The languages are called DOCULECTS ('documented lects') to generalize over their status as dialects, languages, or even small families (following Cysouw & Good 2013).

Value

There are four different possible outputs, depending on the option chosen.

By default, when `splitstrings = T, simplify = F`, the following list of 15 objects is returned. It starts with 8 different character vectors, which are actually the row/column names of the following sparse pattern matrices. The naming of the objects is an attempt to make everything easy to remember.

doculects	Character vector with names of doculects in the data
concepts	Character vector with names of concepts in the data
words	Character vector with all words, i.e. unique counterparts per language. The same string in the same language is only included once, but an identical string occurring in different doculect is separately included for each doculects.
segments	Character vector with all unigram-tokens in order of appearance, including boundary symbols and gap symbols (see splitStrings for more information about the gap symbols)
unigrams	Character vector with all unique unigrams in the data
bigrams	Character vector with all unique bigrams in the data
graphemes	Character vector with all unique graphemes (i.e. combinations of unigrams+doculects) occurring in the data
digraphs	Character vector with all unique digraphs (i.e. combinations of bigrams+doculects) occurring in the data
DW	Sparse pattern matrix of class ngCMatrix linking doculects (D) to words (W)
CW	Sparse pattern matrix of class ngCMatrix linking concepts (C) to words (W)
SW	Sparse pattern matrix of class ngCMatrix linking all token-segments (S) to words (W)
US	Sparse pattern matrix of class ngCMatrix linking unigrams (U) to segments (S)
BS	Sparse pattern matrix of class ngCMatrix linking bigrams (B) to segments (S)
GS	Sparse pattern matrix of class ngCMatrix linking language-specific graphemes (G) to segments (S)
TS	Sparse pattern matrix of class ngCMatrix linking digraphs (T, as no other letter was available) to segments (S)

When splitstrings = F, simplify = F, only the following objects from the above list are returned:

doculects	Character vector with names of doculects in the data
concepts	Character vector with names of concepts in the data
words	Character vector with all words, i.e. unique counterparts per language. The same string in the same language is only included once, but an identical string occurring in different doculect is separately included for each doculects.
DW	Sparse pattern matrix of class ngCMatrix linking doculects (D) to words (W)
CW	Sparse pattern matrix of class ngCMatrix linking concepts (C) to words (W)

When splitstrings = T, simplify = T only the bigram-separation is returned, and all row and columns names are included into the matrices. However, for reasons of space, the words vector is only included once:

DW	Sparse pattern matrix of class ngCMatrix linking doculects (D) to words (W). Doculects are in the rownames, colnames are left empty.
CW	Sparse pattern matrix of class ngCMatrix linking concepts (C) to words (W). Concepts are in the rownames, colnames are left empty.

BW — Sparse pattern matrix of class ngCMatrix linking bigrams (B) to words (W). Bigrams (note: not digraphs!) are in the rownames. This matrix includes all words as colnames.

Finally, when splitstrings = F, simplify = T, only the following subset of the above is returned.

DW — Sparse pattern matrix of class ngCMatrix linking doculects (D) to words (W). Doculects are in the rownames, colnames are left empty.

CW — Sparse pattern matrix of class ngCMatrix linking concepts (C) to words (W). Concepts are in the rownames, colnames are left empty.

Note

Note that the default behavior probably overgenerates information (specifically when splitstrings = T), and might be performing unnecessary computation for specific goals. In practice, it might be useful to tweak the underlying code (mainly by throwing out unnecessary steps) to optimize performance.

Author(s)

Michael Cysouw

References

Cysouw, Michael & Jeff Good. 2013. Languoid, Doculect, Glossonym: Formalizing the notion "language". *Language Documentation and Conservation* 7. 331-359.

Poornima, Shakthi & Jeff Good. 2010. Modeling and Encoding Traditional Wordlists for Machine Applications. *Proceedings of the 2010 Workshop on NLP and Linguistics: Finding the Common Ground.*

See Also

sim.wordlist for various quick similarities that can be computed using these matrices.

