

Package: popEpi (via r-universe)

October 21, 2026

Title Functions for Epidemiological Analysis using Population Data

Version 0.5.0.9028

Description Enables computation of epidemiological statistics, including those where counts or mortality rates of the reference population are used. Currently supported: excess hazard models (Dickman, Sloggett, Hills, and Hakulinen (2012) <[doi:10.1002/sim.1597](https://doi.org/10.1002/sim.1597)>), rates, mean survival times, relative/net survival (in particular the Ederer II (Ederer and Heise (1959)) and Pohar Perme (Pohar Perme, Stare, and Esteve (2012) <[doi:10.1111/j.1541-0420.2011.01640.x](https://doi.org/10.1111/j.1541-0420.2011.01640.x)>) estimators), and standardized incidence and mortality ratios, all of which can be easily adjusted for by covariates such as age. Fast splitting and aggregation of 'Lexis' objects (from package 'Epi') and other computations achieved using 'data.table'.

License MIT + file LICENSE

URL <https://github.com/FinnishCancerRegistry/popEpi>

BugReports <https://github.com/FinnishCancerRegistry/popEpi/issues>

Depends R (>= 3.5)

Imports data.table (>= 1.10.4), directadjusting (>= 0.6.0), Epi (>= 2.0), methods, splines, survival

Suggests knitr, mstate, relsurv, rmarkdown, roxygen2, testthat

VignetteBuilder knitr

ByteCompile true

Config/roxygen2/version 8.1.0

Encoding UTF-8

Language en-GB

LazyData true

Roxygen list(markdown = TRUE)

Repository <https://finnishcancerregistry.r-universe.dev>

Date/Publication 2026-10-21 07:40:39 UTC

RemoteUrl <https://github.com/finnishcancerregistry/popepi>

RemoteRef HEAD

RemoteSha 50ee3e41dbca281c1def749cfabbf649aa129adf

Contents

adjust	3
aggre	4
all_names_present	7
array_df_ratetable_utils	8
as.aggre	10
as.Date.yrs	12
cast_simple	13
cut_bound	14
direct_standardization	15
fac2num	17
flexible_argument	18
get.yrs	21
ICSS	22
is.Date	23
is_leap_year	24
lexis_funs	25
lexis_merge	28
lexis_split_merge_aggregate_by_stratum	33
lexpand	45
lines.sirspline	51
lines.survmean	52
lines.survtab	53
lower_bound	54
ltable	55
meanpop_fi	58
na2zero	58
plot.rate	59
plot.sir	60
plot.sirspline	61
plot.survmean	62
plot.survtab	63
poisson.ci	65
pophaz	66
popmort	67
prepExpo	67
prev_functions	69
print.aggre	80
print.rate	81
print.survtab	82
rate	82
rate_estimate	84

rate_ratio	88
relpois	90
relpois_ag	92
robust_values	94
rpcurve	96
RPL	97
setaggre	98
setclass	99
setcolnull	100
sibr	101
sir	102
sir_exp	105
sir_ratio	107
sire	109
sirspline	110
splitLexisDT	112
splitMulti	114
stdpop101	117
stdpop18	117
summary.aggre	118
summary.survtab	119
Surv	121
surv_estimate	122
survmean	151
survtab	155
survtab_ag	160
try2int	167
Index	168

adjust	<i>Adjust Estimates by Categorical Variables</i>
--------	--

Description

This function is only intended to be used within a formula when supplied to e.g. `[survtab_ag]` and should not be used elsewhere.

Usage

```
adjust(...)
```

Arguments

... variables to adjust by, e.g. `adjust(factor(v1), v2, v3)`

Value

Returns a list of promises of the variables supplied which can be evaluated.

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)

y ~ x + adjust(z)
```

aggre

Aggregation of split Lexis data

Description

Aggregates a split Lexis object by given variables and / or expressions into a long-format table of person-years and transitions / end-points. Automatic aggregation over time scales by which data has been split if the respective time scales are mentioned in the aggregation argument to e.g. intervals of calendar time, follow-up time and/or age.

Usage

```
aggre(
  lex,
  by = NULL,
  type = c("unique", "full"),
  sum.values = NULL,
  subset = NULL,
  verbose = FALSE
)
```

Arguments

lex	a Lexis object split with e.g. <code>[Epi::splitLexis]</code> or <code>[splitMulti]</code>
by	variables to tabulate (aggregate) by. Flexible input , typically e.g. <code>by = c("V1", "V2")</code> . See Details and Examples.
type	determines output levels to which data is aggregated varying from returning only rows with <code>pyrs > 0</code> ("unique") to returning all possible combinations of variables given in <code>aggre</code> even if those combinations are not represented in data ("full"); see Details
sum.values	optional: additional variables to sum by argument by. Flexible input , typically e.g. <code>sum.values = c("V1", "V2")</code>
subset	a logical condition to subset by before computations; e.g. <code>subset = area %in% c("A", "B")</code>
verbose	logical; if TRUE, the function returns timings and some information useful for debugging along the aggregation process

Details

Basics

aggre is intended for aggregation of split Lexis data only. See `[Epi::Lexis]` for forming Lexis objects by hand and e.g. `[Epi::splitLexis]`, `[splitLexisDT]`, and `[splitMulti]` for splitting the data. `[lexexpand]` may be used for simple data sets to do both steps as well as aggregation in the same function call.

Here aggregation refers to computing person-years and the appropriate events (state transitions and end points in status) for the subjects in the data. Hence, it computes e.g. deaths (end-point and state transition) and censorings (end-point) as well as events in a multi-state setting (state transitions).

The result is a long-format data.frame or data.table (depending on `options("popEpi.datatable")`; see `?popEpi`) with the columns `pyrs` and the appropriate transitions named as `fromXtoY`, e.g. `from0to0` and `from0to1` depending on the values of `lex.Cst` and `lex.Xst`.

The by argument

The `by` argument determines the length of the table, i.e. the combinations of variables to which data is aggregated. `by` is relatively flexible, as it can be supplied as

- a character string vector, e.g. `c("sex", "area")`, naming variables existing in `lex`
- an expression, e.g. `factor(sex, 0:1, c("m", "f"))` using any variable found in `lex`
- a list (fully or partially named) of expressions, e.g. `list(gender = factor(sex, 0:1, c("m", "f")), area)`

Note that expressions effectively allow a variable to be supplied simply as e.g. `by = sex` (as a symbol/name in R lingo).

The data is then aggregated to the levels of the given variables or expression(s). Variables defined to be time scales in the supplied Lexis are processed in a special way: If any are mentioned in the `by` argument, intervals of them are formed based on the breaks used to split the data: e.g. if age was split using the breaks `c(0, 50, Inf)`, mentioning age in `by` leads to creating the age intervals `[0, 50)` and `[50, Inf)` and aggregating to them. The intervals are identified in the output as the lower bounds of the appropriate intervals.

The order of multiple time scales mentioned in `by` matters, as the last mentioned time scale is assumed to be a survival time scale for when computing event counts. E.g. when the data is split by the breaks `list(FUT = 0:5, CAL = c(2008, 2010))`, time lines cut short at `CAL = 2010` are considered to be censored, but time lines cut short at `FUT = 5` are not. See `Return`.

Aggregation types (styles)

It is almost always enough to aggregate the data to variable levels that are actually represented in the data (default `aggre = "unique"`; alias `"non-empty"`). For certain uses it may be useful to have also "empty" levels represented (resulting in some rows in output with zero person-years and events); in these cases supplying `aggre = "full"` (alias `"cartesian"`) causes `aggre` to determine the Cartesian product of all the levels of the supplied by variables or expressions and aggregate to them. As an example of a Cartesian product, try

```
merge(1:2, 1:5).
```

Value

A long data.frame or data.table of aggregated person-years (`pyrs`), numbers of subjects at risk (`at.risk`), and events formatted `fromXtoY`, where `X` and `Y` are states transitioning from and to or

states at the end of each `lex.id`'s follow-up (implying $X = Y$). Subjects at risk are computed in the beginning of an interval defined by any Lexis time scales and mentioned in `by`, but events occur at any point within an interval.

When the data has been split along multiple time scales, the last time scale mentioned in `by` is considered to be the survival time scale with regard to computing events. Time lines cut short by the extrema of non-survival-time-scales are considered to be censored ("transitions" from the current state to the current state).

Author(s)

Joonas Miettinen

See Also

[`aggregate`] for a similar base R solution, and [`ltable`] for a `data.table` based aggregator. Neither are directly applicable to split Lexis data.

Other aggregation functions: [as.aggre\(\)](#), [lexexpand\(\)](#), [setaggre\(\)](#), [summary.aggre\(\)](#)

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)

## form a Lexis object
library(Epi)
data(sibr)
x <- sibr[1:10,]
x[1:5,]$sex <- 0 ## pretend some are male
x <- Lexis(data = x,
           entry = list(AGE = dg_age, CAL = get.yrs(dg_date)),
           exit = list(CAL = get.yrs(ex_date)),
           entry.status=0, exit.status = status)
x <- splitMulti(x, breaks = list(CAL = seq(1993, 2013, 5),
                                AGE = seq(0, 100, 50)))

## these produce the same results (with differing ways of determining aggre)
a1 <- aggre(x, by = list(gender = factor(sex, 0:1, c("m", "f")),
                        agegroup = AGE, period = CAL))

a2 <- aggre(x, by = c("sex", "AGE", "CAL"))

a3 <- aggre(x, by = list(sex, agegroup = AGE, CAL))

## returning also empty levels
a4 <- aggre(x, by = c("sex", "AGE", "CAL"), type = "full")

## computing also expected numbers of cases
x <- lexexpand(sibr[1:10,], birth = bi_date, entry = dg_date,
```

```

        exit = ex_date, status = status %in% 1:2,
        pophaz = popmort, fot = 0:5, age = c(0, 50, 100))
x$d.exp <- with(x, lex.dur*pop.haz)
## these produce the same result
a5 <- aggre(x, by = c("sex", "age", "fot"), sum.values = list(d.exp))
a5 <- aggre(x, by = c("sex", "age", "fot"), sum.values = "d.exp")
a5 <- aggre(x, by = c("sex", "age", "fot"), sum.values = d.exp)
## same result here with custom name
a5 <- aggre(x, by = c("sex", "age", "fot"),
            sum.values = list(expCases = d.exp))

## computing pohar-perme weighted figures
x$d.exp.pp <- with(x, lex.dur*pop.haz*pp)
a6 <- aggre(x, by = c("sex", "age", "fot"),
            sum.values = c("d.exp", "d.exp.pp"))
## or equivalently e.g. sum.values = list(expCases = d.exp, expCases.p = d.exp.pp).

```

all_names_present	<i>Check if all names are present in given data</i>
-------------------	---

Description

Given a character vector, checks if all names are present in names(data). Throws error if stops=TRUE, else returns FALSE if some variable name is not present.

Usage

```
all_names_present(data, var.names, stops = TRUE, msg = NULL)
```

Arguments

data	dataset where the variable names should be found
var.names	a character vector of variable names, e.g. c("var1", "var2")
stops	logical, stop returns exception
msg	Custom message to return instead of default message. Special: include %%VARS%% in message string and the missing variable names will be inserted there (quoted, separated by comma, e.g. 'var1 ', 'var2' — no leading or trailing white space).

Value

TRUE if all var.names are in data, else FALSE,

Author(s)

Joonas Miettinen

See Also

[robust_values]

array_df_ratetable_utils
arrays, data.frames *and* ratetables

Description

Utilities to transform objects between array, data.frame, and [survival::ratetable](#).

Usage

```
long_df_to_array(x, stratum.col.nms, value.col.nm)
```

```
long_df_to_ratetable(  
  x,  
  stratum.col.nms,  
  value.col.nm,  
  dim.types,  
  cut.points = NULL  
)
```

```
long_dt_to_array(x, stratum.col.nms, value.col.nm)
```

```
long_dt_to_ratetable(  
  x,  
  stratum.col.nms,  
  value.col.nm,  
  dim.types,  
  cut.points = NULL  
)
```

```
array_to_long_df(x)
```

```
array_to_long_dt(x)
```

```
array_to_ratetable(x, dim.types, cut.points = NULL)
```

```
ratetable_to_array(x)
```

```
ratetable_to_long_df(x)
```

```
ratetable_to_long_dt(x)
```

Arguments

x [data.frame, data.table, array, ratetable] (mandatory, no default)

- long_df_to_array: a data.frame
- long_df_to_ratetable: a data.frame

	• long_dt_to_array: a <code>data.table</code> • long_dt_to_ratetable: a <code>data.table</code> • array_to_long_df: an array • array_to_long_dt: an array • array_to_ratetable: an array • ratetable_to_array: a survival::ratetable • ratetable_to_long_df: a survival::ratetable • ratetable_to_long_dt: a survival::ratetable
stratum.col.nms	[character] (mandatory, no default) a vector of column names in x by which values are stratified
value.col.nm	[character] (mandatory, no default) name of column in x containing values (these will be contents of the array)
dim.types	[integer] (mandatory, no default) see type under Details in survival::ratetable
cut.points	[NULL, list] (optional, default NULL) see cutpoints under Details in survival::ratetable • NULL: automatically set using <code>dimnames(x)</code> and <code>dim.types</code> • list: one element for each dimensions of x

Details

- long_df_to_array: converts a long-format `data.frame` to an array with one or more dimensions
- long_df_to_ratetable: calls `long_df_to_array` and then `array_to_ratetable`
- long_dt_to_array: simply asserts that x is a `data.table` and calls `long_df_to_array`
- long_dt_to_ratetable: calls `long_dt_to_array` and then `array_to_ratetable`
- array_to_long_df: converts an array with one or more dimensions into a long-format `data.frame`; any [dimnames](#) are used to name and fill the stratifying columns; for dimensions without a name, ".dX" is used for stratifying column number X; for each k, if there are no contents in `dimnames(x)[[k]]`, the elements of `seq(dim(x)[k])` are used to fill the corresponding stratifying column; the value column always has the name "value"
- array_to_long_dt: calls `array_to_long_df` and converts result to a `data.table` for convenience
- array_to_ratetable: converts an array to a [survival::ratetable](#)
- ratetable_to_array: converts a [survival::ratetable](#) to an array
- ratetable_to_long_df: calls `ratetable_to_array` and then `array_to_long_df`
- ratetable_to_long_dt: calls `ratetable_to_array` and then `array_to_long_dt`

Value

- long_df_to_array: an array
- long_df_to_ratetable: a [survival::ratetable](#)
- long_dt_to_array: an array
- long_dt_to_ratetable: a [survival::ratetable](#)
- array_to_long_df: an data.frame
- array_to_long_dt: an data.table
- array_to_ratetable: a [survival::ratetable](#)
- ratetable_to_array: an array
- ratetable_to_long_df: a data.frame
- ratetable_to_long_dt: a data.table

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)

long_dt <- popEpi::popmort
arr <- long_df_to_array(long_dt, c("agegroup", "year", "sex"), "haz")
rt <- array_to_ratetable(arr, dim.types = c(2L, 4L, 1L))

arr2 <- ratetable_to_array(rt)
long_df2 <- array_to_long_df(arr2)

identical(sort(long_dt[["haz"]]), sort(long_df2[["value"]]))
```

as.aggre

Coercion to Class aggre**Description**

Coerces an R object to an aggre object, identifying the object as one containing aggregated counts, person-years and other information.

Usage

```
as.aggre(x, values = NULL, by = NULL, breaks = NULL, ...)

