Package 'akin'

June 15, 2026

Type Package

Title Functional Utilities for Data Processing

Version 0.3.7

Description Covers several areas of data processing: batch-splitting, reading and writing of large data files, data tiling, one-hot encoding and decoding of data tiles, stratified proportional (random or probabilistic) data sampling, data normalization and thresholding, substring location and commonalities inside strings, and location and tabulation of amino acids, modifications or associated monoisotopic masses inside modified peptides. The extractor implements code from 'Matrix.utils', Varrichio C (2020), <https://cran.r-project.org/package=Matrix.utils>.

License GPL (>= 3)

Imports data.table (>= 1.18.2.0), RVerbalExpressions (>= 0.1.0), methods

Depends R (>= 4.1.0), callr, fastmatch, listenv, Matrix, RcppAlgos, utils

Encoding UTF-8

Suggests testthat (>= 3.0.0)

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

ByteCompile yes

Author Dragos Bandur [aut, cre]

Maintainer Dragos Bandur <dbandur@sympatico.ca>

NeedsCompilation no

Repository CRAN

Date/Publication 2026-06-15 09:00:03 UTC

Contents

`cover` *Calculate The Number Of Combinations Of String Covering*

Description

Calculates the cardinality of substring family (*the covering*) and returns valid elements.

Usage

```
cover(x, valid. = FALSE, wplot = FALSE)
```

Arguments

`x`	integer, length = 1 or character vector, length > 1, or character string. White space is removed
`valid.`	logical, default, FALSE. When TRUE, valid substrings are returned. Requires x as character or string
`wplot`	logical, FALSE. When TRUE, displays a plot of combinations versus number of characters in string covering

Details

This function finds the total, the minimum and the maximum number of combinations of individual string covering or returns a subset of valid substrings (i.e. elements of the string) by retaining even transpositions with *sequential* values inside the combinations matrix. Very long strings may result in memory allocation error when `valid. = TRUE`. Use fcommon instead.

NOTE: Herein, the *covering* represents the substring family - of min. 2 characters each - that preserves the order of characters inside the string. Hence, the use of combinations instead of partial permutations. Valid substrings form a (lower cardinality) subset having elements in sequence (see Example 3).

Value

When `valid. = FALSE` a prettified named vector showing the total, minimum and maximum number of combinations for respective string covering. When `valid. = TRUE`, a character vector of valid substrings having min. 2 characters each. In both cases, a plot is returned when `wplot = TRUE`.

See Also

fcommon, comboCount, comboGeneral

Examples

```
if (interactive()) {

# 1.1 A string
 x = 'tyrvvsvltvlhqdwlngkeykck'

# 1.2 A vector
 y = strsplit(x, '')[[1]]

# 1.3 An integer
 n = length(y)

ll = list(str = x, char = y, int = n)
print(t(
       sapply(ll, cover)
                        ))

# 2. Valid substrings with plot

d = cover(x, TRUE, TRUE)
print(head(d, 30))                          # first 30 substrings

# 3. Valid set versus covering

# The plot shows 3 combinations of 2-character groups while e
# contains 2 sequences of 2 characters each, substring "ac"
# not being a sequence in x.

x = letters[1:3]
e = cover(x, TRUE, TRUE)
print(e)                                    # valid combinations

}
```

---

fcommon *Identify Common Substrings In A Pair Of Strings*

Description

Checks and identifies substrings that are common to a pair of strings.

Usage

```
fcommon(x, y)
```

Arguments

x, y	character, length 1 each: a string, such as a protein chain. y may be missing. White space is removed

Details

This utility identifies common substrings in the x, y pair of strings by isolating *sequences* of identical characters in both strings which then, are packed into substrings and validated. All one-character substrings are removed. When y is missing, x is cleaved at each letter producing all substrings longer than 1 character. Example 1.3 shows that *all existing* common substrings - of min. 2 characters each - are identified.

Value

A sorted character vector of common substrings of min. 2 characters each. When y is missing from call, a sorted character vector of valid substrings in x of min. 2 characters each.

See Also

cover

Examples

```
if (interactive()) {

 # 1. Check for common substrings in the pair below

 x = 'dvvmtqsplslpvtpgepasiscrssqslaktyrvvsvltvlhqdwlngkeykckvv'
 y = 'mtqspltyrvvsvltvlhqdwlngkeykcksnkalpapiektisk'

# 1.1 Common substrings
 system.time(a <- fcommon(x, y))
 print(head(a, 30))

# 1.2 Cleaving (slow on very long strings!)
 system.time(aa <- fcommon(x))
 system.time(bb <- fcommon(y))

# 1.3 Complete identification of common substrings
 A = sort(intersect(aa, bb))                              # common substrings
 identical(a, A)                                          # TRUE

# 2. Different methods for finding valid substrings

x = 'tyrvvsvltvlhqdwlngkeykck'

# 2.1. Combinations matrix (may be limited by character length)
system.time(am <- cover(x, valid. = TRUE))                # valid substrings

# 2.2 String cleaving
system.time(ac <- fcommon(x))                             # valid substrings
```

```
identical(am, ac)                                    # TRUE

}
```

`findLoc` *Find Substring Locations Inside A String*

Description

Finds all locations of a *known* character substring inside a character string.