Examples

```
# ----- load data -----

# an example wordlist, see the help(huber) for details
data(huber)

# ----- show output -----

# a selection, to see the result of splitWordlist
# only show the simplified output here,
# the full output is rather long even for just these six words
sel <- c(1:3, 1255:1258)
splitWordlist(huber[sel,], simplify = TRUE)

## Not run:
```

```
# ----- split complete data -----

# splitting the complete wordlist is a lot of work !
# it won't get much quicker than this
# most time goes into the string-splitting of the almost 26,000 words
# Default version, included splitStrings:
system.time( H <- splitWordlist(huber) )

# Simplified version without splitStrings is much quicker:
system.time( H <- splitWordlist(huber, splitstrings = FALSE, simplify = TRUE) )

# ----- investigate colexification -----

# The simple version can be used to check how often two concepts
# are expressed identically across all languages ('colexification')
H <- splitWordlist(huber, splitstrings = FALSE, simplify = TRUE)
sim <- tcrossprod(H$CW*1)

# select only the frequent colexifications for a quick visualisation
diag(sim) <- 0
sim <- drop0(sim, tol = 5)
sim <- sim[rowSums(sim) > 0, colSums(sim) > 0]
plot( hclust(as.dist(-sim), method = "average"), cex = .5)

# ----- investigate regular sound correspondences -----

# One central problem with data from many languages is the variation of orthography
# It is preferred to solve that problem separately
# e.g. check the column "TOKENS" in the huber data
# This is a grapheme-separated version of the data.
# can be used to investigate co-occurrence of graphemes (approx. phonemes)
H <- splitWordlist(huber, counterparts = "TOKENS", sep = " ")

# co-occurrence of all pairs of the 2150 different graphemes through all languages
system.time( G <- assocSparse( (H$CW*1) %*% t(H$SW*1) %*% t(H$GS*1), method = poi))
rownames(G) <- colnames(G) <- H$graphemes
G <- drop0(G, tol = 1)

# select only one language pair for a quick visualisation
# check the nice sound changes between bora and muinane!
GD <- H$GS %*% H$SW %*% t(H$DW)
colnames(GD) <- H$doculects
correspondences <- G[GD[,"bora"],GD[,"muinane"]]
heatmap(as.matrix(correspondences))

## End(Not run)
```

---

ttMatrix *Construct a 'type-token' (tt) Matrix from a vector*

Description

A type-token matrix is a sparse matrix representation of a vector of entities. The rows of the matrix ('types') represent all different entities in the vector, and the columns of the matrix ('tokens') represent the entities themselves. The cells in the matrix represent which token belongs to which type. This is basically a convenience wrapper around `factor` and `sparseMatrix`, with an option to influence the ordering of the rows ('types') based on locale settings.

Usage

```
ttMatrix(vector, collation.locale = "C", simplify = FALSE)
```

Arguments

`vector`	a vector of tokens to be represented as a sparse matrix. It will work without complaining just as well when given a factor, but be aware that the ordering of the levels in the factor depends on the locale, which is transparently handled by this function. So better let this function turn the vector into a factor.
`simplify`	by default, the row and column names are not included into the matrix to keep the matrix as lean as possible. The row names ('types') are returned separately. Using `simplify = T` the row and columns names will be added into the matrix. Note that the column names are simply the vector that went into the function.
`collation.locale`	locale determining the ordering ('collation') of the entities. By default R mostly uses 'en_US.UTF-8', though this might depend on the installation. By default, this function sets the ordering to 'C', which means that characters are ordered according to their Unicode-number. For more information about locale settings, see `locales`.

Details

This function is a rather low-level preparation for later high level functions. A few simple uses are described in the examples.

Value

By default (`simplify = F`), then the output is a list with two elements:

`M`	sparse pattern Matrix of type `ngCMatrix`. Because of the structure of these matrices, row-based encoding would be slightly more efficient. If RAM is crucial, consider storing the matrix as its transpose
`rownames`	a separate vector with the names of the types in the order of occurrence in the matrix. This vector is separated from the matrix itself for reasons of efficiency when dealing with many matrices.

When `simplify = T`, then only the matrix M with row and columns names is returned.