## S3 method for class 'data.frame'
as.aggre(x, values = NULL, by = NULL, breaks = NULL, ...)

## S3 method for class 'data.table'
```

```
as.aggre(x, values = NULL, by = NULL, breaks = NULL, ...)

## Default S3 method:
as.aggre(x, ...)
```

Arguments

x	a data.frame or data.table
values	a character string vector; the names of value variables
by	a character string vector; the names of variables by which values have been tabulated
breaks	a list of breaks, where each element is a breaks vector as usually passed to e.g. [splitLexisDT]. The list must be fully named, with the names corresponding to time scales at the aggregate level in your data. Every unique value in a time scale variable in data must also exist in the corresponding vector in the breaks list.
...	arguments passed to or from methods

Value

Returns a copy of x with attributes set to those of an object of class "aggre".

Methods (by class)

- as.aggre(data.frame): Coerces a data.frame to an aggre object
- as.aggre(data.table): Coerces a data.table to an aggre object
- as.aggre(default): Default method for as.aggre (stops computations if no class-specific method found)

Author(s)

Joonas Miettinen

See Also

Other aggregation functions: [aggre\(\)](#), [lexexpand\(\)](#), [setaggre\(\)](#), [summary.aggre\(\)](#)

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)
library("data.table")
df <- data.frame(sex = rep(c("male", "female"), each = 5),
                 obs = rpois(10, rep(7,5, each=5)),
                 pyrs = rpois(10, lambda = 10000))
dt <- as.data.table(df)
```

```

df <- as.aggre(df, values = c("pyrs", "obs"), by = "sex")
dt <- as.aggre(dt, values = c("pyrs", "obs"), by = "sex")

class(df)
class(dt)

BL <- list(fot = 0:5)
df <- data.frame(df)
df <- as.aggre(df, values = c("pyrs", "obs"), by = "sex", breaks = BL)

```

as.Date.yrs

Coerce Fractional Year Values to Date Values

Description

Coerces an yrs object to a Date object. Some loss of information comes if `year.length = "approx"` was set when using `[get.yrs]`, so the transformation back to Date will not be perfect there. With `year.length = "actual"` the original values are perfectly retrieved.

Usage

```

## S3 method for class 'yrs'
as.Date(x, ...)

```

Arguments

<code>x</code>	an yrs object created by <code>get.yrs</code>
<code>...</code>	unused, included for compatibility with other <code>as.Date</code> methods

Value

A vector of Date values based on the input fractional years.

Author(s)

Joonas Miettinen

See Also

`[get.yrs]`

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)
data("sire", package = "popEpi")

## approximate year lengths: here 20 % have an extra day added
sire$dg_yrs <- get.yrs(sire$dg_date)
summary(sire$dg_yrs)
dg_date2 <- as.Date(sire$dg_yrs)
summary(as.numeric(dg_date2 - as.Date(sire$dg_date)))

## using actual year lengths
sire$dg_yrs <- get.yrs(sire$dg_date, year.length = "actual")
summary(sire$dg_yrs)
dg_date2 <- as.Date(sire$dg_yrs)
summary(as.numeric(dg_date2 - as.Date(sire$dg_date)))
```

cast_simple

Cast data.table/data.frame from long format to wide format

Description

Convenience function for using `[data.table::dcast.data.table]`; inputs are character strings (names of variables) instead of a formula.

Usage

```
cast_simple(data = NULL, columns = NULL, rows = NULL, values = NULL)
```

Arguments

data	a data.table or data.frame
columns	a character string vector; the (unique combinations of the) levels of these variable will be different rows
rows	a character string vector; the (unique combinations of the) levels of these variable will be different columns
values	a character string; the variable which will be represented on rows and columns as specified by columns and rows

Details

This function is just a small interface for `dcast / dcast.data.table` and less flexible than the originals.

Note that all data.table objects are also data.frame objects, but that each have their own dcast method. `[data.table::dcast.data.table]` is faster.

If any values in `value.vars` need to be aggregated, they are aggregated using `sum`. See `?dcast`.

Value

A data.table just like [data.table::dcast].

Author(s)

Matti Rantanen, Joonas Miettinen

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)
library("data.table")
## e.g. silly counts from a long-format table to a wide format
test <- data.table::copy(popEpi::sire)
test$dg_y <- year(test$dg_date)
test$ex_y <- year(test$ex_date)
tab <- ltable(test, c("dg_y", "ex_y"))
cast_simple(tab, columns='dg_y', rows="ex_y", values="obs")
```

cut_bound

Change output values from cut(..., labels = NULL) output

Description

Selects lowest values of each factor after cut() based on the assumption that the value starts from index 2 and end in comma ",".

Usage

```
cut_bound(t, factor = TRUE)
```

Arguments

t is a character vector of elements, e.g. "(20,60)"
 factor logical; TRUE returns informative character string, FALSE numeric (left value)

Details

type = 'factor': "[50,52)" -> "50-51" OR "[50,51)" -> "50"
 type = 'numeric': lowest bound in numeric.

Value

If factor = TRUE, returns a character vector; else returns a numeric vector.

Author(s)

Matti Rantanen

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)
cut_bound("[1900, 1910)") ## "1900-1909"
```

direct_standardization

*Direct Adjusting in **popEpi** Using Weights*

Description

Several functions in **popEpi** have support for direct standardization of estimates. This document explains the usage of weighting with those functions.

Details

Direct standardization is performed by computing estimates of E by the set of adjusting variables A, to which a set of weights W is applicable. The weighted average over A is then the direct-adjusted estimate of E (E^*).

To enable both quick and easy as well as more rigorous usage of direct standardization with weights, the weights arguments in **popEpi** can be supplied in several ways. Ability to use the different ways depends on the number of adjusting variables.

The weights are always handled internally to sum to 1, so they do not need to be scaled in this manner when they are supplied. E.g. counts of subjects in strata may be passed.

Basic usage - one adjusting variable

In the simple case where we are adjusting by only one variable (e.g. by age group), one can simply supply a vector of weights:

```
FUN(weights = c(0.1, 0.25, 0.25, 0.2, 0.2))
```

which may be stored in advance:

```
w <- c(0.1, 0.25, 0.25, 0.2, 0.2)
```

```
FUN(weights = w)
```

The order of the weights matters. **popEpi** functions with direct adjusting enabled match the supplied weights to the adjusting variables as follows: If the adjusting variable is a factor, the order of the levels is used. Otherwise, the alphabetic order of the unique values is used (try sort

to see how it works). For clarity and certainty we recommend using factor or numeric variables when possible. character variables should be avoided: to see why, try `sort(15:9)` and `sort(as.character(15:9))`.

It is also possible to supply a character string corresponding to one of the age group standardization schemes integrated into **popEpi**:

- 'europe_1976_18of5' - European std. population (1976), 18 age groups
- 'nordic_2000_18of5' - Nordic std. population (2000), 18 age groups
- 'world_1966_18of5' - world standard (1966), 18 age groups
- 'world_2000_18of5' - world standard (2000), 18 age groups
- 'world_2000_20of5' - world standard (2000), 20 age groups
- 'world_2000_101of1' - world standard (2000), 101 age groups

Additionally, [ICSS] contains international weights used in cancer survival analysis, but they are not currently usable by passing a string to `weights` and must be supplied by hand.

You may also supply `weights = "internal"` to use internally computed weights, i.e. usually simply the counts of subjects / person-time experienced in each stratum. E.g.

```
FUN(weights = "world_2000_18of5")
```

will use the world standard population from 2000 as weights for 18 age groups, that your adjusting variable is assumed to contain. The adjusting variable must be coded in this case as a numeric variable containing 1:18 or as a factor with 18 levels (coded from the youngest to the oldest age group).

More than one adjusting variable

In the case that you employ more than one adjusting variable, separate weights should be passed to match to the levels of the different adjusting variables. When supplied correctly, "grand" weights are formed based on the variable-specific weights by multiplying over the variable-specific weights (e.g. if men have $w = 0.5$ and the age group 0-4 has $w = 0.1$, the "grand" weight for men aged 0-4 is 0.5×0.1). The "grand" weights are then used for adjusting after ensuring they sum to one.

When using multiple adjusting variables, you are allowed to pass either a named list of weights or a data.frame of weights. E.g.

```
WL <- list(agegroup = age_w, sex = sex_w)
```

```
FUN(weights = WL)
```

where `age_w` and `sex_w` are numeric vectors. Given the conditions explained in the previous section are satisfied, you may also do e.g.

```
WL <- list(agegroup = "world_2000_18of", sex = sex_w)
```

```
FUN(weights = WL)
```

and the world standard pop is used as weights for the age groups as outlined in the previous section.

Sometimes using a data.frame can be clearer (and it is fool-proof as well). To do this, form a data.frame that repeats the levels of your adjusting variables by each level of every other adjusting variable, and assign the weights as a column named "weights". E.g.

```
wdf <- data.frame(sex = rep(0:1, each = 18), agegroup = rep(1:18, 2))
```

```
wdf$weights <- rbinom(36, size = 100, prob = 0.25)
```

```
FUN(weights = wdf)
```

If you want to use the counts of subjects in strata as the weights, one way to do this is by e.g.

```
wdf <- as.data.frame(x$V1, x$V2, x$V3) names(wdf) <- c("V1", "V2", "V3", "weights")
```

Author(s)

Joonas Miettinen

References

Source of the Nordic standard population in 5-year age groups (also contains European & 1966 world standards): <https://www-dep.iarc.fr/NORDCAN/english/glossary.htm>

Source of the 1976 European standard population:

Waterhouse, J., Muir, C.S., Correa, P., Powell, J., eds (1976). Cancer Incidence in Five Continents, Vol. III. IARC Scientific Publications, No. 15, Lyon, IARC. ISBN: 9789283211150

Source of 2000 world standard population in 1-year age groups: <https://seer.cancer.gov/stdpopulations/stdpop.singleages.html>

See Also

Other weights: [ICSS](#), [stdpop101](#), [stdpop18](#)

Other popEpi argument evaluation docs: [flexible_argument](#)

fac2num

Convert factor variable to numeric

Description

Convert factor variable with numbers as levels into a numeric variable

Usage

```
fac2num(x)
```

Arguments

x a factor variable with numbers as levels

Details

For example, a factor with levels `c("5", "7")` is converted into a numeric variable with values `c(5, 7)`.

Value

A numeric vector based on the levels of x.

Source

[Stackoverflow thread](#)

See Also

[robust_values]

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)
## this is often not intended
as.numeric(factor(c(5,7))) ## result: c(1,2)
## but this
fac2num(factor(c(5,7))) ## result: c(5,7)

## however
as.numeric(factor(c("5","7","a")))) ## 1:3

suppressWarnings(
  fac2num(factor(c("5","7","a")))) ## c(5,7,NA)
)
```

flexible_argument

*Flexible Variable Usage in **popEpi** Functions*

Description

Certain arguments in **popEpi** can be passed in multiple ways. This document shows the usage and a pitfall in the usage of such flexible arguments.

Details

Flexible arguments in **popEpi** are used to pass variables existing in your data or in the environment where the function is used (for everyday users this is the global environment - in simple terms, where your data is / your work space). The flexible arguments are modelled after the `by` argument in `data.tables` - see `?data.table`. There are many ways to supply the same information to certain functions in **popEpi**, but the possible ways listed below may be limited in some of them to only allow for using only a part of them.

Everyday usage

Most commonly you may pass variable names as character strings, e.g.

```
FUN(arg = c("V1", "V2"), data = x)
```

which may be stored in advance:

```
vars <- c("V1", "V2")
```

```
FUN(arg = vars, data = x)
```

where `x` contains those variables. You may also supply variable names as symbols:

```
FUN(arg = V1, data = x)
```

Or as a list of symbols (similarly to as in `[aggregate]`):

```
FUN(arg = list(V1, V2), data = x)
```

Or as a list of expressions:

```
FUN(arg = list(V1 + 1, factor(V2)), data = x)
```

A formula without a left-hand-side specified is sometimes allowed as well:

```
FUN(arg = ~ I(V1 + 1) + factor(V2), data = x)
```

Using a symbol or a list of symbols/expressions typically causes the function to look for the variable(s) first in the supplied data (if any) and then where the function was called. For everyday users this means you might define e.g.

```
V3 <- factor(letters)
```

and do e.g.

```
FUN(arg = list(V1 + 1, factor(V2), V3), data = x)
```

provided `V1` and `V2` exist in `x` or in the function calling environment.

A pitfall

There is one way to use flexible arguments incorrectly: By supplying the name of a variable which exists both in the supplied data and the calling environment, and intending the latter to be used. E.g.

```
vars <- c("V2")
```

```
FUN(arg = V3, data = x)
```

where `x` has a column named `vars`. This causes the function to use `x$vars` and NOT `x$V2`.

Advanced

Function programmers are advised to pass character strings whenever possible. To fool-proof against conflicts as described in the section above, refer to the calling environment explicitly when passing the variable containing the character strings:

```
TF <- environment() ## current env to refer to
```

```
vars <- c("V1", "V2")
```

```
FUN(arg = TF$vars, data = x)
```

Even if `x` has columns named `vars` and `TF`, using `TF$vars` does not use those columns but only evaluates `TF$vars` in the calling environment. This is made possible by the fact that data is always passed as a `data.frame`, within which evaluation of expressions using the dollar operator is not

possible. Therefore it is safe to assume the data should not be used. However, lists of expressions will not be checked for dollar use and will fail in conflict situations:

```
TF <- environment() ## current env to refer to
vars <- letters[1:5]
x <- data.frame(vars = 1:5, TF = 5:1, V1 = 10:6)
FUN(arg = list(TF$vars, V1), data = x)
```

On the other hand you may typically also pass quoted ([quote]) or substituted [substitute] expressions etc., where the env\$object trick will work as well:

```
q <- quote(list(vars, V1))
FUN(arg = TF$q, data = x)
```

This works even with

```
a <- 1:5
V1 <- quote(TF$a)
FUN(arg = TF$V1, data = x)
```

So no conflicts should occur.