Usage

```
findLoc(
  subchain,
  chain,
  outlist = FALSE,
  named = FALSE,
  all. = FALSE,
  which = min,
  ignore.case = TRUE,
  perl = FALSE,
  fixed = FALSE,
  useBytes = FALSE
)
```

Arguments

`subchain`	character, length 1, e.g. a peptide sequence. White space is removed
`chain`	(named) character, length 1 or a (named) list of such characters such as a list of protein chains obtained from a *fasta* file. White space is removed
`outlist`	logical. Default, FALSE, the output is a (named) integer vector of locations. Otherwise, it is a (named) list of location vectors, each corresponding to a chain in a list of chains
`named`	logical. Default, FALSE. Output is not named. Otherwise, the output is named
`all.`	logical, default FALSE, returns the leftmost or the rightmost location inside the chain. When TRUE, returns all locations inside the chain for each chain in a list
`which`	symbol. Location to report. Default, **min**. Requires `all. = FALSE`. `which = min` returns the leftmost location inside the chain, e.g. closest to the N-terminus of a protein chain. `which = max` returns the rightmost location, e.g. closest to the C-terminus of a protein chain. Overwritten when `all. = TRUE`
`ignore.case`, `perl`, `fixed`, `useBytes`	arguments to base::gregexpr

Details

Wrapper to base::gregexpr, the function scans all chains in a list of chains to find subchain locations. The location is defined as the position inside the chain relative to the left end of the chain of the first subchain character.

Value

A (named) integer or a (named) list of integer vectors of `subchain` locations inside the `chain`.

See Also

gregexpr

Examples

```
if (interactive()) {

# 1. List of chains

chain = list(chain1 = 'alpdxoipoyloiekladxoipoylyl',
             chain2  = 'kdxoipoylyydxoipoylopldxoipoylac')
subchain = 'DXOIPOYL'                                                  # ignoring the case

findLoc(subchain, chain, outlist = TRUE, named = TRUE, all. = TRUE)    # named list
findLoc(subchain, chain, named = TRUE, all. = TRUE)                    # named integer
findLoc(subchain, chain, outlist = TRUE, named = TRUE)             # the leftmost positions
findLoc(subchain, chain, which = max, named = TRUE)               # the rightmost positions

# 2. Single chain

chain = chain[[1]]

findLoc(subchain, chain, all. = TRUE)
findLoc(subchain, chain, which = max)
findLoc(subchain, chain)                                               # default location
findLoc(subchain, chain, which = max, ignore.case = FALSE)          # not ignoring the case

}
```

`getEV` *Extract Encoded Variables From Encoded Split Or Tiled Data*

Description

Extracts a single encoded variable from a list or listenv of encoded matrices containing multiple encoded variables

Usage

```
getEV(en, name, ...)
```

Arguments

en	a (named) list, or listenv of matrices or a single matrix, all containing multiple encoded variables. See oneHot decoder for lists and matrices containing single encoded variables
name	character, length 1. Column name as found in source data
...	default, empty. Used to convert the class of extracted matrix to 'dgCMatrix' or 'matrix'

Details

This function includes code from package "Matrix.utils" v 0.9.8, published under GPL-3 license, currently removed from CRAN. With thanks to the package Author!

NOTE 1: If name is a source data column name that appears inside other column names, the extracted matrix will combine all encoded matrices having this name inside their column names. Although the extracted matrix is a proper matrix of encodings, it no longer represents *a single* encoded data column. As result, upon decoding, the oneHot decoder will report ambiguous decoding.

NOTE 2: a warning reading either *"single-column encoded matrix for ..."* or *"number of columns of result is not a multiple of vector length (arg 2) ..."* may appear when extracting an encoded categorical variable from a list of encoded matrices. Most likely, this happens with low cardinality encoded variables. The warning signals that most encoded matrices associated with respective variable contain subsets of only one category (level) when, ideally, most of these matrices should contain a mixture of two or more categories or levels; thus, allowing matrix row-binding by category's label. One or more of the following suggestions will solve the issue: a) shuffle the data before encoding, b) increase the number of rows in data chunks when encoding, c) if memory allows, opt for tileHot encoding single matrix output, as shown in Example 2.1, solution c.

Value

A dense or sparse matrix of single encoded variable which can be decoded with the oneHot decoder.