Author(s)

Michael Cysouw

See Also

This function is used in various high-level functions like pwMatrix, splitText, splitTable and splitWordlist.

Examples

```
# Consider two nominal variables
# one with eight categories, and one with three categories
var1 <- sample(8, 1000, TRUE)
var2 <- sample(3, 1000, TRUE)

# turn them into type-token matrices
M1 <- ttMatrix(var1, simplify = TRUE)
M2 <- ttMatrix(var2, simplify = TRUE)



# Then taking  the `residuals' from assocSparse ...
x <- as.matrix(assocSparse(t(M1), t(M2), method = res))

# ... is the same as the residuals as given by a chi-square
x2 <- chisq.test(var1, var2)$residuals
class(x2) <- "matrix"
all.equal(x, x2, check.attributes = FALSE) # TRUE

# A second quick example: consider a small piece of English text:
text <- "Once upon a time in midwinter, when the snowflakes were
falling like feathers from heaven, a queen sat sewing at her window,
which had a frame of black ebony wood. As she sewed she looked up at the snow
and pricked her finger with her needle. Three drops of blood fell into the snow.
The red on the white looked so beautiful that she thought to herself:
If only I had a child as white as snow, as red as blood, and as black
as the wood in this frame. Soon afterward she had a little daughter who was
as white as snow, as red as blood, and as black as ebony wood, and therefore
they called her Little Snow-White. And as soon as the child was born,
the queen died."

# split by characters, make lower-case, and turn into a type-token matrix
split.text <- tolower(strsplit(text,"")[[1]])
M <- ttMatrix(split.text, simplify = TRUE)

# rowSums give the character frequency
freq <- rowSums(M)
names(freq) <- rownames(M)
sort(freq, decreasing = TRUE)

# shift the matrix one character to the right using a bandSparse matrix
S <- bandSparse(n = ncol(M), k = 1)
N <- M %*% S

# use rKhatriRao on M and N to get frequencies of bigrams
```

```
B <- rKhatriRao(M, N, binder = "")
freqB <- rowSums(B$M)
names(freqB) <- B$rownames
sort(freqB, decreasing = TRUE)

# then the association between N and M is related
# to the transition probabilities between the characters.
P <- assocSparse(t(M), t(N))
plot(hclust(as.dist(-P), method = "ward"))
```

`unfold` *Unfolding of Arrays*

Description

Multidimensional Arrays ("Tensors") can be unfolded, i.e. multiple dimensions can be combined into a single dimension in a block-wise fashion. Such unfoldings are central to tensor decomposition. In general, computations on tensors are regularly performed by reducing tensors to matrices ("2-dimensional tensors") and then use regular matrix algebra.

Usage

```
unfold(x, MARGINS)

unfold_to_matrix(x, ROWS, COLS = NULL)
tenmat(x, ROWS, COLS = NULL)
```

Arguments

`x`	Sparse array to be unfolded, using `simple_sparse_array` from the package `spam`.
`MARGINS`	Margins ("dimensions") to be unfolded. The margins specified will be turned into a single dimension, to be added as the last dimension of the resulting array (see Details).
`ROWS`	Margins of the original array to be unfolded into the rows of the resulting matrix.
`COLS`	Margins of the original array to be unfolded into the columns of the resulting matrix. If `NULL`, then all remaining margins, not included in `ROWS` are unfolded here.

Details

The function `unfold` is a general approach to combining of multiple dimensions into a single dimensions. The function `unfold_to_matrix` is a special case in which the result is a 2-dimensional matrix. This second function is made to emulate the functionality of the `tenmat` ("tensor to matrix") from the Matlab Tensor Toolbox. For convenience, the function-name `tenmat` is also added as a synonym for `unfold_to_matrix`.

Unfolding basically works by interspercing margins subsequently. E.g. margin A of size 3 (A1, A2, A3) and a margin B of size 2 (B1, B2) are unfolded through `c(A,B)` as (A1B1, A2B1, A3B1, A1B2, A2B2, A3B2), but they are unfolded through `c{B,A}` as (B1A1, B2A1, B1A2, B2A2, B1A3, B2A3).