Author(s)

Joonas Miettinen

See Also

Other popEpi argument evaluation docs: [direct_standardization](#)

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)

data(sire)
## prepare data for e.g. 5-year "period analysis" for 2008-2012
## note: sire is a simulated cohort integrated into popEpi.
BL <- list(fot=seq(0, 5, by = 1/12))
x <- lexpand(sire, birth = bi_date, entry = dg_date, exit = ex_date,
             status = status %in% 1:2,
             breaks = BL)

x <- aggreg(x, by = fot)

## silly example of referring to pyrs data by fixed character string;
## its possible that the real name wont be fixed in a real-life application.
pyrs <- "actual_pyr"
TF <- environment()
x$actual_pyr <- as.numeric(x$pyrs)
```

```

x$pyrs <- 1

## this works (uses actual_pyrns eventually)
st <- survtab_ag(fot ~ 1, data = x, surv.type = "surv.obs",
                 pyrs = TF$pyrs, d = from0to1,
                 surv.method = "hazard")
## this would be wrong (sees expression 'pyrs' and uses that column,
## which is not what is intended here)
st <- survtab_ag(fot ~ 1, data = x, surv.type = "surv.obs",
                 pyrs = pyrs, d = from0to1,
                 surv.method = "hazard")

```

get.yrs

*Fractional Years***Description**

Using Date objects, calculates given dates as fractional years.

Usage

```
get.yrs(x, year.length = "approx", ...)
```

Arguments

<code>x</code>	a Date object, or anything that <code>[as.Date]</code> accepts
<code>year.length</code>	character string, either "actual" or "approx"; can be abbreviated; see Details
<code>...</code>	additional arguments passed on to <code>[as.Date]</code> ; typically format when <code>x</code> is a character string variable, and origin when <code>x</code> is numeric

Details

`x` should preferably be a Date or IDate object, although it can also be a character string variable which is coerced internally to Date format using `[as.Date.character]`.

When `year.length = 'actual'`, fractional years are calculated as $\text{year} + (\text{day_in_year} - 1) / 365$ for non-leap-years and as $\text{year} + (\text{day_in_year} - 1) / 366$ for leap years. If `year.length = 'approx'`, fractional years are always calculated as $\text{year} + (\text{day_in_year} - 1) / 365.242199$.

There is a slight difference, then, between the two methods when calculating durations between fractional years. For meticulous accuracy one might instead want to calculate durations using dates (days) and convert the results to fractional years.

Note that dates are effectively converted to fractional years at 00:00:01 o'clock:

```
get.yrs("2000-01-01") = 2000, and get.yrs("2000-01-02") = 2000 + 1/365.242199.
```

Value

A numeric vector of fractional years.

Author(s)

Joonas Miettinen

See Also

[Epi::cal.yr], [as.Date.yrs], [as.Date]

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)

data("sire")
sire$dg_yrs <- get.yrs(sire$dg_date)
summary(sire$dg_yrs)

## see: ?as.Date.yrs
dg_date2 <- as.Date(sire$dg_yrs)
summary(as.numeric(dg_date2 - as.Date(sire$dg_date)))

## Epi's cal.yr versus get.yrs
d <- as.Date("2000-01-01")
Epi::cal.yr(d) ## 1999.999
get.yrs(d) ## 2000

## "..." passed on to as.Date, so character / numeric also accepted as input
## (and whatever else as.Date accepts)
get.yrs("2000-06-01")
get.yrs("20000601", format = "%Y%m%d")
get.yrs("1/6/00", format = "%d/%m/%y")

get.yrs(100, origin = "1970-01-01")
```

ICSS

Age standardisation weights from the ICSS scheme.

Description

Contains three sets age-standardisation weights for age-standardized survival (net, relative or observed).

Format

data.table with columns

- age - lower bound of the age group
- ICSS1 - first set of weights, sums to 100 000
- ICSS2 - second set of weights, sums to 100 000
- ICSS3 - third set of weights, sums to 100 000

Source

ICSS weights (US National Cancer Institute website)

Corazziari, Isabella, Mike Quinn, and Riccardo Capocaccia. "Standard cancer patient population for age standardising survival ratios." *European Journal of Cancer* 40.15 (2004): 2307-2316.

See Also

Other popEpi data: [meanpop_fi](#), [popmort](#), [sibr](#), [sire](#), [stdpop101](#), [stdpop18](#)

Other weights: [direct_standardization](#), [stdpop101](#), [stdpop18](#)

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)
## aggregate weights to a subset of age groups
data(ICSS)
cut <- c(0, 30, 50, 70, Inf)
agegr <- cut(ICSS$age, cut, right = FALSE)
aggregate(ICSS1~agegr, data = ICSS, FUN = sum)
```

is.Date

Test if object is a Date object

Description

Test whether obj inherits one of Date or IDate.

Usage

```
is.Date(obj)
```

Arguments

obj object to test on

Value

TRUE if obj is of class "Date" or "IDate".

Author(s)

Joonas Miettinen

See Also

[get.yrs], [is_leap_year], [as.Date]

is_leap_year

Detect leap years

Description

Given a vector or column of year values (numeric or integer), [is_leap_year] returns a vector of equal length of logical indicators, i.e. a vector where corresponding leap years have value TRUE, and FALSE otherwise.

Usage

```
is_leap_year(years)
```

Arguments

years a vector or column of year values (numeric or integer)

Value

A logical vector where TRUE indicates a leap year.

Author(s)

Joonas Miettinen

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)
## can be used to assign new columns easily, e.g. a dummy indicator column
df <- data.frame(yrs=c(1900,1904,2005,1995))
df$lyd <- as.integer(is_leap_year(df$yrs))

## mostly it is useful as a condition or to indicate which rows have leap years
which(is_leap_year(df$yrs)) # 2
```

```
df[is_leap_year(df$yrs),] # 2nd row
```

lexis_funs

Lexis *Datasets*

Description

Make [Epi::Lexis] objects.

Usage

```
Lexis_fpa(
  data,
  birth = NULL,
  entry = NULL,
  exit = NULL,
  entry.status = NULL,
  exit.status = NULL,
  subset = NULL,
  ...
)

Lexis_dt(...)
```

Arguments

data	a data.frame; mandatory
birth	the time of birth; A character string naming the variable in data or an expression to evaluate - see Flexible input
entry	the time at entry to follow-up; supplied the same way as birth
exit	the time at exit from follow-up; supplied the same way as birth
entry.status	passed on to [Epi::Lexis] if not NULL; supplied the same way as birth
exit.status	passed on to [Epi::Lexis] if not NULL; supplied the same way as birth
subset	a logical condition to subset by before passing data and arguments to [Epi::Lexis]
...	arguments passed to [Epi::Lexis]

Value

popEpi::Lexis_fpa

Returns a Lexis object with the additional class data.table. It has the usual columns that Lexis objects have, and with time scale columns fot, per, and age. They are calculated as

fot = entry - entry (to ensure correct format, e.g. difftime)

per = entry

and

age = entry - birth.

popEpi::Lexis_dt

Returns a Lexis object that is also a `data.table` — therefore output has the classes `c("Lexis", "data.table", "data.frame")`.

Functions

popEpi::Lexis_fpa

`popEpi::Lexis_fpa` collects data from its inputs to call `[Epi::Lexis]`. This is a convenience function for making a Lexis object with the time scales `fot`, `per`, and `age`.

popEpi::Lexis_dt

`popEpi::Lexis_dt` performs the following steps:

- Calls `[Epi::Lexis]`.
- Calls `[data.table::setDT]` to make output into a `data.table`, `[data.table::setattr]` to set its class to `c("Lexis", "data.table", "data.frame")`, and `[data.table::setkeyv]` to sort the output by `lex.id` and the time scales.
- Returns the Lexis / `data.table` object.

News for version 0.5.0

`Lexis_fpa` output is now also a `data.table`, so the complete class vector is `c("Lexis", "data.table", "data.frame")`.

New function `Lexis_dt`, a wrapper of `Epi::Lexis` which also sets class to `c("Lexis", "data.table", "data.frame")`.

See Also

Other Lexis_functions: [lexis_merge\(\)](#), [lexis_split_merge_aggregate_by_stratum\(\)](#)

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)

# popEpi::Lexis_fpa
data("sire", package = "popEpi")
lex <- Lexis_fpa(sire,
                 birth = "bi_date",
                 entry = dg_date,
                 exit = ex_date + 1L,
                 exit.status = "status")

## some special cases
```

```

myVar <- "bi_date"
l <- list(myVar = "bi_date")
sire$l <- sire$myVar <- 1

## conflict: myVar taken from data when "bi_date" was intended
lex <- Lexis_fpa(sire,
  birth = myVar,
  entry = dg_date,
  exit = ex_date + 1L,
  exit.status = "status")

## no conflict with names in data
lex <- Lexis_fpa(sire,
  birth = l$myVar,
  entry = dg_date,
  exit = ex_date + 1L,
  exit.status = "status")

stopifnot(
  identical(class(lex), c("Lexis", "data.table", "data.frame")),
  Epi::timeScales(lex) %in% c("fot", "per", "age")
)

# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)

# popEpi::Lexis_dt
lex_1 <- popEpi::Lexis_dt(
  data = popEpi::sire,
  entry = list(
    ts_fut = 0L,
    ts_age = as.integer(dg_date - bi_date),
    ts_cal = as.integer(dg_date)
  ),
  exit = list(ts_cal = as.integer(ex_date)),
  entry.status = 0L,
  exit.status = status
)
stopifnot(
  class(lex_1) == c("Lexis", "data.table", "data.frame"),
  Epi::timeScales(lex_1) %in% c("ts_fut", "ts_age", "ts_cal")
)

lex_2 <- popEpi::Lexis_dt(
  data = popEpi::sire,
  entry = list(
    ts_fut = 0L,
    ts_age = as.integer(dg_date - bi_date),
    ts_cal = as.integer(dg_date)
  ),
  duration = as.integer(ex_date - dg_date),

```

```

    entry.status = 0L,
    exit.status = status
  )
  stopifnot(
    class(lex_2) == c("Lexis", "data.table", "data.frame"),
    Epi::timeScales(lex_2) %in% c("ts_fut", "ts_age", "ts_cal")
  )

```

lexis_merge

Merge Data into Lexis Object

Description

Function(s) to merge data into Lexis objects intelligently when merging is (partially) based on time scales.

Usage