See Also

oneHot, tileHot

Examples

```
if (interactive()) {

# 1. mtcars data have all columns type "double"

data(mtcars)
a = lapply(mtcars, oneHot, encode)                    # encode mtcars data
print(a)                                              # list of sparse matrices
b = getEV(a, 'cyl')                                   # extract encoded "cyl" column
print(b)                                              # a 32x3 sparse matrix
```

```
c = oneHot(b, decode)                                    # revert
identical(mtcars$cyl, c)                              # FALSE. 'mtcars$cyl' is type "double"
isTRUE(all.equal(mtcars$cyl, c))                         # TRUE

# 2. Warnings associated with low cardinality categorical variable

# See tileHot() Examples for full decoding of a dataset

# 2.1 Make 'csv' file
data(iris)                                               # low cardinality "Species"
tempf = tempfile(fileext = '.csv')
write.table(iris, tempf , sep = ',', row.names = FALSE, quote = FALSE)

A = tileHot(readpath = tempf, rows = 14, splits = 3)     # encoded tiles list
print(A[[11]][[5]])                                      # e.g. one-column matrix
a = getEV(A, 'Species')                                  # warning
colSums(a)                                               # incorrect!

# solution b
B = tileHot(readpath = tempf, rows = 60, splits = 3)     # increase number of rows
b = getEV(B, 'Species')                                  # still warning
colSums(b)                                               # incorrect!

# Solution b) could work in combination with solution a)

# solution c
C = tileHot(tempf, rows = 14, splits = 3, orn = TRUE)    # encoded matrix
c = getEV(C, 'Species')                                  # no warning
colSums(c)                                               # correct!

unlink(tempf)

# 2.2 Shuffled 'csv' file
tempf = tempfile(fileext = '.csv')
iris22 = iris[{ set.seed(327); sample.int(150) },]      # shuffled iris data
write.table(iris22, tempf , sep = ',', row.names = FALSE, quote = FALSE)

A = tileHot(readpath = tempf, rows = 14, splits = 3)    # same as above

#solution a
a = getEV(A, 'Species')                                  # no warning
colSums(a)                                               # correct!

unlink(tempf)

}
```

`locateMod` *Locate And Extract Modifications Or Monoisotopic Masses From A Modified Peptide*

Description

Finds and tabulates amino acid modification sites and extracts modifications or monoisotopic masses from modified peptide data representation.

Usage

```
locateMod(string, wrap = "]", inbracket = ")", except = NULL, rmve = NULL)
```

Arguments

`string`	character, length 1. Modified or unmodified peptide, or NULL
`wrap`	character, length 1. The closing (right-hand) side of any of the bracket types ']', ')', '}' that wrap the modifications, such as in protein mass spectrometry data representation of modified peptides. Default, ']'
`inbracket`	character, length 1. Same as above for brackets used inside modification wrappings. Default, ')'
`except`	character, length >= 1. Default, NULL. Punctuation marks or characters that appear along modifications and are needed to remain present in the output: '-', '+', ',', ';', ':', '=', '.', `[:digit:]`, `[:alpha:]`, '\w+', ' '
`rmve`	character, length 1. Default, NULL. Regular expression. Digits or extra characters that need to be removed from the output (see Examples)

Details

Although capable of handling most situations, it is recommended that the `wrapping` bracket type remains consistent throughout and the `inbracket` type *be different* from `wrapping` type. No extra characters are removed from result when `except` = `rmve` = `NULL`.

This utility covers most data representation styles for modified peptide. However, clean data results are not guaranteed. The template for letter casing accepted for modified peptide and for modifications should match those presented in Examples: upper case for peptide and mixed case for modifications.

Value

A 'data.table' class data frame containing the unmodified peptide, the modified peptide, the modification site (i.e. the amino acid code letter and location inside the peptide) and the associated modification(s). In case of monoisotopic mass extraction, monoisotopic mass values populate column "Modification" as "character" types. Multiple modifications (identical or not) found at the same site are listed as many times as they appear at that site. Unmodified, endogenous peptides are listed with no other information. Empty strings are listed as such with a warning.

See Also

regex

Examples

```
if (interactive()) {

# Completely made-up modified peptides:

# 1. Modifications

# 1.1 Default brackets
string = 'K[Prop_A][Met][Prop (C)]PSSABCELR[Prop][Prop][Prop]FQC[Carba (C)]GQQ[Met +44]TARP'

a = locateMod(string)
print(a)                                                    # with extra-characters
b = locateMod(string, except = '\\w+', rmve = '(\\(.*\\)|_[A-Z]|[0-9])')
print(b)                                                  # without extra-characters

# In this example argument "rmve" contains the default in-brackets

# 1.2 Alternative bracketing

string = 'K{Prop_A}{Met}{Prop [A]}PSSABCELR{Prop +15}{Prop}{Prop}FQC{Carba [C]}GQQ{Met +44}TARP'

c = locateMod(string, '}', ']')
print(c)
d = locateMod(string, '}', ']', except = '\\w+', rmve = '(\\[.*\\]|_[A-Z]|[0-9])')
print(d)

# In this example argument "rmve" contains the alternative in-brackets

# 2. Empty string

empty = locateMod(""); print(empty)

# 3. Monoisotopic masses

string = 'TAAC[+57.021464]PPC[+57.021464]PAPPAPS[+162.052824]VFLTLMISR'
e = locateMod(string)
print(e)                                                  # with extra-characters
f = locateMod(string, rmve = '[[:punct:]]')$Modification
print(f)                                                      # incorrect values
g = locateMod(string, rmve = '\\+')$Modification
print(g)                                                      # correct!
class(g)                                                      # character
}
```

oneHot *One-hot Encoder And Decoder Of Variables*

Description

Encodes logical, categorical, integer and double type variables.

Usage

```
oneHot(x, type, omc = "dgCMatrix", verbose = TRUE)
```

Arguments

x	a (named) vector or list for encoding. Missing data are removed. For decoding, a dense or sparse matrix (preferably, the result of encoding) representing a single source data column
type	symbol. Choices: **encode** - one-hot encoding, **decode** - revert to original
omc	character length 1. **O**utput **m**atrix **c**lass. Default, 'dgCMatrix', other option, 'matrix'
verbose	logical, default TRUE, display messages

Details

This utility one-hot encodes when `type = encode` and verifies the encoded result (or any matrix of encodings obtained with getEV extractor) when `type = decode`. It detects illicit states.