Value

`unfold` returns a `simple_sparse_array` with the new combined dimension added as the last dimension. All original dimensions are shifted forward. The relation between the original dimensions and the new dimensions is stored as an `permutation` attribute, e.g. try `attr(x, "p")`. When multiple unfoldings are performed after each other, these permutations can be subsetted on each other to obtain the final permutation. See examples below.

`unfold_to_matrix` and `tenmat` return a sparse matrix of class `dgTMatrix`.

Author(s)

Michael Cysouw <cysouw@mac.com>

References

see http://www.cs.cornell.edu/cv/SummerSchool/Unfold.pdf for some notes on tensor unfolding. The Matlab Tensor Toolbox can be found at http://www.sandia.gov/~tgkolda/TensorToolbox/index-2.6.html. A newer Matlab implementation is http://www.tensorlab.net.

Examples

```
# example from \url{http://www.cs.cornell.edu/cv/SummerSchool/Unfold.pdf}
x <- array(c(111, 211, 311, 411, 121, 221, 321,
    421, 131, 231, 331, 431, 112, 212, 312, 412,
    122, 222, 322, 422, 132, 232, 332, 432), dim = c(4, 3, 2))
x

s <- as.simple_sparse_array(x)
( s1  <- as.array(unfold_to_matrix(s,1)) )

# note this is identical to:
( s23 <- as.array(unfold(s,c(2,3))) )
all.equal(s23, s1)

# larger example from same source
x <- array(0, dim = c(2,3,2,2,3))
x[1,2,1,1,2] <- 12112
x[2,3,1,2,2] <- 23122
x[2,2,2,1,1] <- 22211
x[2,2,1,2,3] <- 22123
s <- as.simple_sparse_array(x)

as.array(unfold_to_matrix(s, c(1,2,3), c(4,5)))

# use attribute "permutation" to track dimensions
# first step: unfold 1,2,3 to become dimension 3
```

```
# original dimensions 4,5 now become 1,2
s1 <- unfold(s, c(1,2,3))
( p1 <- attr(s1, "permutation") )

# now take these dimension 1,2 (originally 4,5) and unfold them
s2 <- unfold(s1, c(1,2))
( p2 <- attr(s2, "permutation") )

# use subsetting to track dimensions through subsequent unfolding
p2[p1]
```

`unfoldBlockMatrix` *Unfolding of block matrices (sparse matrices)*

Description

Utility function for some matrix manipulations for sparse block matrices.

Usage

```
unfoldBlockMatrix(X, colGroups, rowGroups = NULL)
```

Arguments

`X`	Sparse block matrix to be unfolded into the individual blocks
`colGroups, rowGroups`	either vectors with group indices of the columns and rows, or sparse pattern matrices with the groups as rows and the columns/rows of the X matrix as columns. See example below.

Details

For some sparse manipulation it turns out the profitable to 'unfold' sparse block matrices, i.e. to separate the blocks into their own rows and columns. Each block can then separately be manipulated by using matrix products with diagonal matrices (see example below). For convenience, the function also returns two matrices to 'refold' the unfolded matrix. Specifically, X = L %*% U %*% R

Value

When `rowGroups != NULL` then the result is a list of three matrices:

`U`	The unfolded block matrix
`L`	The left matrix for refolding the unfolded matrix
`R`	The right matrix for refolding the unfolded matrix

When `rowGroups = NULL` then the R matrix is not returned, and the refolding works with only the L matrix: X = L %*% U.

Note

The use of kronecker for sparse matrices in this function often leads to warnings about the sparse format of the resulting matrices. These warnings can be ignored.

Author(s)

Michael Cysouw

See Also

This is used in the sparse computation of assocCol to divide each block by a different N.