```

lexis_merge(
  lexis,
  merge_dt,
  merge_dt_by,
  merge_dt_harmonisers = NULL,
  subset = NULL,
  optional_steps = NULL,
  lex_dur_multiplier = NULL
)

```

Arguments

lexis	[Lexis] (no default) A Lexis dataset ([Epi::Lexis] / [Lexis_dt]).
merge_dt	[data.table] (no default) A data.table to merge with lexis. Usually lexis has been split with e.g. [splitMulti].
merge_dt_by	[character] (no default) Names of columns in both merge_dt and lexis by which merge_dt will be merged with lexis.
merge_dt_harmonisers	[NULL, list] (default NULL) Optional list of quoted expressions which, when evaluated, harmonise data in lexis (possibly after splitting) to look the same as the data in merge_dt. For example, if merge_dt contains expected hazards by calendar year ts_cal and 1-year age group ts_age, and if lexis has been split into monthly survival intervals, then we must somehow find the correct row in merge_dt for each row in lexis. E.g. ts_cal = 2010.5323 and ts_age = 76.4435 need to be

harmonised into values found in merge_dt such as ts_cal = 2010 and ts_age = 76.

- **NULL**: This function comes up with reasonable harmonisers if possible. See **Details**.
- **list**: Each element must a quoted expression ([quote]) and each element must have a name corresponding to a column name in both lexis and merge_dt. See **Examples**.

subset [NULL, logical, integer] (default NULL)

Merge data only into specific rows in lexis.

- **NULL**: All rows.
- **logical**: Only rows where this is TRUE. Must of length nrow(lexis).
- **integer**: Only rows identified by these indices. Every index must be in 1:nrow(lexis).

optional_steps [NULL, list] (default NULL)

Optional steps to perform during the function's run.

- **NULL**: No additional steps are performed.
- **list**: Each element is named and a function. See **Details** For what each functions you can make use of what their arguments should be.

lex_dur_multiplier

[NULL, integer, numeric] (default NULL)

- **NULL**: Use 0.5.
- **integer / numeric**: Use this is multiplier.

lex_dur_multiplier is used in the following expression when a default harmoniser is being made: `col + lex.dur * lex_dur_multiplier`. Here `col` is the column in question. When merging population expected hazards into split lexis data it is reasonable to use `lex_dur_multiplier = 0.5`, e.g. with `ts_cal = 2001.9`, `ts_fut = 0.0`, and `lex_dur = 0.5` we arrive to the middle of the interval (or end of follow-up for that subject) at `ts_cal = 2002.15` and `ts_fut = 0.25`. Here it is more reasonable to merge data from 2002 rather than 2001 because the majority of follow-up occurs in 2002 for that record.

When `lex_dur_multiplier == 0` we simplify the expression to just `col` instead of using `col + lex.dur * 0`. This handles correctly edge cases such as `lex.dur = Inf`.

As an aside, of course if you split also by calendar time (and age and whatever you have in your merge_dt) then this makes no difference because every split lexis record is strictly within each interval of your merge_dt. But in practice this does not improve much and can be computationally costly.

Value

popEpi::lexis_merge

Returns lexis after adding value columns from merge_dt into lexis.

Functions

popEpi::lexis_merge

popEpi::lexis_merge can be used to merge additional information into Lexis data, allowing the use of the Lexis time scales in the merge. The typical use-case is to split Lexis data and then merge population (expected) hazards to the subject-intervals. popEpi::lexis_merge performs the following steps:

- Run optional_steps[["on_entry"]](call_env = call_env, eval_env = eval_env) if that optional_steps element exists. call_env is the environment where this function was called and eval_env is the temporary environment in which the commands that this function consists of are evaluated.
- Run on.exit(optional_steps[["on_exit"]](call_env = call_env, eval_env = eval_env)) if that optional_steps element exists.
- If is.null(merge_dt_harmonisers), attempt to automatically determine the harmonisers making use of cut for each time scale column to merge by as follows:
 - If merge_dt[[col_nm]] is a factor column, look at the levels() of the column. If they all look like [x, y[,]x, y], or x - y, extract breaks from them, infer whether right = FALSE or TRUE, and use the levels() as labels. E.g. factor("[60, 61[") produces list(breaks = 60:61, right = FALSE, labels = "[60, 61[").
 - If merge_dt[[col_nm]] contains numbers we define cut breaks as the unique values of merge_dt[[col_nm]] and as the ceiling max(merge_dt[[col_nm]]) + last_diff. Here last_diff is the difference between the highest and second highest values. E.g. c(1950, 1960, 1970:2020, 2021) for merge_dt[[col_nm]] containing unique values c(1950, 1960, 1970:2020).
 - If merge_dt[[col_nm]] is not of class integer, numeric, or factor, an error is raised because we don't know how to automatically form a harmoniser.
 - Run optional_steps[["pre_default_harmoniser_creation"]](call_env = call_env, eval_env = eval_env, lapply_eval_env = lapply_eval_env) if that optional_steps element exists. Here make_harmoniser_eval_env is similar to eval_env but it is the evaluation environment of the function passed to lapply which attempts to make each default harmoniser.
 - With the cut breaks defined, the automatically created harmoniser becomes a cut call with arguments inferred above and with
 - * x = col + lex.dur * lex_dur_multiplier, where col is the current column.
 - * dig.lab = 10.
 - Depending on what we are harmonising with, the harmoniser will return either a factor with labels based on the cut call, e.g. ts_cal = 2021.524 turns into [2021, 2022[with both 2021 and 2022 in breaks, or the identified interval's lower bound such as 2021.
 - Run optional_steps[["post_default_harmoniser_creation"]](call_env = call_env, eval_env = eval_env, lapply_eval_env = lapply_eval_env) if that optional_steps element exists.
- Run optional_steps[["post_merge_dt_harmonisers"]](call_env = call_env, eval_env = eval_env) if that optional_steps element exists.
- Armed with either user-defined or automatically created merge_dt_harmonisers, they are each evaluated to create a temporary data.table with harmonised data from lexis. This is

performed via eval with `envir = lexis` and `enclos = harmoniser_eval_env` where `harmoniser_eval_env` is a temporary environment which contains every argument passed to `lexis_merge` as well as `eval_env` and `call_env` which you can read about in the detailed explanation of `optional_steps`. `lexis` is a subset if `!is.null(subset)`. However, if a column has no harmoniser at this point then it is used as-is. For instance there is no need to harmonise stratifying columns such as `area` because they are not changed by splitting the Lexis data.

- Run `optional_steps[["post_harmonisation"]](call_env = call_env, eval_env = eval_env)` if that `optional_steps` element exists.
- Then we perform the actual merge between `merge_dt` and the harmonised data. The value columns in `merge_dt` are added into `lexis` either using `data.table::set` for a `data.table` or a simple `lexis[, merge_value_col_nms] <- join_result` assignment. Note that `data.table::set` modifies the `lexis` object in-place.
- Run `optional_steps[["post_merge"]](call_env = call_env, eval_env = eval_env)` if that `optional_steps` element exists.
- Each merged-in column from `merge_dt` (all columns not in `merge_dt_by`) are inspected for missing values. If there are any, an error is raised. This usually occurs if `merge_dt` does not contain data for all data in `lexis`. For instance it only covers years 1950-2020 but `lexis` contains also data for 2021. This error helps you to spot those problems early instead of producing nonsense results downstream.
- Returns `lexis` after adding value columns from `merge_dt` into `lexis`.

See Also

Other Lexis_functions: [lexis_funs](#), [lexis_split_merge_aggregate_by_stratum\(\)](#)

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)
# popEpi::lexis_merge
lexis <- Epi::Lexis(
  entry = list(ts_fut = 0.0, ts_cal = 2010.3, ts_age = 56.8),
  exit = list(ts_cal = 2024.9999),
  entry.status = 0L,
  exit.status = 0L
)
lexis$sex <- 0L
lexis <- popEpi::splitMulti(
  data = lexis,
  breaks = list(ts_fut = seq(0, 3, 1 / 12))
)
my_merge_dt <- data.table::CJ(sex = 0:1, ts_age = 0:100, ts_cal = 2000:2025)
data.table::set(
  x = my_merge_dt,
  j = "merge_value",
  value = runif(nrow(my_merge_dt))
)
```

```

)
# lexis is also a data.table and is modified in-place. no need to keep
# output of popEpi::lexis_merge call.
popEpi::lexis_merge(
  lexis = lexis,
  merge_dt = my_merge_dt,
  merge_dt_by = c("sex", "ts_age", "ts_cal")
)
stopifnot(
  "merge_value" %in% names(lexis),
  !is.na(lexis[["merge_value"]])
)
data.table::set(
  x = lexis,
  j = "merge_value",
  value = NULL
)
popEpi::lexis_merge(
  lexis = lexis,
  merge_dt = my_merge_dt,
  merge_dt_by = c("sex", "ts_age", "ts_cal"),
  merge_dt_harmonisers = list(
    ts_cal = quote(as.integer(ts_cal)),
    ts_age = quote(as.integer(ts_age))
  )
)
stopifnot(
  "merge_value" %in% names(lexis),
  !is.na(lexis[["merge_value"]])
)

# special factor columns in merge_dt
data.table::set(
  x = lexis,
  j = "merge_value",
  value = NULL
)
data.table::set(
  x = my_merge_dt,
  j = c("ts_age", "ts_cal"),
  value = list(
    cut(my_merge_dt[["ts_age"]], breaks = 0:101, right = FALSE),
    cut(my_merge_dt[["ts_cal"]], breaks = 2000:2026, right = FALSE,
        dig.lab = 4)
  )
)
popEpi::lexis_merge(
  lexis = lexis,
  merge_dt = my_merge_dt,
  merge_dt_by = c("sex", "ts_age", "ts_cal")
)
stopifnot(
  "merge_value" %in% names(lexis),

```

```

    !is.na(lexis[["merge_value"]])
  )

```

lexis_split_merge_aggregate_by_stratum

Split, Merge, and Aggregate a Lexis Object

Description

Function(s) which split, merge, and aggregate a Lexis object in one go.

Usage

```

lexis_split_merge_aggregate_by_stratum(
  lexis,
  breaks,
  aggre_exprs,
  aggre_by = NULL,
  merge_dt = NULL,
  merge_dt_by = NULL,
  merge_optional_args = NULL,
  subset = NULL,
  first_record_by = NULL,
  weight_col_nm = NULL,
  split_lexis_column_exprs = NULL,
  breaks_collapse_args = NULL,
  optional_steps = NULL
)

```

Arguments

lexis	[Lexis] (no default) A Lexis dataset ([Epi::Lexis] / [Lexis_dt]).
breaks	[list] (no default) List of breaks to split Lexis data by. Passed to [splitMulti]. E.g. list(ts_cal = c(2001, 2006), ts_fut = seq(0, 5, 1 / 12)).
aggre_exprs	[character, list] (no default) Defines what is aggregated within every stratum-box defined via aggre_by and breaks. See Details for how and where the expressions are evaluated. Each element must be either named and an R expression (see e.g. [quote]) or a character string which identifies the aggregation to perform from a table of pre-defined expressions within popEpi (shown in Details). E.g. c("n_events", "t_at_risk"), list("n_events", t_at_risk = quote(sum(lex.dur))).
aggre_by	[data.table, character, list, NULL] (default NULL) • NULL: No stratification of output.

- `data.table`: Compute produce results for each stratum defined in this table, e.g. `data.table::data.table(sex = 1:2)`. You may even use this to take a subset at the same time by doing e.g. `data.table::data.table(sex = 1L)` even if your `dt` contains data for both sexes.
- `character`: Compute results for each stratum defined by these columns in `dt`. E.g. `"sex"`.
- `list`: Each element must be either a `data.table`, a `character` vector, or `NULL`. These are combined to yield a big `data.table`. E.g. `list("sex", data.table::data.table(ag = 1:18))` leading to the same as `data.table::CJ(sex = 1:2, ag = 1:18)`.

`merge_dt` Passed to `[lexis_merge]`.

`merge_dt_by` Passed to `[lexis_merge]`.

`merge_optional_args`

`[NULL, list]` (default `NULL`)

Each element passed to `[lexis_merge]`. E.g. `list(merge_dt_harmonisers = my_harmonisers)`.

`subset` `[NULL, logical, integer, data.table]` (default `NULL`)

This argument is evaluated within the context of `dt` and the environment where the function with this argument was called.

- `NULL`: Implies no subsetting, i.e. retain all observations.
- `logical`: Keep only observations where this is `TRUE`. Must be of length `nrow(dt)`. E.g. `subset = my_column == 1`.
- `integer`: Keep only observations at these row numbers. E.g. `subset = 1:5`.
- `data.table`: Each column must also be a column of `dt`. Only those rows in `dt` are retained that match a row in `subset`. E.g. `subset = data.table::data.table(my_column = 1)`.

Due to the non-standard evaluation method used here, you can make use of data in the calling environment as well. E.g. with variable `my_value = 1` defined and `dt` containing column `my_column` you can do `subset = my_column == my_value` if you want.

`first_record_by`

`[NULL, character]` (default `NULL`)

Optional argument to only use each nonduplicate record as identified by these column names. See **Examples**.

- `NULL`: No handling of multiples.
- `character`: Only use the first record as identified by these column names in `lexis`. E.g. `first_record_by = "lex.id"` causes only the first record by `lex.id` to be included in the calculations. Firstness is based on the order of the records as they are in `lexis` and this function does not sort the data first in any way. None of the Lexis time scale columns nor `lex.dur` are allowed to be used.

`weight_col_nm` `[NULL, character]` (default `NULL`)

Name of weight column in `lexis` if you want to perform individual weighting. See **Details** for where this comes into play.

- NULL: No individual weighting is performed.
 - character: This is the name of the column. E.g. "my_iw".
- split_lexis_column_exprs
[NULL, list]
Additional columns to create in Lexis data after splitting and before aggregation. Any column you create this way can be the used in an aggregation expression.
- NULL: Don't create additional columns.
 - list: Each element is named and an R expression object, e.g. list(my_col = quote(my_function(lex.Cst, lex.Xst))). See **Details** for more information on when and how these expressions are evaluated.
- breaks_collapse_args
[NULL, list] (default NULL)
Optional, if you supply this argument then [lexis_breaks_collapse_1d] will be called for each stratum defined via aggre_by separately.
- NULL: [lexis_breaks_collapse_1d] is not called.
 - list: E.g. list(mandatory_breaks = 0:5).
- optional_steps [NULL, list] (default NULL)
Optional steps to perform along the way.
- NULL: No optional steps are performed.
 - list: Each named element is a function that is called in a specific stage of the run. See **Details** for what functions are recognised.

Details

popEpi::lexis_split_merge_aggregate_by_stratum can be used to split Lexis ([Epi::Lexis]) data, merge something to it after the merge, and then perform an aggregation step. The following steps are performed:

- Handle aggre_exprs as follows:
 - An aggre_exprs element that is string such as "n_events" is replaced with the appropriate expression retrieved from a table of pre-defined expressions (see below).
 - If a string element is transition-specific, e.g. n_events_[0, 1], we first turn it into its general form (e.g. n_events_[x, y]). Additionally, any uses of x and y in the fetched expression is replaced with the correct states, e.g. sum(lex.Cst == x & lex.Xst == y) is turned into sum(lex.Cst == 0 & lex.Xst == 1).
 - An aggre_exprs element that is an expression already (e.g. my_n_events = quote(sum(lex.Cst == 0 & lex.Xst != 0))) does not receive the same treatment regarding the states and when you write your own expressions you must specify them yourself. Therefore e.g. my_n_events = quote(sum(lex.Cst == x & lex.Xst != y)) will not work.
 - Regardless of the type of the aggre_exprs element, we remove every use of * iw in the expression if individual weights are not used.
 - Table of general aggregation expressions known to popEpi:

Name	Explanation
------	-------------

<code>n_at_risk_eff</code>	Effective number of subjects at risk. Whereas in the Kaplan-Meier (product-limit) estimator, the effective number of subjects at risk is the number of subjects at risk at the start of the interval, in the log-rank test, the effective number of subjects at risk is the average number of subjects at risk over the interval.
<code>n_at_risk_eff_bw</code>	Effective number of subjects at risk. Whereas in the Kaplan-Meier (product-limit) estimator, the effective number of subjects at risk is the number of subjects at risk at the start of the interval, in the log-rank test, the effective number of subjects at risk is the average number of subjects at risk over the interval.
<code>n_at_risk_eff_pp</code>	The same as <code>n_at_risk_eff</code> but with Pohar Perme weighting.
<code>n_at_risk_eff_pp_bw</code>	The same as <code>n_at_risk_eff</code> but with Pohar Perme weighting. The <code>iw</code> in the expression stands for the individual weight.
<code>n_entered_late_during_interval</code>	Number of Lexis subjects who entered a follow-up time scale interval late.
<code>n_entered_late_during_interval_bw</code>	Number of Lexis subjects who entered a follow-up time scale interval late. The <code>iw</code> in the expression stands for the individual weight.
<code>n_events</code>	Number of events, i.e. the total number of all kinds of transitions.
<code>n_events[x, y]</code>	The number of transitions from <code>x</code> to <code>y</code> .
<code>n_events[x, y]_bw</code>	The number of transitions from <code>x</code> to <code>y</code> . The <code>iw</code> in the expression stands for the individual weight.
<code>n_events_bw</code>	Number of events, i.e. the total number of all kinds of transitions. The <code>iw</code> in the expression stands for the individual weight.
<code>n_events_exp_e2</code>	Number of expected events according to the Ederer II method. Well, there is one problem with the Ederer II method: it does not take into account the fact that subjects who enter late during the interval are not at risk for the whole interval.
<code>n_events_exp_e2_bw</code>	Number of expected events according to the Ederer II method. Well, there is one problem with the Ederer II method: it does not take into account the fact that subjects who enter late during the interval are not at risk for the whole interval.
<code>n_events_exp_pp</code>	The number of expected events with Pohar Perme weighting.
<code>n_events_exp_pp_bw</code>	The number of expected events with Pohar Perme weighting. The <code>iw</code> in the expression stands for the individual weight.
<code>n_events_pp</code>	The same <code>n_events</code> but with Pohar Perme weighting.
<code>n_events_pp_bw</code>	The same <code>n_events</code> but with Pohar Perme weighting. The <code>iw</code> in the expression stands for the individual weight.
<code>n_events_pp_double_weighted</code>	The same as <code>n_events</code> but with Pohar Perme weighting where the weights are squared.
<code>n_events_pp_double_weighted_bw</code>	The same as <code>n_events</code> but with Pohar Perme weighting where the weights are squared. The <code>iw</code> in the expression stands for the individual weight.
<code>n_in_follow_up_at_interval_start</code>	Number of Lexis subjects in follow-up at the start of a follow-up time scale interval.
<code>n_in_follow_up_at_interval_start_bw</code>	Number of Lexis subjects in follow-up at the start of a follow-up time scale interval. The <code>iw</code> in the expression stands for the individual weight.
<code>n_left_early_during_interval</code>	Number of Lexis subjects who left a follow-up time scale interval early.
<code>n_left_early_during_interval_bw</code>	Number of Lexis subjects who left a follow-up time scale interval early. The <code>iw</code> in the expression stands for the individual weight.
<code>t_at_risk</code>	Total time at risk.
<code>t_at_risk_bw</code>	Total time at risk. The <code>iw</code> in the expression stands for the individual weight.
<code>t_at_risk_pp</code>	The same as <code>t_at_risk</code> but with Pohar Perme weighting.
<code>t_at_risk_pp_bw</code>	The same as <code>t_at_risk</code> but with Pohar Perme weighting. The <code>iw</code> in the expression stands for the individual weight.