Value

Encoding returns a matrix of `length(x)` rows and `length(unique(x))` columns or a warning. Decoding returns a (named) vector or a warning. List vectors are returned unlisted. Integer(ish) vectors, converted to integer, character vectors - to factor, double or logical vector types remain unchanged.

Examples

```
if (interactive()) {

# 1. Encode type "double"

x = runif(9)                          # numeric, length 9
names(x) = letters[1:9]               # named
typeof(x)
a = oneHot(x, encode)                 # a sparse matrix of "dgCMatrix" class
b = oneHot(a, decode)                 # a type "double" named numeric, length 9
isTRUE(all.equal(x, b))               # TRUE
typeof(b)
print(x); print(b)
```

```
# 2. Type "logical" with missing values

y = c(TRUE, TRUE, NA, FALSE, TRUE, NA) # logical, length 6 with missing values
typeof(y)
a = oneHot(y, encode, 'matrix')
print(a)                               # a dense matrix
b = oneHot(a, decode)                  # revert
all.equal(y, b)                        # missing values in y removed
typeof(b)
print(x); print(b)

# 3. iris data

data(iris)
a = lapply(iris, oneHot, encode)       # encode entire data
b = as.data.frame(
         lapply(a, oneHot, decode)     # revert
     )
identical(iris, b)                     # TRUE. Now, replace iris data with
                                       # mtcars data!

# 4. Illicit states in one-hot encoding

`3.41` = c(1,0,0,1,1,0,0,1)            # encoded type "double"
`0.12` = c(0,1,0,0,0,1,1,0)
 a = cbind(`3.41`, `0.12`)             # form encoded matrix
 print(a)                              # matrix resembling one-hot encoding
 x = oneHot(a, decode)                 # illicit state detected
 print(x)                              # list with 2 different data types

}
```

---

score *Scaling And Thresholding Of Numeric Variables*

Description

Implements classical methods for data scaling: range, z-score normalization, location and location-scale normalization, as well as data thesholding through the simplest form of ReLU rectifier. Missing values are removed in all cases.

Usage

```
score(x, how, filter = NULL, ...)
```

Arguments

x	numeric vector, length > 1. Variable to be scaled or filtered
how	symbol. Choices are **range**, **stdev** or **relu**

`filter`	character, length 1. Default NULL. Choices "positive", "negative". Requires `how = relu`
`...`	list reserved for User input of paired values, statistic or otherwise. The list uses individual ellipsis arguments therefore, order of values needs be respected at all times e.g. when `how = range` min value *first*, max value *second*. When `how = stdev`, mean value *first*, standard deviation *second*.

Details

Normalization (scaling) can be applied locally on subsets of `x` when User inputs the values in the `...` list. Otherwise, the scaling is global i.e. it is applied to `x` as a whole. No assumptions regarding the underlying distribution of `x` are made.

`how != relu`. When `...` is empty, the function uses the sample statistics of `x` e.g. the mean, range or standard deviation. Otherwise, it uses values inputted by User case in which, *location-scale* normalization requires `how = stdev` and the `...` list filled as follows: the `min(x)` *or* `max(x)`, or any other value first and `sd(x) > 0` or any other positive value second. In particular, *location* normalization works similarly but with the second value = 1. Other location types, e.g. x/max(x), are obtainable.

NOTE: when `...` is populated with custom values, all other arguments should be present and named (see Examples).

`how = relu`. This option acts as numeric thresholding locally as well as globally. It stands for **re**ctified **l**inear **u**nit and involves no statistics. It applies to numeric types that have ordering property (double, integer). On return, all `x` attributes are dropped. When `filter = 'positive'`, all negative values are set to zero while positive values remain unchanged. Alternatively, when `filter = 'negative'`, all negative values remain unchanged while all other values are set to zero. The "negative" option was added for symmetry.

Value

Numeric. When missing `...` and `how != relu`, scaled values using `x` own sample statistics. Otherwise, scaling is based on values inputted by User. When `how = relu`, `x >= 0` or `x <= 0`, depending on `filter` setting.