Examples

```
# specially prepared block matrix. For illustration purpuse, this one is not sparse
( X <- Matrix( c( rep(c(1,1,1,1,2,2,3,3,3,4),3),
rep(c(5,5,5,5,6,6,7,7,7,8),2)),10,5, sparse = TRUE) )

# this matrix has two column groups, and four row groups
# groups can be specified as sparse matrices, or as grouping vectors
colG <- ttMatrix(c(1,1,1,2,2))$M*1
rowG <- ttMatrix(c(1,1,1,1,2,2,3,3,3,4))$M*1

# unfold the matrix, with the result that each block has it's own rows/columns
# the $L and $R matrices can be used to refold the matrix to it's original state
( M <- unfoldBlockMatrix(X, colG, rowG) )

# unfold and refold back: result is identical with M
with(M, all.equal(X, L %*% U %*% R) )

# Unfolded, each block can be easily reached for computations using diagonal matrices
# for example, multiply each block by the sum of its cells, and then fold back again
# this is a trick to apply computations to blocks in sparse block matrices
sums <- drop(crossprod(kronecker(Diagonal(nrow(colG)),rowG)) %*% rowSums(M$U))
S <- Diagonal( x = sums )

with(M, L %*% S %*% U %*% R )
```

WALS *The World Atlas of Language Structures (WALS)*

Description

The World Atlas of Language Structures (WALS) is a large database of structural (phonological, grammatical, lexical) properties of languages gathered from descriptive materials (such as reference grammars) by a team of 55 authors.

The first version of WALS was published as a book with CD-ROM in 2005 by Oxford University Press. The first online version was published in April 2008. The second online version was published in April 2011. The current dataset is WALS 2013, published on 14 November 2013.

The included dataset `wals` takes a somewhat sensible selection from the complete WALS data. It excludes attributes ("features" in WALS-parlance) that are definitially duplicates of others (3, 25, 95, 96, 97), those attributes that only list languages that are incompatible with other attributes (132, 133, 134, 135, 139, 140, 141, 142), and the 'additional' attributes that are marked as 'B' through 'Z'. Further, it removes those languages that do not have any data left after removing those attributes. The result is a dataset with 2566 languages and 131 attributes.

Usage

```
data(wals)
```

Format

A list with two dataframes:

`data` the actual WALS data. The object `wals$data` contains a dataframe with data from 2566 languages on 131 different attributes. The column names identify the WALS features. For details about these features, see http://wals.info/chapter

`meta` some metadata for the languages. The object `wals$meta` contains a dataframe with some limited meta-information about these 2566 languages.

The three-letter WALS-codes are used as rownames in both dataframes. Further, the object `wals$meta` contains the following variables.

`name` a character vector giving a name for each language

`genus` a factor with 522 levels with the genera according to M. Dryer

`family` a factor with 215 levels with the families according to M. Dryer

`longitude` a numeric vector with geo coordinates for all languages

`latitude` a numeric vector with geo coordinates for all languages

Details

All details about the meaning of the variables and much more meta-information is available at http://wals.info.

Source

The current data was downloaded from http://wals.info in May 2014. The data is licensed as http://creativecommons.org/licenses/by-nc-nd/2.0/de/deed.en. Some minor corrections on the metadata have been performed (naming of variables, addition of missing coordinates).

References

Dryer, Matthew S. & Haspelmath, Martin (eds.) 2013. *The World Atlas of Language Structures Online.* Leipzig: Max Planck Institute for Evolutionary Anthropology. (Available online at http://wals.info, Accessed on 2013-11-14.)

Examples

```
data(wals)

# plot all locations of the WALS languages, looks like a world map
plot(wals$meta[,4:5])

# turn the large and mostly empty dataframe into sparse matrices
# recoding is nicely optimized and quick for this reasonably large dataset
# this works perfect as long as things stay within available RAM of the computer
system.time(
  W <- splitTable(wals$data)
)

# as an aside: note that the recoding takes only about 30% of the space
as.numeric( object.size(W) / object.size(wals$data) )

# compute similarities (Chuprov's T, similar to Cramer's V)
# between all pairs of variables using sparse Matrix methods
system.time(sim <- sim.att(wals$data, method = "chuprov"))

# some structure visible
rownames(sim) <- colnames(wals$data)
plot(hclust(as.dist(1-sim), method = "ward"), cex = 0.5)
```

Index