- If `weight_col_nm` was supplied, include in subset only records where `lexis[[weight_col_nm]] > 0`.
- Call `optional_steps[["on_entry"]](eval_env = eval_env, call_env = call_env)` if that `optional_steps` element exists. `eval_env` is the temporary evaluation environment of `popEpi::lexis_split_merge_aggregate_by_stratum` which contains all the arguments of `popEpi::lexis_split_merge_aggregate_by_stratum` and `call_env` is the environment where it was called.
- Call `on.exit(optional_steps[["on_exit"]](eval_env = eval_env, call_env = call_env))` if that `optional_steps` element exists.
- For each stratum in `aggre_by`:
 - Run `optional_steps[["stratum_on_entry"]](stratum_eval_env = stratum_eval_env, eval_env = eval_env, call_env = call_env)` if that `optional_steps` element exists. `stratum_eval_env` is the environment where the stratum-specific steps are performed.
 - Loop over every box defined by all other time scales except the last one. E.g. if `breaks = list(ts_cal = c(2001, 2004, 2007), ts_fut = 0:5)` then we perform what follows below separately for the boxes `[2001, 2004]`, `[2004, 2007]`.
 - * Run `optional_steps[["stratum_pre_split"]](stratum_eval_env = stratum_eval_env, eval_env = eval_env, call_env = call_env)` if that `optional_steps` element exists.

- * Using a subset lexis containing data only for the current stratum (e.g. `sex = 0`), drop data outside the current stratum box (e.g. `ts_cal` outside of `]2001, 2004]`). Also crop follow-up to the end of the box. This does not split the data yet.
- * If `!is.null(breaks_collapse_args)`, run `popEpi::lexis_breaks_collapse_1d` on the last element of `breaks`. Arguments `lexis` and `breaks` are set automatically. This makes it possible to "collapse" the breaks for each stratum and stratum box combination separately.
- * Run `[splitMulti]` for the second time, this time splitting the data by the last time scale in `breaks`. Remember that we are inside the box defined by the other time scales already.
- * Run `optional_steps[["stratum_post_split"]]`(`stratum_eval_env = stratum_eval_env`, `eval_env = eval_env`, `call_env = call_env`) if that `optional_steps` element exists.
- * Run `[lexis_merge]` with `merge_dt`, `merge_dt_by`, and `merged_optional_args`, if `merge_dt` has been supplied.
- * Run `optional_steps[["stratum_post_merge"]]`(`stratum_eval_env = stratum_eval_env`, `eval_env = eval_env`, `call_env = call_env`) if that `optional_steps` element exists.
- * Evaluate `aggre_exprs` as follows:
 - First collect variables mentioned in `aggre_exprs` using `[all.vars]`.
 - There is a pre-defined table of expressions that are evaluated and added as new columns in the split data if they appear in any of the `aggre_exprs`. See below for the table.
 - Table of expressions which create new columns into split data:

Name	Explanation
<code>in_follow_up_at_interval_start</code>	This column indicates (TRUE / FALSE) whether a subject is in follow-up at the start of
<code>entered_late_during_interval</code>	This column indicates (TRUE / FALSE) whether a subject entered a follow-up time scale
<code>left_early_during_interval</code>	This column indicates (TRUE / FALSE) whether a subject left a follow-up time scale in
<code>pp</code>	This is the ""Pohar Perme weight"" which we use to weigh summary statistics such as

- + It can happen that e.g. a survival interval has absolutely no data in it. Especially in sparse data and with delayed entry. For the pre-specified aggregation expressions such as ``n_events`` we ensure that empty time scale boxes have value zero. Your custom aggregation expressions will result in ``NA`` values in empty time scale boxes.

* Run

```
`optional_steps[["stratum_post_aggregation"]](stratum_eval_env = stratum_eval_env, eval_env = eval_env, call_env = call_env)
```

if that ``optional_steps`` element exists.

- Replace NA values in value columns starting with `t_` or with `n_` with zeroes. E.g. `t_at_risk` will then be zero in intervals with no subjects in follow-up. Even a custom expression such as `t_at_risk_squared` will be handled in this manner.
- Set proper `data.table` attributes on the resulting big table and sort it by the stratifying variables.

- Store as metadata a list with names `stratum_col_nms`, `ts_col_nms`, and `value_col_nms` into the attribute named `lexis_split_merge_aggregate_by_stratum_meta`, where `stratum_col_nms = names(aggre_by)`, `ts_col_nms = names(breaks)`, and `value_col_nms` are the names of the columns resulting from `aggre_exprs`.
- Run `optional_steps[["post_aggregation"]](eval_env = eval_env, call_env = call_env)` if that `optional_steps` element exists.
- Return a `data.table` with stratum columns as specified via `aggre_by`, time scale columns specified via `breaks` and value columns as specified via `aggre_exprs`.

Value

Returns a `data.table` with stratum columns as specified via `aggre_by`, time scale columns specified via `breaks` and value columns as specified via `aggre_exprs`.

See Also

Other Lexis_functions: [lexis_funs](#), [lexis_merge\(\)](#)

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)
```

```
# popEpi::split_merge_aggregate_by_stratum
make_pm <- function() {
  pm <- data.table::copy(popEpi::popmort)
  data.table::setnames(
    pm,
    c("year", "agegroup", "haz"),
    c("ts_cal", "ts_age", "h_exp")
  )
  data.table::setkeyv(pm, c("sex", "ts_cal", "ts_age"))
  data.table::setkeyv(pm, c("sex", "ts_cal", "ts_age"))
  return(pm[])
}
```

```
make_column_icss_ag <- function(age) {
  cut(
    age,
    breaks = c(0, 60, 70, 80, Inf),
    right = FALSE,
    labels = c("0-59", "60-69", "70-79", "80+")
  )
}
```

```
make_standard_weight_dt <- function() {
  return(popEpi::ICSS[
    j = list(
```

```

        weight = as.double(sum(.SD[["ICSS1"]]))
      ),
      keyby = list(
        icss_ag = make_column_icss_ag(popEpi::ICSS[["age"]])
      )
    ][])
  }

make_sire <- function() {
  sire <- popEpi::sire
  sire <- sire[
    sire[["dg_date"]] < sire[["ex_date"]] &
    data.table::between(
      sire[["ex_date"]],
      as.Date("1999-01-01"),
      as.Date("2003-12-31"),
      incbounds = TRUE
    ) &
    (get.yrs(sire[["ex_date"]]) - get.yrs(sire[["bi_date"]])) < 100
  ]
  sire[j = "my_stratum" := sample(2L, size = nrow(sire), replace = TRUE)]
  sire <- sire[
    j = .SD[as.integer(seq(1L, .N, length.out = 50L))],
    keyby = "my_stratum"
  ]
  # you can also use popEpi::Lexis_dt
  sire <- Epi::Lexis(
    entry = list(
      ts_cal = popEpi::get.yrs(dg_date),
      ts_age = popEpi::get.yrs(dg_date) - popEpi::get.yrs(bi_date),
      ts_fut = 0.0
    ),
    duration = popEpi::get.yrs(ex_date) - popEpi::get.yrs(dg_date),
    entry.status = 0L,
    exit.status = status,
    data = sire
  )
  sire[["icss_ag"]] <- make_column_icss_ag(sire[["dg_age"]])
  sire
}

pm <- make_pm()
wdt <- make_standard_weight_dt()
sire <- make_sire()

# note that we use here 1-year survival intervals which are too wide for
# practical survival estimation. we also aggregate for period analysis
# for demonstration purposes.
bl <- list(ts_cal = c(1999, 2004), ts_fut = 0:5)

# using some pre-defined aggregation expressions
agdt <- popEpi::lexis_split_merge_aggregate_by_stratum(
  lexis = sire,

```

```

    breaks = bl,
    aggre_exprs = c("n_events", "t_at_risk", "n_events_[0, 1]"),
    aggre_by = "my_stratum"
  )
  stopifnot(
    c("n_events", "t_at_risk", "n_events_[0, 1]") %in% names(agdt)
  )

  # using some pre-defined aggregation expressions + using expected hazard.
  agdt <- popEpi::lexis_split_merge_aggregate_by_stratum(
    lexis = sire,
    breaks = bl,
    aggre_exprs = c(
      "n_events", "t_at_risk", "n_events_[0, 1]",
      "n_events_exp_e2"
    ),
    aggre_by = "my_stratum",
    merge_dt = pm,
    merge_dt_by = c("sex", "ts_cal", "ts_age")
  )

  # using custom expressions --- `h_exp` is merged from `merge_dt`
  agdt <- popEpi::lexis_split_merge_aggregate_by_stratum(
    lexis = sire,
    breaks = bl,
    aggre_exprs = list(
      "n_events", "t_at_risk", "n_events_[0, 1]",
      "n_events_exp_e2",
      "my_n_events" = quote(sum(lex.Cst == 0 & lex.Xst != 0)),
      "my_n_events_exp_e2" = quote(sum(h_exp * lex.dur))
    ),
    aggre_by = "my_stratum",
    merge_dt = pm,
    merge_dt_by = c("sex", "ts_cal", "ts_age")
  )
  stopifnot(
    agdt[["my_n_events"]] == agdt[["n_events"]],
    all.equal(agdt[["my_n_events_exp_e2"]], agdt[["n_events_exp_e2"]])
  )

  # using custom expressions --- some pre-defined variables which are added
  # automatically
  agdt <- popEpi::lexis_split_merge_aggregate_by_stratum(
    lexis = sire,
    breaks = bl,
    aggre_exprs = list(
      "n_events_pp",
      "my_n_events_pp" = quote(sum((lex.Cst == 0 & lex.Xst != 0) * pp)),
      "n_at_risk_eff",
      "my_n_at_risk_eff" = quote(sum(
        in_follow_up_at_interval_start +
        0.5 * entered_late_during_interval -
        0.5 * left_early_during_interval
      ))
    )
  )

```

```

)),
"my_n_at_risk_eff_2" = quote({
  # custom definition for detecting late entry. in this example we add
  # a larger tolerance than is used normally. you can find the normal
  # tolerance used in the table of expressions which create columns into
  # the split lexis dataset.
  ts_fut_floor <- eval_env[["box_dt"]][["ts_fut_start"]][box_id]
  distance_from_floor <- (ts_fut - ts_fut_floor)
  entered_late_during_interval <- distance_from_floor > 1e-3
  sum(
    in_follow_up_at_interval_start +
    0.5 * entered_late_during_interval -
    0.5 * left_early_during_interval
  )
})
),
aggre_by = "my_stratum",
merge_dt = pm,
merge_dt_by = c("sex", "ts_cal", "ts_age")
)
stopifnot(
  all.equal(agdt[["my_n_events_pp"]], agdt[["n_events_pp"]]),
  all.equal(agdt[["n_at_risk_eff"]], agdt[["my_n_at_risk_eff"]]),
  all.equal(agdt[["n_at_risk_eff"]], agdt[["my_n_at_risk_eff_2"]])
)

# using custom expressions --- also a custom column at the split lexis level.
# this is for demonstration of what is possible and probably no-one would
# need such an h_exp_mean in a real application.
agdt <- popEpi::lexis_split_merge_aggregate_by_stratum(
  lexis = sire,
  breaks = bl,
  aggre_exprs = list(
    "my_stat" = quote(sum(lex.dur * h_exp_mean))
  ),
  aggre_by = "my_stratum",
  merge_dt = pm,
  merge_dt_by = c("sex", "ts_cal", "ts_age"),
  split_lexis_column_exprs = list(
    h_exp_mean = quote({
      dt <- data.table::setDT(list(lex.id = lex.id, h_exp = h_exp))
      h_exp_lag1 <- dt[
        j = list(lag1 = data.table::shift(h_exp, n = 1L, type = "lag")),
        by = "lex.id"
      ][["lag1"]]
      h_exp_lag1[is.na(h_exp_lag1)] <- h_exp[is.na(h_exp_lag1)]
      h_exp_lead1 <- dt[
        j = list(lead1 = data.table::shift(h_exp, n = 1L, type = "lead")),
        by = "lex.id"
      ][["lead1"]]
      h_exp_lead1[is.na(h_exp_lead1)] <- h_exp[is.na(h_exp_lead1)]
      (h_exp + h_exp_lag1 + h_exp_lead1) / 3
    })
  )
)

```

```

    )
  )
  stopifnot(
    "my_stat" %in% names(agdt),
    !is.na(agdt[["my_stat"]])
  )

  # this function was written for survival estimation but is in fact more
  # general. for instance we can look at a calendar year / age grid.
  agdt <- popEpi::lexis_split_merge_aggregate_by_stratum(
    lexis = sire,
    breaks = list(ts_cal = 1999:2004, ts_age = seq(0, 100, 10)),
    aggre_exprs = list(
      "n_events_exp_e2"
    ),
    merge_dt = pm,
    merge_dt_by = c("sex", "ts_cal", "ts_age")
  )
  stopifnot(
    c("ts_cal_start", "ts_age_start") %in% names(agdt)
  )
  agdt_wide <- data.table::dcast(
    data = agdt,
    formula = ts_age_start ~ ts_cal_start,
    value.var = "n_events_exp_e2"
  )
  # print(round(agdt_wide, 2))
  # Key: <ts_age_start>
  #   ts_age_start 1999 2000 2001 2002 2003
  #   <num> <num> <num> <num> <num> <num>
  # 1:      0 0.00 0.00 0.00 0.00 0.00
  # 2:     10 0.00 0.00 0.00 0.00 0.00
  # 3:     20 0.00 0.00 0.00 0.00 0.00
  # 4:     30 0.00 0.00 0.00 0.00 0.00
  # 5:     40 0.00 0.00 0.00 0.00 0.00
  # 6:     50 0.01 0.01 0.01 0.00 0.00
  # 7:     60 0.04 0.04 0.03 0.01 0.02
  # 8:     70 0.33 0.28 0.31 0.16 0.05
  # 9:     80 1.36 1.04 0.89 0.48 0.20
  # 10:    90 0.39 0.29 0.47 0.37 0.56

  # this is what happens when there is an empty interval
  agdt <- popEpi::lexis_split_merge_aggregate_by_stratum(
    lexis = popEpi::Lexis_dt(
      entry = list(ts_fut = c(0.0, 3.5)),
      exit = list(ts_fut = c(1.5, 4.5)),
      entry.status = 0L,
      exit.status = 1L
    ),
    breaks = list(ts_fut = 0:5),
    aggre_exprs = list(
      "t_at_risk",
      "n_events",

```

```

    my_n_events = quote(sum(lex.Cst == 0 & lex.Xst != 0))
  )
)
stopifnot(
  identical(agdt[["t_at_risk"]], c(1.0, 0.5, 0.0, 0.5, 0.5)),
  identical(agdt[["n_events"]], c(0L, 1L, 0L, 0L, 1L)),
  identical(agdt[["my_n_events"]], c(0L, 1L, NA, 0L, 1L))
)