References

Ancillary Statistic for location and location-scale distributions

Examples

```
if (interactive()) {

# 1. ReLU thresholding

x = { set.seed(223); sort(runif(10, -3, 3)) }
y = score(x, relu, 'positive'); y
z = score(x, relu, 'negative'); z

# 1.1 ReLU Plot
olp = par(no.readonly = TRUE)
par(list(mar = c(1,1,1,1), mgp = c(0,0,0), tcl = -0.01, pty = 's'))
```

```
plot(x, y, type = 'l', col = 'steelblue', lwd = 2 ,
       xlim = c(min(x), max(x)), ylim = c(min(x), max(x))
     , ylab = expression(ReLU(x)), xaxs = 'i', yaxs = 'i', axes = FALSE, cex.lab = 0.7)
axis(1, pos = 0, cex.axis = 0.6) ; axis(2, pos = 0, cex.axis = 0.6)
points(x, z, type = 'l', col = 'orangered', lwd = 2)
legend('topleft', legend = c('positive', 'negative'),
       col = c('steelblue', 'orangered'), pch = 'l', lwd = 2, cex = 0.6, bty = 'n')
par(olp)

# 2. Location and location-scale

# 2.1 Location (e.g. "x - max(x)")
x = 1:10
M = max(x)
std = 1
a = score(x, stdev, NULL, M, std); a

# 2.2 Location (e.g. "x/max(x)")
m = 0                                                    # the mean
M = max(x)                                               # or any value
b = score(x, range, NULL, m, M); b

# 2.3 Location-scale (e.g. "(x - max(x))/sd(x)")
M = max(x)                                               # or any value
std = sd(x)                                              # or any value > 0
c = score(x, stdev, NULL, M, std); c

# m, M and std above can be replaced with any values decided by User

# 3. Classical normalization

# 3.1 Range
d = score(x, range); d

# 3.2 z-score
e = score(x, stdev); e

# 4. Local vs. global z-score normalization

data(mtcars)
  x = mtcars$wt
  m = mean(x)
std = sd(x)

ll = split(x, f = as.factor(mtcars$cyl))        # partitioned x

# 4.1 Local scaling
aa = lapply(ll, score, stdev, NULL, m, std)     # filled ... list
na = unlist(aa, FALSE, FALSE)

# 4.2 Global scaling
nb = score(x, stdev)
```

```
# 4.3 Local as well as global hold
identical(sort(na), sort(nb))                  # TRUE

}
```

---

splitH *Read Or Write Subsets Of Data Files From Or To Disk*

Description

Reads or writes data files from/to disk in disjoint subsets. This is a two-stage function (see Examples).

Usage

```
splitH(readpath, writepath = NULL)
```

Arguments

readpath	character length 1. Full path to the source file
writepath	character length 1. Full path to the destination file

Details

Above arguments apply to Stage 1 only. The arguments for Stage 2 function, which is the output of Stage 1, are the following:

rows integer, length 1. Number of rows per subset. When rows = Inf, the data can be either copied *as is* or moved to a new location

seq logical, default TRUE: read discrete subsets. Otherwise, progressively appended subsets from first to current

dropcols character of length < ncol(data). Columns to drop. Works only when rows is finite. Replaces argument select from data.table::fread

how symbol. Works only when rows = Inf and writepath location is given. Options: how = scp, data file is copied *as is* to writepath location; how = mv, data file is moved to writepath location

print logical, default TRUE, each subset written to disk is shown in console. Setting print to FALSE could increase writing speed

orn logical, default FALSE. When TRUE, the **o**riginal data **r**ow **n**umbers are shown in each subset

The main purpose of this utility is to bring manageable subsets from very large data into the working environment for further processing when writepath = NULL. When orn = TRUE, each subset receives a new column named "srn" showing **s**ource data **r**ow **n**umbers. This column is absent from subsets written to disk regardless of orn value. The source data file can be any type of file readable by data.table::fread.

At the first stage:

- the utility retrieves information about source data without loading them into memory and also provides the new function which, in the second stage:
- reads source data in successive disjoint subsets (`rows < Inf`) and brings them into the work environment (writepath = NULL), **or**
- writes subsets to `writepath` location appending them automatically to the destination file. During writing, if (`print = TRUE`) the displayed subsets are just printouts (class "NULL"). When `writepath = NULL`, displayed subsets are objects.

There is a functional difference between `rows = Inf` and `rows = nrow(data)`:

- when `rows = Inf`, the size of source data is irrelevant. They can be either copied (`how = scp`) or moved (`how = mv`) to `writepath` destination without being loaded into memory.
- when `rows` has finite value, the size of source data is relevant and data columns can be dropped.

Value

At stage 1, displayed information and a function (a closure). At stage 2, a "data.table" class subset of data or a printout of said subset when written on disk.

References

Part of internal code for Stage 1 was inspired by data.table Issue# 7169

See Also

Linux commands **scp** and **mv**

Examples

```
if (interactive()) {

# Make a 'csv' file

data(mtcars)
tmpf = tempfile(fileext = '.csv')
write.table(mtcars,tmpf , sep = ',', row.names = FALSE, quote = FALSE)

# 1. Read data file step by step

# 1.1 Get information on data
r = splitH(readpath = tmpf)                              # stage 1
class(r)                                                 # function

# 1.2 Read data iteratively                              # stage 2
a = r(rows = 11, dropcols = c('am', 'vs'))               # iter1 no original row numbers
b = r(rows = 11, dropcols = c('am', 'vs'), orn = TRUE)   # iter2 w. original row numbers
c = r(rows = 11, dropcols = c('am', 'vs'))               # iter3 the last subset
## Not run:
d = r(rows = 11, dropcols = c('am', 'vs'))               # iter4 stop! Return to stage 1

## End(Not run)
```

```
print(list(a, b, c))

# 2. Read data file completely

r = splitH(readpath = tmpf)                              # stage 1
n = ceiling(32/13)                                       # read 13 rows at a time
a = replicate(n, r(rows = 13), simplify = FALSE)         # read file
class(a)                                                 # list
print(a)                                                 # a list of tables

tmpf1 = tempfile(fileext = '.csv')                       # new location

# 3. Iteratively write to new location

r = splitH(readpath = tmpf, writepath = tmpf1)           # stage 1
n = ceiling(32/11)                                       # 11 rows each time
invisible(
 replicate(n, r(rows = 11) , simplify = FALSE)           # write to new location
 )
a = data.table::fread(tmpf1)                             # check result
dim(a)
print(head(a))

unlink(tmpf1)

tmpf2 = tempfile(fileext = '.csv')                       # new location

# 4. Move file from tmpf to another location

r = splitH(readpath = tmpf, writepath = tmpf2)           # stage 1
r(rows = Inf, how = mv, print = FALSE)                   # move to new location
a = data.table::fread(tmpf2)                             # check result
print(head(a))

unlink(tmpf)
unlink(tmpf2)

}
```