# this is what happens when an empty stratum is requested
agdt <- popEpi::lexis_split_merge_aggregate_by_stratum(
  lexis = sire,
  aggre_by = data.table::data.table(sex = 0:2),
  breaks = list(ts_fut = 0:5),
  aggre_exprs = list(
    "t_at_risk",
    "my_stat" = quote(sum(lex.dur ^ 2)),
    "t_at_risk_squared" = quote(sum(lex.dur ^ 2))
  )
)
stopifnot(
  nrow(agdt) == length(0:2) * (length(0:5) - 1),
  agdt[["t_at_risk"]][!agdt[["sex"]] %in% sire[["sex"]]] == 0,
  is.na(agdt[["my_stat"]][!agdt[["sex"]] %in% sire[["sex"]]]),
  agdt[["t_at_risk_squared"]][!agdt[["sex"]] %in% sire[["sex"]]] == 0
)

# and this is what happens when no records are in the dataset
agdt <- popEpi::lexis_split_merge_aggregate_by_stratum(
  lexis = sire,
  subset = rep(FALSE, nrow(sire)),
  aggre_by = data.table::data.table(sex = 0:2),
  breaks = list(ts_fut = 0:5),
  aggre_exprs = list(
    "t_at_risk",
    "my_stat" = quote(sum(lex.dur ^ 2))
  )
)
stopifnot(
  nrow(agdt) == length(0:2) * (length(0:5) - 1),
  agdt[["t_at_risk"]] == 0,
  is.na(agdt[["my_stat"]])
)

# and this is what happens when no records are in the dataset
agdt <- popEpi::lexis_split_merge_aggregate_by_stratum(
  lexis = sire,
  subset = rep(FALSE, nrow(sire)),
  aggre_by = data.table::data.table(sex = 0:2),
  breaks = list(ts_fut = 0:5),
  aggre_exprs = list(
    "t_at_risk",
    "my_stat" = quote(sum(lex.dur ^ 2))
  )
)

```

```

    )
  )
  stopifnot(
    nrow(agdt) == length(0:2) * (length(0:5) - 1),
    agdt[["t_at_risk"]] == 0,
    is.na(agdt[["my_stat"]])
  )

  # handling multiple records for one subject.
  # often in survival analysis we want to use the first record per output stratum.
  # first, the normal behaviour without handling multiples in any way:
  lexis_mult <- popEpi::Lexis_dt(
    entry = list(ts_fut = c(0.0, 0.0, 0.0)),
    exit = list(ts_fut = c(1.5, 4.5, 2.5)),
    entry.status = 0L,
    exit.status = 1L,
    id = c(1L, 1L, 1L),
    data = data.frame(sex = 0L, age_group = c("45-49", "45-49", "50-54"))
  )
  agdt <- popEpi::lexis_split_merge_aggregate_by_stratum(
    lexis = lexis_mult,
    aggre_by = c("sex", "age_group"),
    breaks = list(ts_fut = c(0, 5)),
    aggre_exprs = c("t_at_risk", "n_at_risk_eff")
  )
  stopifnot(
    nrow(agdt) == 2,
    agdt[["n_at_risk_eff"]] == c(2L, 1L)
  )

  # take first by lex.id
  agdt <- popEpi::lexis_split_merge_aggregate_by_stratum(
    lexis = lexis_mult,
    aggre_by = c("sex", "age_group"),
    breaks = list(ts_fut = c(0, 5)),
    aggre_exprs = c("t_at_risk", "n_at_risk_eff"),
    first_record_by = "lex.id"
  )
  stopifnot(
    nrow(agdt) == 2,
    agdt[["n_at_risk_eff"]] == c(1L, 0L)
  )

  # take first by lex.id and stratum
  agdt <- popEpi::lexis_split_merge_aggregate_by_stratum(
    lexis = lexis_mult,
    aggre_by = c("sex", "age_group"),
    breaks = list(ts_fut = c(0, 5)),
    aggre_exprs = c("t_at_risk", "n_at_risk_eff"),
    first_record_by = c("lex.id", "sex", "age_group")
  )
  stopifnot(
    nrow(agdt) == 2,

```

```

    agdt[["n_at_risk_eff"]] == c(1L, 1L),
    agdt[["t_at_risk"]] == c(1.5, 2.5)
  )

  # multipleness is handled *after* subsetting --- so we can take the first
  # record within the subset, within the stratum.
  agdt <- popEpi::lexis_split_merge_aggregate_by_stratum(
    lexis = lexis_mult,
    aggre_by = c("sex", "age_group"),
    breaks = list(ts_fut = c(0, 5)),
    aggre_exprs = c("t_at_risk", "n_at_risk_eff"),
    subset = c(FALSE, TRUE, TRUE),
    first_record_by = c("lex.id", "sex", "age_group")
  )
  stopifnot(
    nrow(agdt) == 2,
    agdt[["n_at_risk_eff"]] == c(1L, 1L),
    agdt[["t_at_risk"]] == c(4.5, 2.5)
  )

```

lexpand

*Split case-level observations***Description**

Given subject-level data, data is split by calendar time (per), age, and follow-up time (fot, from 0 to the end of follow-up) into subject-time-interval rows according to given breaks and additionally processed if requested.

Usage

```

lexpand(
  data,
  birth = NULL,
  entry = NULL,
  exit = NULL,
  event = NULL,
  status = status != 0,
  entry.status = NULL,
  breaks = list(fot = c(0, Inf)),
  id = NULL,
  overlapping = TRUE,
  aggre = NULL,
  aggre.type = c("unique", "cartesian"),
  drop = TRUE,
  pophaz = NULL,
  pp = TRUE,
  subset = NULL,

```

```

merge = TRUE,
verbose = FALSE,
...
)

```

Arguments

<code>data</code>	dataset of e.g. cancer cases as rows
<code>birth</code>	birth time in date format or fractional years; string, symbol or expression
<code>entry</code>	entry time in date format or fractional years; string, symbol or expression
<code>exit</code>	exit from follow-up time in date format or fractional years; string, symbol or expression
<code>event</code>	advanced: time of possible event differing from exit; typically only used in certain SIR/SMR calculations - see Details; string, symbol or expression
<code>status</code>	variable indicating type of event at exit or event; e.g. <code>status = status != 0</code> ; expression or quoted variable name
<code>entry.status</code>	input in the same way as <code>status</code> ; status at entry; see Details
<code>breaks</code>	a named list of vectors of time breaks; e.g. <code>breaks = list(fot=0:5, age=c(0,45,65,Inf))</code> ; see Details
<code>id</code>	optional; an id variable; e.g. <code>id = my_id</code> ; string, symbol or expression
<code>overlapping</code>	advanced, logical; if FALSE AND if data contains multiple rows per subject, ensures that the timelines of id-specific rows do not overlap; this ensures e.g. that person-years are only computed once per subject in a multi-state paradigm
<code>aggre</code>	e.g. <code>aggre = list(sex, fot)</code> ; a list of unquoted variables and/or expressions thereof, which are interpreted as factors; data events and person-years will be aggregated by the unique combinations of these; see Details
<code>aggre.type</code>	one of <code>c("unique", "cartesian")</code> ; can be abbreviated; see Details
<code>drop</code>	logical; if TRUE, drops all resulting rows after splitting that reside outside the time window as defined by the given breaks (all time scales)
<code>pophaz</code>	a dataset of population hazards to merge with split data; see Details
<code>pp</code>	logical; if TRUE, computes Pohar-Perme weights using <code>pophaz</code> ; adds variable with reserved name <code>pp</code> ; see Details for computing method
<code>subset</code>	a logical vector or any logical condition; data is subsetted before splitting accordingly
<code>merge</code>	logical; if TRUE, retains all original variables from the data
<code>verbose</code>	logical; if TRUE, the function is chatty and returns some messages along the way
<code>...</code>	e.g. <code>fot = 0:5</code> ; instead of specifying a breaks list, correctly named breaks vectors can be given for <code>fot</code> , <code>age</code> , and <code>per</code> ; these override any breaks in the breaks list; see Examples

Details

Basics

[lexpand] splits a given data set (with e.g. cancer diagnoses as rows) to subintervals of time over calendar time, age, and follow-up time with given time breaks using [splitMulti].

The dataset must contain appropriate Date / IDate format or other numeric variables that can be used as the time variables.

You may take a look at a simulated cohort [sire] as an example of the minimum required information for processing data with lexpand.

Many arguments can be supplied as a character string naming the appropriate variable (e.g. "sex"), as a symbol (e.g. sex) or as an expression (e.g. factor(sex, 0:1, c("m", "f"))) for flexibility.

Breaks

You should define all breaks as left inclusive and right exclusive time points (e.g. [a,b)) for 1-3 time dimensions so that the last member of a breaks vector is a meaningful "final upper limit", e.g. per = c(2002, 2007, 2012) to create a last subinterval of the form [2007, 2012).

All breaks are explicit, i.e. if drop = TRUE, any data beyond the outermost breaks points are dropped. If one wants to have unspecified upper / lower limits on one time scale, use Inf: e.g. breaks = list(fot = 0:5, age = c(0, 45, Inf)). Breaks for per can also be given in Date / IDateformat, whereupon they are converted to fractional years before used in splitting.

The age time scale can additionally be automatically split into common age grouping schemes by naming the scheme with an appropriate character string:

- "18of5": age groups 0-4, 5-9, 10-14, ..., 75-79, 80-84, 85+
- "20of5": age groups 0-4, 5-9, 10-14, ..., 85-89, 90-94, 95+
- "101of1": age groups 0, 1, 2, ..., 98, 99, 100+

Time variables

If any of the given time variables (birth, entry, exit, event) is in any kind of date format, they are first coerced to fractional years before splitting using [get.yrs] (with year.length = "actual").

Sometimes in e.g. SIR/SMR calculation one may want the event time to differ from the time of exit from follow-up, if the subject is still considered to be at risk of the event. If event is specified, the transition to status is moved to event from exit using [Epi::cutLexis]. See Examples.

The status variable

The statuses in the expanded output (lex.Cst and lex.Xst) are determined by using either only status or both status and entry.status. If entry.status = NULL, the status at entry is guessed according to the type of variable supplied via status: For numeric variables it will be zero, for factors the first level (levels(status)[1]) and otherwise the first unique value in alphabetical order (sort(unique(status))[1]).

Using numeric or factor status variables is strongly recommended. Logical expressions are also allowed (e.g. status = my_status != 0L) and are converted to integer internally.

Merging population hazard information

To enable computing relative/net survivals with [survtab] and [relpois], lexpand merges an appropriate population hazard data (pophaz) to the expanded data before dropping rows outside the specified time window (if drop = TRUE). pophaz must, for this reason, contain at a minimum

the variables named `agegroup`, `year`, and `haz`. `pophaz` may contain additional variables to specify different population hazard levels in different strata; e.g. `popmort` includes `sex`. All the strata-defining variables must be present in the supplied data. `lexpand` will automatically detect variables with common names in the two datasets and merge using them.

Currently `year` must be an integer variable specifying the appropriate year. `agegroup` must currently also specify one-year age groups, e.g. `popmort` specifies 101 age groups of length 1 year. In both `year` and `agegroup` variables the values are interpreted as the lower bounds of intervals (and passed on to a `cut` call). The mandatory variable `haz` must specify the appropriate average rate at the person-year level; e.g. `haz = -log(survProb)` where `survProb` is a one-year conditional survival probability will be the correct hazard specification.

The corresponding `pophaz` population hazard value is merged by using the mid points of the records after splitting as reference values. E.g. if `age=89.9` at the start of a 1-year interval, then the reference age value is 90.4 for merging. This way we get a "typical" population hazard level for each record.

Computing Pohar-Perme weights

If `pp = TRUE`, Pohar-Perme weights (the inverse of cumulative population survival) are computed. This will create the new `pp` variable in the expanded data. `pp` is a reserved name and `lexpand` throws exception if a variable with that name exists in data.

When a survival interval contains one or several rows per subject (e.g. due to splitting by the per scale), `pp` is cumulated from the beginning of the first record in a survival interval for each subject to the mid-point of the remaining time within that survival interval, and that value is given for every other record that a given person has within the same survival interval.

E.g. with 5 rows of duration 1/5 within a survival interval $[0, 1]$, `pp` is determined for all records by a cumulative population survival from 0 to 0.5. The existing accuracy is used, so that the weight is cumulated first up to the end of the second row and then over the remaining distance to the mid-point (first to 0.4, then to 0.5). This ensures that more accurately merged population hazards are fully used.

Event not at end of follow-up & overlapping time lines

`event` may be used if the event indicated by `status` should occur at a time differing from `exit`. If event is defined, `cutLexis` is used on the data set after coercing it to the `Lexis` format and before splitting. Note that some values of `event` are allowed to be `NA` as with `cutLexis` to accommodate observations without an event occurring.

Additionally, setting `overlapping = FALSE` ensures that (irrespective of using `event`) the each subject defined by `id` only has one continuous time line instead of possibly overlapping time lines if there are multiple rows in data by `id`.

Aggregating

Certain analyses such as SIR/SMR calculations require tables of events and person-years by the unique combinations (interactions) of several variables. For this, `aggre` can be specified as a list of such variables (preferably factor variables but not mandatory) and any arbitrary functions of the variables at one's disposal. E.g.

```
aggre = list(sex, agegr = cut(dg_age, 0:100))
```

would tabulate events and person-years by `sex` and an ad-hoc age group variable. Every ad-hoc-created variable should be named.

fot, per, and age are special reserved variables which, when present in the aggre list, are output as categories of the corresponding time scale variables by using e.g.

```
cut(fot, breaks$fot, right=FALSE).
```

This only works if the corresponding breaks are defined in breaks or via "...". E.g.

```
aggre = list(sex, fot.int = fot) with
breaks = list(fot=0:5).
```

The output variable fot.int in the above example will have the lower limits of the appropriate intervals as values.

aggre as a named list will output numbers of events and person-years with the given new names as categorizing variable names, e.g. `aggre = list(follow_up = fot, gender = sex, agegroup = age)`.

The output table has person-years (pyrs) and event counts (e.g. from0to1) as columns. Event counts are the numbers of transitions (`lex.Cst != lex.Xst`) or the `lex.Xst` value at a subject's last record (subject possibly defined by id).

If `aggre.type = "unique"` (alias "non-empty"), the above results are computed for existing combinations of expressions given in aggre, but also for non-existing combinations if `aggre.type = "cartesian"` (alias "full"). E.g. if a factor variable has levels "a", "b", "c" but the data is limited to only have levels "a", "b" present (more than zero rows have these level values), the former setting only computes results for "a", "b", and the latter also for "c" and any combination with other variables or expression given in aggre. In essence, "cartesian" forces also combinations of variables used in aggre that have no match in data to be shown in the result.

If aggre is not NULL and pophaz has been supplied, lexpand also aggregates the expected counts of events, which appears in the output data by the reserved name `d.exp`. Additionally, having `pp = TRUE` causes lexpand to also compute various Pohar-Perme weighted figures necessary for computing Pohar-Perme net survivals with `[survtab_ag]`. This can be slow, so consider what is really needed. The Pohar-Perme weighted figures have the suffix `.pp`.

```
[0,1]: R:0,1) [survtab_ag]: R:survtab_ag
```

Value

If `aggre = NULL`, returns a `data.table` or `data.frame` (depending on `options("popEpi.datatable")`; see `?popEpi`) object expanded to accommodate split observations with time scales as fractional years and pophaz merged in if given. Population hazard levels in new variable `pop.haz`, and Pohar-Perme weights as new variable `pp` if requested.

If `aggre` is defined, returns a long-format `data.table/data.frame` with the variable `pyrs` (person-years), and variables for the counts of transitions in state or state at end of follow-up formatted from `XtoY`, where `X` and `Y` are the states transitioned from and to, respectively. The data may also have the columns `d.exp` for expected numbers of cases and various Pohar-Perme weighted figures as identified by the suffix `.pp`; see Details.

Author(s)

Joonas Miettinen

See Also

[Epi::Lexis], [popmort]

Other splitting functions: [splitLexisDT\(\)](#), [splitMulti\(\)](#)

Other aggregation functions: [aggre\(\)](#), [as.aggre\(\)](#), [setaggre\(\)](#), [summary.aggre\(\)](#)

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)

## prepare data for e.g. 5-year cohort survival calculation
x <- lexpand(sire, breaks=list(fot=seq(0, 5, by = 1/12)),
             birth = bi_date, entry = dg_date, exit = ex_date,
             status = status != 0, pophaz=popmort)

## prepare data for e.g. 5-year "period analysis" for 2008-2012
BL <- list(fot = seq(0, 5, by = 1/12), per = c("2008-01-01", "2013-01-01"))
x <- lexpand(sire, breaks = BL,
             birth = bi_date, entry = dg_date, exit = ex_date,
             pophaz=popmort, status = status != 0)

## aggregating
BL <- list(fot = 0:5, per = c("2003-01-01", "2008-01-01", "2013-01-01"))
ag <- lexpand(sire, breaks = BL, status = status != 0,
             birth = bi_date, entry = dg_date, exit = ex_date,
             aggre=list(sex, period = per, surv.int = fot))

## aggregating even more
ag <- lexpand(sire, breaks = BL, status = status != 0,
             birth = bi_date, entry = dg_date, exit = ex_date,
             aggre=list(sex, period = per, surv.int = fot),
             pophaz = popmort, pp = TRUE)

## using "...
x <- lexpand(sire, fot=0:5, status = status != 0,
             birth = bi_date, entry = dg_date, exit = ex_date,
             pophaz=popmort)

x <- lexpand(sire, fot=0:5, status = status != 0,
             birth = bi_date, entry = dg_date, exit = ex_date,
             aggre=list(sex, surv.int = fot))

## using the "event" argument: it just places the transition to given "status"
## at the "event" time instead of at the end, if possible using cutLexis
x <- lexpand(sire, status = status, event = dg_date,
             birth = bi_date, entry = dg_date, exit = ex_date,)

## aggregating with custom "event" time
## (the transition to status is moved to the "event" time)
```

```
x <- leexpand(sire, status = status, event = dg_date,
              birth = bi_date, entry = dg_date, exit = ex_date,
              per = 1970:2014, age = c(0:100, Inf),
              aggre = list(sex, year = per, agegroup = age))
```

lines.sirspline	<i>lines method for sirspline-object</i>
-----------------	--

Description

Plot SIR spline lines with R base graphics

Usage

```
## S3 method for class 'sirspline'
lines(x, conf.int = TRUE, print.levels = NA, select.spline, ...)
```

Arguments

x	an object returned by function sirspline
conf.int	logical; default TRUE draws also the 95 confidence intervals
print.levels	name(s) to be plotted. Default plots all levels.
select.spline	select which spline variable (a number or a name) is plotted.
...	arguments passed on to lines()

Details

In lines.sirspline most of graphical parameters is user adjustable. Desired spline variable can be selected with select.spline and only one can be plotted at a time. The spline variable can include several levels, e.g. gender (these are the levels of print from sirspline). All levels are printed by default, but a specific level can be selected using argument print.levels. Printing the levels separately enables e.g. to give different colours for each level.

Value

Always returns NULL invisibly. This function is called for its side effects.

Author(s)

Matti Rantanen

See Also

Other sir functions: [plot.sirspline\(\)](#), [sir\(\)](#), [sir_exp\(\)](#), [sir_ratio\(\)](#), [sirspline\(\)](#)

`lines.survmean`*Graphically Inspect Curves Used in Mean Survival Computation*

Description

Plots the observed (with extrapolation) and expected survival curves for all strata in an object created by `[survmean]`

Usage

```
## S3 method for class 'survmean'  
lines(x, ...)
```

Arguments

<code>x</code>	a <code>survmean</code> object
<code>...</code>	arguments passed (ultimately) to <code>matlines</code> ; you may, therefore, supply e.g. <code>lwd</code> through this, though arguments such as <code>lty</code> and <code>col</code> will not work

Details

This function is intended to be a workhorse for `[plot.survmean]`. If you want finer control over the plotted curves, extract the curves from the `survmean` output using

```
attr(x, "curves")
```

where `x` is a `survmean` object.

Value

Always returns `NULL` invisibly. This function is called for its side effects.

Author(s)

Joonas Miettinen

See Also

Other `survmean` functions: [Surv\(\)](#), [plot.survmean\(\)](#), [survmean\(\)](#)

lines.survtab	lines <i>method for survtab objects</i>
---------------	---

Description

Plot lines from a survtab object

Usage

```
## S3 method for class 'survtab'
lines(x, y = NULL, subset = NULL, conf.int = TRUE, col = NULL, lty = NULL, ...)
```

Arguments

x	a survtab output object
y	a variable to plot; a quoted name of a variable in x; e.g. y = "surv.obs"; if NULL, picks last survival variable column in order in x
subset	a logical condition; obj is subset accordingly before plotting; use this for limiting to specific strata, e.g. subset = sex == "male"
conf.int	logical; if TRUE, also plots any confidence intervals present in obj for variables in y
col	line colour passed to matlines
lty	line type passed to matlines
...	additional arguments passed on to a matlines call; e.g. lwd can be defined this way

Value

Always returns NULL invisibly. This function is called for its side effects.

Author(s)

Joonas Miettinen

See Also

Other survtab functions: [Surv\(\)](#), [plot.survtab\(\)](#), [print.survtab\(\)](#), [summary.survtab\(\)](#), [survtab\(\)](#), [survtab_ag\(\)](#)

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)
```

```

data(sire)
data(sibr)
si <- rbind(sire, sibr)
si$period <- cut(si$dg_date, as.Date(c("1993-01-01", "2004-01-01", "2013-01-01")), right = FALSE)
si$cancer <- c(rep("rectal", nrow(sire)), rep("breast", nrow(sibr)))
x <- lexpand(si, birth = bi_date, entry = dg_date, exit = ex_date,
            status = status %in% 1:2,
            fot = 0:5, aggre = list(cancer, period, fot))
st <- survtab_ag(fot ~ cancer + period, data = x,
                surv.method = "lifetable", surv.type = "surv.obs")

plot(st, "surv.obs", subset = cancer == "breast", ylim = c(0.5, 1), col = "blue")
lines(st, "surv.obs", subset = cancer == "rectal", col = "red")

## or
plot(st, "surv.obs", col = c(2,2,4,4), lty = c(1, 2, 1, 2))

```

lower_bound

Return lower_bound value from char string (20,30]

Description

selects lowest values of each factor after cut() based on that the value starts from index 2 and end in comma ",".

Usage

```
lower_bound(cut)
```

Arguments

cut is a character vector of elements "(20,60]"

Value

A numeric vector.

Author(s)

Matti Rantanen

ltable	<i>Tabulate Counts and Other Functions by Multiple Variables into a Long-Format Table</i>
--------	---

Description

ltable makes use of data.table capabilities to tabulate frequencies or arbitrary functions of given variables into a long format data.table/data.frame. expr.by.cj is the equivalent for more advanced users.

Usage

```
ltable(
  data,
  by.vars = NULL,
  expr = list(obs = .N),
  subset = NULL,
  use.levels = TRUE,
  na.rm = FALSE,
  robust = TRUE
)

expr.by.cj(
  data,
  by.vars = NULL,
  expr = list(obs = .N),
  subset = NULL,
  use.levels = FALSE,
  na.rm = FALSE,
  robust = FALSE,
  .SDcols = NULL,
  enclos = parent.frame(1L),
  ...
)
```

Arguments

data	a data.table/data.frame
by.vars	names of variables that are used for categorization, as a character vector, e.g. c('sex', 'agegroup')
expr	object or a list of objects where each object is a function of a variable (see: details)
subset	a logical condition; data is limited accordingly before evaluating expr - but the result of expr is also returned as NA for levels not existing in the subset. See Examples.

<code>use.levels</code>	logical; if TRUE, uses factor levels of given variables if present; if you want e.g. counts for levels that actually have zero observations but are levels in a factor variable, use this
<code>na.rm</code>	logical; if TRUE, drops rows in table that have NA as values in any of <code>by.vars</code> columns
<code>robust</code>	logical; if TRUE, runs the output data's <code>by.vars</code> columns through <code>robust_values</code> before outputting
<code>.SDcols</code>	advanced; a character vector of column names passed to inside the <code>data.table</code> 's brackets <code>DT[, , ...]</code> ; see <code>[data.table::data.table]</code> ; if NULL, uses all appropriate columns. See Examples for usage.
<code>enclos</code>	advanced; an environment; the enclosing environment of the data.
<code>...</code>	advanced; other arguments passed to inside the <code>data.table</code> 's brackets <code>DT[, , ...]</code> ; see <code>[data.table::data.table]</code>

Details

Returns `expr` for each unique combination of given `by.vars`.

By default makes use of any and all `[levels]` present for each variable in `by.vars`. This is useful, because even if a subset of the data does not contain observations for e.g. a specific age group, those age groups are nevertheless presented in the resulting table; e.g. with the default `expr = list(obs = .N)` all age group levels are represented by a row and can have `obs = 0`.

The function differs from the vanilla `[table]` by giving a long format table of values regardless of the number of `by.vars` given. Make use of e.g. `[cast_simple]` if data needs to be presented in a wide format (e.g. a two-way table).

The rows of the long-format table are effectively Cartesian products of the levels of each variable in `by.vars`, e.g. with `by.vars = c("sex", "area")` all levels of area are repeated for both levels of sex in the table.

The `expr` allows the user to apply any function(s) on all levels defined by `by.vars`. Here are some examples:

- `.N` or `list(.N)` is a function used inside a `data.table` to calculate counts in each group
- `list(obs = .N)`, same as above but user assigned variable name
- `list(sum(obs), sum(pyr), mean(dg_age))`, multiple objects in a list
- `list(obs = sum(obs), pyr = sum(pyr))`, same as above with user defined variable names

If `use.levels = FALSE`, no levels information will be used. This means that if e.g. the agegroup variable is a factor and has 18 levels defined, but only 15 levels are present in the data, no rows for the missing levels will be shown in the table.

`na.rm` simply drops any rows from the resulting table where any of the `by.vars` values was NA.

Value

A `data.table` of statistics (e.g. counts) stratified by the columns defined in `by.vars`.

Functions

- `expr.by.cj()`: Somewhat more streamlined `ltable` with defaults for speed. Explicit determination of enclosing environment of data.

Author(s)

Joonas Miettinen, Matti Rantanen

See Also

`[table]`, `[cast_simple]`, `[data.table::melt]`

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)
data("sire", package = "popEpi")
sr <- sire
sr$agegroup <- cut(sr$dg_age, breaks=c(0,45,60,75,85,Inf))
## counts by default
ltable(sr, "agegroup")

## any expression can be given
ltable(sr, "agegroup", list(mage = mean(dg_age)))
ltable(sr, "agegroup", list(mage = mean(dg_age), vage = var(dg_age)))

## also returns levels where there are zero rows (expressions as NA)
ltable(sr, "agegroup", list(obs = .N,
                             minage = min(dg_age),
                             maxage = max(dg_age)),
       subset = dg_age < 85)

#### expr.by.cj
expr.by.cj(sr, "agegroup")

## any arbitrary expression can be given
expr.by.cj(sr, "agegroup", list(mage = mean(dg_age)))
expr.by.cj(sr, "agegroup", list(mage = mean(dg_age), vage = var(dg_age)))

## only uses levels of by.vars present in data
expr.by.cj(sr, "agegroup", list(mage = mean(dg_age), vage = var(dg_age)),
           subset = dg_age < 70)

## .SDcols trick
expr.by.cj(sr, "agegroup", lapply(.SD, mean),
           subset = dg_age < 70, .SDcols = c("dg_age", "status"))
```

meanpop_fi	<i>Mean population counts in Finland year, sex, and age group.</i>
------------	--

Description

Mean population counts in Finland year, sex, and age group.

Format

- data.table with columns
- sex gender coded as male, female (0, 1)
 - year calendar year 1981-2016
 - agegroup - coded 0 to 100; one-year age groups
 - meanpop the mean population count; that is, the mean of the annual population counts of two consecutive years; e.g. for 1990 meanpop is the mean of population counts for 1990 and 1991 (counted at 1990-01-01 and 1991-01-01, respectively)

Source

Statistics Finland

See Also

Other popEpi data: [ICSS](#), [popmort](#), [sibr](#), [sire](#), [stdpop101](#), [stdpop18](#)

na2zero	<i>Convert NA's to zero in data.table</i>
---------	---

Description

Given a data.table DT, replaces any NA values in the variables given in vars in DT. Takes a copy of the original data and returns the modified copy.

Usage

na2zero(DT, vars = NULL)

Arguments

- | | |
|------|--|
| DT | data.table object |
| vars | a character string vector of variables names in DT; if NULL, uses all variable names in DT |

Details

Given a `data.table` object, converts NA values to numeric (double) zeros for all variables named in `vars` or all variables if `vars = NULL`.

Value

A copy of DT where NA values have been replaced with zero.

Author(s)

Joonas Miettinen

plot.rate	<i>plot method for rate object</i>
-----------	------------------------------------

Description

Plot rate estimates with confidence intervals lines using R base graphics

Usage

```
## S3 method for class 'rate'
plot(x, conf.int = TRUE, eps = 0.2, left.margin, xlim, ...)
```

Arguments

<code>x</code>	a rate object (see <code>[rate]</code>)
<code>conf.int</code>	logical; default TRUE draws the confidence intervals
<code>eps</code>	is the height of the ending of the error bars
<code>left.margin</code>	set a custom left margin for long variable names. Function tries to do it by default.
<code>xlim</code>	change the x-axis location
<code>...</code>	arguments passed on to graphical functions <code>points</code> and <code>segment</code> (e.g. <code>col</code> , <code>lwd</code> , <code>pch</code> and <code>cex</code>)

Details

This is limited explanatory tool but most graphical parameters are user adjustable.