---

splitV *Split Data Vertically Into Subsets*

Description

Splits data into unequal and disjoint groups of columns (i.e. vertical splits)

Usage

```
splitV(data, splits)
```

Arguments

`data`	a "data.table" class data frame or convertible to "data.table" class
`splits`	integer, length 1, of value <= `ncol(data)`. The number of disjoint selections (i.e. vertical splits). When `splits = 0`, there are no splits

Details

The smaller the `splits` value, the wider the column groups. Column order from source data is not preserved

Value

An exhaustive listenv of disjoint column groups from original data

Examples

```
if (interactive()) {

# 1. Split iris data vertically

data(iris)
a = splitV(iris, splits = 3)     # split data in 3 column groups
class(a)                         # listenv, environment
print(as.list(a))                # list

}
```

`stratify` *Extract A Proportional Stratified Sample From A Data Set*

Description

Obtains a proportional stratified sample from any data convertible to "data.table" class containing categorical variables.

Usage

```
stratify(
  X,
  target,
  stratum = NULL,
  size,
  thresh,
  seed = NULL,
  indx = TRUE,
  dis = NULL,
  args = list(),
  ext = FALSE,
```

```
  replace = FALSE,
  verbose = TRUE
)
```

Arguments

X	any data array convertible to "data.table" class
target	character length 1. The name of column considered to be the root stratum. For example, the name of the 'target' categorical column in a classification training set. This argument should always have a value
stratum	character of length <= ncol(data) - 1. Default, NULL. Names of additional categorical data columns which deepen the stratification
size	integer length 1. Default, none. Value set by User. In this case, it is upper-bounded by the size of the thinnest stratum having more than one row. Setting size value above this bound requires sampling with replacement
thresh	integer, length 1. Default, none. An automatic switch between sample size calculation formulae. Can be set when size is missing from call. It can take as value any of the stratum thicknesses shown in the output message NOTE: it is recommended that *both* size and thresh values are missing from call until information on stratification becomes available after first run
seed	integer length 1. Seed value for output reproducibility
indx	logical. Default TRUE, returns the sample row index only. FALSE, returns the sampled data
dis	symbol. Default NULL. One of the density or function distributions used for generating probability vectors for probabilistic sampling
args	list of arguments required by distributions as described in stats::distributions documentation. Default, none. **NB** The list should *never* include the first argument (**x** or **n**) required in documentation, as it is collected internally from each stratum NOTE: Even if seed is set, the sample row index changes if either the distribution in dis or the values in args is changed
ext	logical, default FALSE. When TRUE, expands the output sampled data with the following extra columns: **row** - sample rows, **strat** - stratum, **n** - stratum total rows (i.e. thickness) and **size** - the sample size extracted from each stratum. Requires indx = FALSE
replace	logical, default FALSE. When TRUE, sampling with replacement if size is present in call and exceeds the thinnest stratum with more than one row
verbose	logical, default TRUE, display messages

Details

This utility is designed to find a true sample representation of the data under current stratification by matching closely the proportionality of strata as long as argument size is missing from call. Each distinct combination of target and stratum levels defines a stratum. For minimal stratification, argument target must always have a value present in call. All one-row strata, when formed, are simply appended to the compounded output.

`size`. As column in the extended output, it represents the size of the sample extracted from each stratum, internally derived to be proportional to stratum thickness, unbounded by the thinnest stratum with more than one row. Deep stratification along with high cardinality and imbalance may severely restrict the size of the compounded output which is the sum of all stratum sizes plus the number of one-row strata. The sampling occurs at stratum level except for one-row strata for which `size = 0` is interpreted as "no sampling".

As function argument, `size` is interpreted as the largest sample size without replacement that can be requested, being bounded by the thinnest stratum with more than one row. The presence of `size` in call alters the proportionality since each stratum - except one-row strata - contributes equally to the output size which is the number of strata times the `size` value plus the number of one-row strata.

`thresh`. Automatic switch that modifies stratum sample size calculation method based on the extreme stratum thickness values, stratification depth and total data rows. Internally, it searches for the formula that finds at least one sample size accommodating the thinnest stratum with more than one row. Messages are displayed at runtime although, in most cases the condition is satisfyed at first iteration. When `thresh >= nrow(data)`, each stratum is sampled proportional with the ratio between thinnest and thickest strata, which may lead to a relatively small size output. All other `thresh` values compromise slightly between output size and proportionality (see Example 3).

Probabilistic Sampling:

`dis`. The `prob` argument in base::sample cannot be used as required since the length of probability vector varies with stratum thickness. Herein, stratum probability vectors are determined by the distribution specified in argument `dis` which associates each stratum with a probability vector of thickness (**n**) length. When `args` is missing from call, `dis` uses the default argument values for respective distribution. An error is thrown when the probability vector has insufficient number of non-zero values. See package **stats**, "Distributions" documentation.

NOTE: The *random variate generators* i.e. the **r*** version of distributions, generate vectors of absolute *random deviate* values which play the role of pseudo-probabilities (i.e. not necessarily elements of (**n-1**)-simplex) conformant with requirements listed in base::sample documentation.

Value

A proportional or non-proportional stratified sample (depending on whether `size` is absent or present in call), either as row index or as sampled data, compounded from random or probability samples taken from each stratum. Informative messages are displayed. Existing data row names are preserved in the output case in which, the sampled data output gains the column named "rn".

See Also

sample, distributions

Examples

```
if (interactive()) {

# 1. Row index for sampling

data(mtcars)
rowID = stratify(mtcars
                 , target = 'cyl'
```

```
               , stratum = c('vs', 'am')
               , seed = 314)                                  # display information
print(rowID)                                                  # integer

# 2. Sampled data with extra-columns

smp = stratify(mtcars
            , 'cyl'
            , c('vs', 'am')
            , seed = 314
            , indx = FALSE                                    # sampled data
            , ext = TRUE)                                     # extra columns
print(smp)
identical(rowID, smp$row)                                     # TRUE

# 3. Impact of "thresh" value on output size

sl = list()
thresholds = c(2, 4, 12, 32)                                  # stratum thicknesses

for (t in seq(along=thresholds)) {
                  sl[[t]] = stratify(mtcars
                                  , 'cyl'
                                  , c('am', 'vs')
                                  , thresh = thresholds[t]
                                  , seed = 314
                                  , indx = FALSE, ext = TRUE)
                }
names(sl) = quote(thresholds)
print(sl)                                                     # stratified samples
                                                              # of various sizes

# 4. Probabilistic sampling

rowIDn = stratify(mtcars
             , 'cyl'
             , c('vs', 'am')
             , seed = 314
             , dis = pnorm                                    # Normal distribution
             , args = c(mean = 1, sd = 3))                    # no first argument!
rowIDb = stratify(mtcars
             , 'cyl'
             , c('vs', 'am')
             , seed = 314                                     # same seed
             , dis = pbeta                                    # Beta distribution
             , args = c(shape1 = 1, shape2 = 3))              # no first argument!

# Same seed but changing the distribution changes the sample row index
identical(rowIDn, rowIDb)                                     # FALSE

}
```

`tileData` *Tile And Write Tiled Data To Disk*

Description

Splits long and wide data files in lists of disjoint tiles for further processing.

Usage

```
tileData(readpath, writepath = NULL, rows, splits, ...)
```

Arguments

`readpath`	character length 1. Full path to the source file
`writepath`	character length 1. Full path to the destination file
`rows`	integer length 1. Number of rows in each subset. Internally, it determines the total number of subsets before the vertical split
`splits`	integer, length 1. Number of vertical data splits in each above subset. See splitV
`...`	extra arguments to splitH e.g. `dropcols` for columns dropped from source data

Details

Facilitates local operations on small size tiles by partitioning the data horizontally and vertically. The list of tiles can be written to disk as "rds" file when a `writepath` destination is given. The written data can then be read entirely or in subsets (see Example 2).

NOTE: This utility uses background processing. Check "Security Considerations" in **callr** package documentation.

Value

A listenv of "data.table" class tiles. When `writepath` is given, it produces a "rds" file containing data tiles.

See Also

splitH, splitV, tileHot, readRDS

Examples

```
if (interactive()) {

# Make a 'csv' file

data(iris)
tmpf = tempfile(fileext = '.csv')
write.table(iris, tmpf , sep = ',', row.names = FALSE, quote = FALSE)
```

```
# 1. Tile data

a = tileData(tmpf, rows = 10, splits = 3)     # 10x2 and 10x1 tiles
class(a)                                      # listenv, environment
str(a)                                        # nested list

tmpf1 = tempfile(fileext = '.rds')            # new location

# 2. Write tiled data

tileData(tmpf, tmpf1, rows = 10, splits = 3)
a = readRDS(tmpf1)[[1]]                 # partial read from new location
print(a)                                    # list component

unlink(tmpf)
unlink(tmpf1)
}
```

`tileHot` *One-hot Encoder Of Tiled Data*

Description

One-hot encodes tiled data.

Usage

```
tileHot(readpath, rows, splits, omc = "dgCMatrix", ...)
```

Arguments

`readpath`	character, length 1. Path to source data that is readable with data.table::fread
`rows`	integer length 1. Number of rows in each data subset. Internally, it determines the total number of subsets before the vertical split
`splits`	integer, length 1. Number of vertical data splits in each subset, see splitV. Recommended for very wide data frames. When `splits = 0`, no vertical splitting occurs
`omc`	character length 1. **O**utput **m**atrix **c**lass. Default, "dgCMatrix". Other option: "matrix"
`...`	reserved for splitH function arguments, such as `dropcols` or `orn = TRUE` which is needed for single matrix output

Details

This utility reads the data in disjoint subsets, tiles them and then one-hot encodes each tile. Encoded tiles are returned as nested list of matrices, as a single matrix, as data frame or as a two-component data frame and sparse matrix list, decided through combinations of `dropcols`, `omc` and `orn` values.

NOTE 1: traceability is assured by assembling the data as character names and values from columns marked for encoding. As side effect, at run time the encoding is reported as being applied to "integer(ish)" values only with no loss in accuracy. Empty source data columns gain the "NA" suffix and become single-column, single-valued matrices.

NOTE 2: this utility implements background, processing. Check "Security Considerations" in **callr** package documentation.

Value

- When `orn = FALSE`, an unnamed listenv of sparse matrices. Recommended for very large source data files. Before proceeding with list output, read NOTE 2 in getEV documentation. See Examples 1 and 2.
- When `orn = TRUE`, a matrix.

NOTE 3: In this case, row and column binding operations were avoided to prevent situations described in NOTE 2, getEV documentation. As result, the output matrix is gradually populated instead of being gradually expanded.

While `orn = TRUE` and `dropcols != NULL`:

- When `omc = 'matrix'`, a data.table containing encoded, as well as unencoded, dropped columns placed in the leftmost positions.
- When `omc = 'dgCMatrix'`, a two-component listenv: a data table containing dropped, unencoded columns and a sparse matrix containing the encoded columns. The row order in both components is identical. See Examples below, and Example 2 in getEV documentation.

NOTE 4: In all above cases, specific encoded variables can be obtained with getEV extractor. When `orn = TRUE`, oneHot *decoded* variables extracted from matrix outputs return named vectors having row numbers as names.

See Also

splitH, splitV, oneHot, listenv, Matrix

Examples

```
if (interactive()) {

# 1. Shuffled data

tempf = tempfile(fileext = '.csv')
data(iris)
iris22 = iris[{ set.seed(327); sample.int(150) },]          # shuffled iris data
rownames(iris22) <- NULL                                    # remove shuffled row names
write.table(iris22, tempf, sep = ',', row.names = FALSE, quote = FALSE)
```

```
# 1.1 Output as List
# In most cases, list output requires shuffled data!

A = tileHot(readpath = tempf
          , rows = 14, splits = 3, print = FALSE)           # encoded data tiles
print(A)                                                    # a listenv
print(A[[1]])                                               # a snapshot

# 1.2 Retrieve iris22 data from encoded list output
X = sapply(names(iris22), \(n) getEV(A, n))                # extract all encoded columns
Y = lapply(
         lapply(X, oneHot, decode)
                          , unname)                 # decoded columns are named vectors!
d = as.data.frame(Y)
identical(iris22, d)                                        # TRUE

unlink(tempf)

# 2. Unshuffled data

# Make unshuffled data 'csv' file
tempf = tempfile(fileext = '.csv')
write.table(iris, tempf, sep = ',', row.names = FALSE, quote = FALSE)

# 2.1 Output as list
# List output fails low cardinality variables on unshuffled data.

E = tileHot(readpath = tempf
              , rows = 14, splits = 3, print = FALSE)      # same as above

# 2.2 Retrieve iris data from encoded list output
V = sapply(names(iris), \(n) getEV(E, n))                  # warning
W = lapply(
         lapply(V, oneHot, decode)
                          , unname)                 # decoded columns are named vectors!
dd = as.data.frame(W)
identical(iris, dd)                                        # FALSE
all.equal(iris, dd)                                        # low cardinality "Species"

# 2.3 Output as matrix
# Matrix output handles low cardinality variables. No data shuffling required.

m = tileHot(readpath = tempf                               # low cardinality "Species"
          , rows = 14
          , splits = 3
          , orn = TRUE,                                    # needed for matrix output
          , print = FALSE)
print(m)                                                   # 150x126 sparse matrix

# 2.4 Retrieve iris data from encoded matrix output
P = sapply(names(iris), \(n) getEV(m, n))                  # extract encoded columns
Q = lapply(
           lapply(P, oneHot, decode)
```

```
                          , unname)                  # decoded columns are named vectors!
R = as.data.frame(Q)
identical(iris, R)                                   # TRUE

# 2.5 Output as "data.table" class
D = tileHot(readpath = tempf
          , rows = 14
          , splits = 3
          , omc = 'matrix'                           # encoded dense matrix
          , dropcols = c('Petal.Width', 'Petal.Length')    # unencoded columns
          , orn = TRUE                               # needed for matrix output
          , print = FALSE)
print(head(D, 10))                                   # a "data.table" class
dim(D)                                               # 150x63

# 2.6 Output as a 2-component list
Dl = tileHot(readpath = tempf
          , rows = 14
          , splits = 3
          , omc = 'dgCMatrix'                        # the default class
          , dropcols = c('Petal.Width', 'Petal.Length')    # unencoded columns
          , orn = TRUE                               # needed for matrix output
          , print = FALSE)
print(Dl)                                            # 2-component listenv
print(Dl[[1]])                                       # unencoded columns
print(Dl[[2]])                                       # encoded sparse matrix

# iris data can be retrieved from the Dl list in similar fashion described above

unlink(tempf)

}
```

Index