Value

Always returns NULL invisibly. This function is called for its side effects.

Author(s)

Matti Rantanen

plot.sir

*Plot method for sir-object***Description**

Plot SIR estimates with error bars

Usage

```
## S3 method for class 'sir'
plot(
  x,
  conf.int = TRUE,
  ylab,
  xlab,
  xlim,
  main,
  eps = 0.2,
  abline = TRUE,
  log = FALSE,
  left.margin,
  ...
)
```

Arguments

x	an object returned by function sir
conf.int	default TRUE draws confidence intervals
ylab	overwrites default y-axis label
xlab	overwrites default x-axis label
xlim	x-axis minimum and maximum values
main	optional plot title
eps	error bar vertical bar height (works only in 'model' or 'univariate')
abline	logical; draws a grey line in $SIR = 1$
log	logical; SIR is not in log scale by default
left.margin	adjust left marginal of the plot to fit long variable names
...	arguments passed on to plot(), segment and lines()

Details

Plot SIR estimates and confidence intervals

- univariate - plots SIR with univariate confidence intervals
- model - plots SIR with Poisson modelled confidence intervals

Customize Normal plot parameters can be passed to plot. These can be a vector when plotting error bars:

- pch - point type
- lty - line type
- col - line/point colour
- lwd - point/line size

Tips for plotting splines It's possible to use plot to first draw the confidence intervals using specific line type or colour and then plotting again the estimate using lines(..., conf.int = FALSE) with different settings. This works only when plot.type is 'splines'.

Value

Always returns NULL invisibly. This function is called for its side effects.

Author(s)

Matti Rantanen

See Also

[sir], [sirspline]

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)

# Plot SIR estimates
# plot(sir.by.gender, col = c(4,2), log=FALSE, eps=0.2, lty=1, lwd=2, pch=19,
#      main = 'SIR by gender', abline=TRUE)
```

plot.sirspline

plot *method for sirspline-object*

Description

Plot SIR splines using R base graphics.

Usage

```
## S3 method for class 'sirspline'
plot(x, conf.int = TRUE, abline = TRUE, log = FALSE, type, ylab, xlab, ...)
```

Arguments

x	an object returned by function <code>sirspline</code>
conf.int	logical; default TRUE draws also the 95 confidence intervals
abline	logical; draws a reference line where $SIR = 1$
log	logical; default FALSE. Should the y-axis be in log scale
type	select type = 'n' to plot only figure frames
ylab	overwrites default y-axis label; can be a vector if multiple splines fitted
xlab	overwrites default x-axis label; can be a vector if multiple splines fitted
...	arguments passed on to <code>plot()</code>

Details

In `plot.sirspline` almost every graphical parameter are user adjustable, such as `ylim`, `xlim`. `plot.sirsplines` calls `lines.splines` to add lines.

The plot axis without lines can be plotted using option `type = 'n'`. On top of the frame it's then possible to add a grid, `abline` or text before plotting the lines (see: `sirspline`).

Value

Always returns NULL invisibly. This function is called for its side effects.

Author(s)

Matti Rantanen

See Also

Other sir functions: [lines.sirspline\(\)](#), [sir\(\)](#), [sir_exp\(\)](#), [sir_ratio\(\)](#), [sirspline\(\)](#)

plot.survmean

Graphically Inspect Curves Used in Mean Survival Computation

Description

Plots the observed (with extrapolation) and expected survival curves for all strata in an object created by `[survmean]`

Usage

```
## S3 method for class 'survmean'
plot(x, ...)
```

Arguments

`x` a survmean object

`...` arguments passed (ultimately) to `matlines`; you may, therefore, supply e.g. `xlab` through this, though arguments such as `lty` and `col` will not work

Details

For examples see `[survmean]`. This function is intended only for graphically inspecting that the observed survival curves with extrapolation and the expected survival curves have been sensibly computed in `survmean`.

If you want finer control over the plotted curves, extract the curves from the `survmean` output using `attr(x, "curves")` where `x` is a `survmean` object.

Value

Always returns `NULL` invisibly. This function is called for its side effects.

Author(s)

Joonas Miettinen

See Also

Other `survmean` functions: [Surv\(\)](#), [lines.survmean\(\)](#), [survmean\(\)](#)

plot.survtab	plot method for survtab objects
--------------	---------------------------------

Description

Plotting for `survtab` objects

Usage

```
## S3 method for class 'survtab'
plot(
  x,
  y = NULL,
  subset = NULL,
  conf.int = TRUE,
  col = NULL,
  lty = NULL,
  ylab = NULL,
  xlab = NULL,
  ...
)
```

Arguments

<code>x</code>	a survtab output object
<code>y</code>	survival a character vector of a variable names to plot; e.g. <code>y = "r.e2"</code>
<code>subset</code>	a logical condition; obj is subset accordingly before plotting; use this for limiting to specific strata, e.g. <code>subset = sex == "male"</code>
<code>conf.int</code>	logical; if TRUE, also plots any confidence intervals present in obj for variables in <code>y</code>
<code>col</code>	line colour; one value for each stratum; will be recycled
<code>lty</code>	line type; one value for each stratum; will be recycled
<code>ylab</code>	label for Y-axis
<code>xlab</code>	label for X-axis
<code>...</code>	additional arguments passed on to <code>plot</code> and <code>lines.survtab</code> ; e.g. <code>ylim</code> can be defined this way

Value

Always returns NULL invisibly. This function is called for its side effects.

Author(s)

Joonas Miettinen

See Also

Other survtab functions: [Surv\(\)](#), [lines.survtab\(\)](#), [print.survtab\(\)](#), [summary.survtab\(\)](#), [survtab\(\)](#), [survtab_ag\(\)](#)

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)
data(sire)
data(sibr)
si <- rbind(sire, sibr)
si$period <- cut(si$dg_date, as.Date(c("1993-01-01", "2004-01-01", "2013-01-01")), right = FALSE)
si$cancer <- c(rep("rectal", nrow(sire)), rep("breast", nrow(sibr)))
x <- lexpand(si, birth = bi_date, entry = dg_date, exit = ex_date,
            status = status %in% 1:2,
            fot = 0:5, aggre = list(cancer, period, fot))
st <- survtab_ag(fot ~ cancer + period, data = x,
                surv.method = "lifetable", surv.type = "surv.obs")

plot(st, "surv.obs", subset = cancer == "breast", ylim = c(0.5, 1), col = "blue")
lines(st, "surv.obs", subset = cancer == "rectal", col = "red")
```

```
## or  
plot(st, "surv.obs", col = c(2,2,4,4), lty = c(1, 2, 1, 2))
```

poisson.ci

Get rate and exact Poisson confidence intervals

Description

Computes confidence intervals for Poisson rates

Usage

```
poisson.ci(x, pt = 1, conf.level = 0.95)
```

Arguments

x	observed
pt	expected
conf.level	alpha level

Value

A data.frame with columns

- x: arg x
- pt: arg pt
- rate: result of x / pt
- lower: lower bound of CI
- upper: upper bound of CI
- conf.level: arg conf.level

Author(s)

epitools

Examples

```
# this data.table::setDTthreads call is included here only to  
# conform to the CRAN submission requirement to only use at most 2  
# threads. you do not need to set this to use popEpi.  
# however some long calculations may benefit from using more threads.  
data.table::setDTthreads(2L)  
  
poisson.ci(x = 4, pt = 5, conf.level = 0.95)
```

Description

Several functions in **popEpi** make use of population or expected hazards in computing the intended estimates (e.g. `[survtab]`). This document explains using such data sets in this package.

Details

Population hazard data sets (pophaz for short) in **popEpi** should be `data.frames` in the "long" format where one of the columns must be named `haz` (for hazard), and other columns define the values or levels in variables relating to subjects in your data. For example, `[popmort]` contains Finnish population mortality hazards by sex, calendar year, and 1-year age group.

sex	year	agegroup	haz
0	1951	0	0.036363176
0	1951	1	0.003616547
0	1951	2	0.002172384
0	1951	3	0.001581249
0	1951	4	0.001180690
0	1951	5	0.001070595

The names of the columns should match to the names of the variables that you have in your subject-level data. Time variables in your pophaz may also correspond to `Lexis` time scales; see `[survtab]`.

Any time variables (as they usually have) should be coded consistently: When using fractional years in your data, the time variables in your pophaz must also be coded in fractional years. When using e.g. Dates in your data, ensure that the pophaz time variables are coded at the level of days (or Dates for calendar time).

The `haz` variable in your pophaz should also be coded consistently with the used time variables. E.g. `haz` values in life-tables reported as deaths per person-year should be multiplied by 365.25 when using day-level time variables. Typically you'll have calendar time and age expressed in years, which means `haz` should be expressed as the number of deaths per person-year.

If you have your population hazards in a `ratetable` object usable by functions in **survival** and **rehsurv**, you may transform them to long-format `data.frames` using `[ratetable_to_long_dt]`. Ensure, however, that the created `haz` column is coded at the right level (events per days or years typically).

National statistical institutions, the WHO, and e.g. the Human Life-Table Database supply life-table data.

Author(s)

Joonas Miettinen

popmort	<i>Population mortality rates in Finland 1951 - 2013 in 101 age groups and by gender. This is an example of a population hazard table as used in popEpi; for the general help page, see [pophaz].</i>
---------	--

Description

Population mortality rates in Finland 1951 - 2013 in 101 age groups and by gender. This is an example of a population hazard table as used in **popEpi**; for the general help page, see [pophaz].

Format

data.table with columns

- sex gender coded as male, female (0, 1)
- year calendar year
- agegroup - coded 0 to 100; one-year age groups
- haz the average population mortality rate per person-year ($d/(pyrs)$, where d is the number of deaths and pyrs is the person-years)

Source

Statistics Finland

See Also

[pophaz]

Other popEpi data: [ICSS](#), [meanpop_fi](#), [sibr](#), [sire](#), [stdpop101](#), [stdpop18](#)

prepExpo	<i>Prepare Exposure Data for Aggregation</i>
----------	--

Description

prepExpo uses a Lexis object of periods of exposure to fill gaps between the periods and overall entry and exit times without accumulating exposure time in periods of no exposure, and splits the result if requested.

Usage

```

prepExpo(
  lex,
  freezeScales = "work",
  cutScale = "per",
  entry = min(get(cutScale)),
  exit = max(get(cutScale)),
  by = "lex.id",
  breaks = NULL,
  freezeDummy = NULL,
  subset = NULL,
  verbose = FALSE,
  ...
)

```

Arguments

<code>lex</code>	a <code>[Epi::Lexis]</code> object with ONLY periods of exposure as rows; one or multiple rows per subject allowed
<code>freezeScales</code>	a character vector naming Lexis time scales of exposure which should be frozen in periods where no exposure occurs (in the gap time periods)
<code>cutScale</code>	the Lexis time scale along which the subject-specific ultimate entry and exit times are specified
<code>entry</code>	an expression; the time of entry to follow-up which may be earlier, at, or after the first time of exposure in <code>freezeScales</code> ; evaluated separately for each unique combination of <code>by</code> , so e.g. with <code>entry = min(Var1)</code> and <code>by = "lex.id"</code> it sets the <code>lex.id</code> -specific minima of <code>Var1</code> to be the original times of entry for each <code>lex.id</code>
<code>exit</code>	the same as <code>entry</code> but for the ultimate exit time per unique combination of <code>by</code>
<code>by</code>	a character vector indicating variable names in <code>lex</code> , the unique combinations of which identify separate subjects for which to fill gaps in the records from <code>entry</code> to <code>exit</code> ; for novices of <code>[data.table::data.table]</code> , this is passed to a <code>data.table</code> 's <code>by</code> argument.
<code>breaks</code>	a named list of breaks; e.g. <code>list(work = 0:20, per = 1995:2015)</code> ; passed on to <code>[splitMulti]</code> so see that function's help for more details
<code>freezeDummy</code>	a character string; specifies the name for a dummy variable that this function will create and add to output which identifies rows where the <code>freezeScales</code> are frozen and where not (0 implies not frozen, 1 implies frozen); if <code>NULL</code> , no dummy is created
<code>subset</code>	a logical condition to subset data by before computations; e.g. <code>subset = sex == "male"</code>
<code>verbose</code>	logical; if <code>TRUE</code> , the function is chatty and returns some messages and timings during its run.
<code>...</code>	additional arguments passed on to <code>[splitMulti]</code>

Details

prepExpo is a convenience function for the purpose of eventually aggregating person-time and events in categories of not only normally progressing [Epi::Lexis] time scales but also some time scales which should not progress sometimes. For example a person may work at a production facility only intermittently, meaning exposure time (to work-related substances for example) should not progress outside of periods of work. This allows for e.g. a correct aggregation of person-time and events by categories of cumulative time of exposure.

Given an [Epi::Lexis] object containing rows (time lines) where a subject is exposed to something (and NO periods without exposure), fills any gaps between exposure periods for each unique combination of by and the subject-specific "ultimate" entry and exit times, "freezes" the cumulative exposure times in periods of no exposure, and splits data using breaks passed to [splitMulti] if requested. Results in a (split) Lexis object where freezeScales do not progress in time periods where no exposure was recorded in lex.

This function assumes that entry and exit arguments are the same for each row within a unique combination of variables named in by. E.g. with by = "lex.id" only each lex.id has a unique value for entry and exit at most.

The supplied breaks split the data using splitMulti, with the exception that breaks supplied concerning any frozen time scales ONLY split the rows where the time scales are not frozen. E.g. with freezeScales = "work", breaks = list(work = 0:10, cal = 1995:2010) splits all rows over "cal" but only non-frozen rows over "work".

Only supports frozen time scales that advance and freeze contemporaneously: e.g. it would not currently be possible to take into account the cumulative time working at a facility and the cumulative time doing a single task at the facility, if the two are not exactly the same. On the other hand one might use the same time scale for different exposure types, supply them as separate rows, and identify the different exposures using a dummy variable.

Value

Returns a Lexis object that has been split if breaks is specified. The resulting time is also a data.table if options("popEpi.datatable") == TRUE (see: ?popEpi)

```
prev_functions
```

```
Prevalence
```

Description

Function(s) to compute prevalence statistics.

Usage

```
prev_lexis(
  lexis,
  observation_time_points,
  stratum_breaks,
  aggreg_by = NULL,
```

```

subset = NULL,
first_record_by = NULL,
merge_dt = NULL,
merge_optional_args = NULL,
n_at_risk_dt = NULL,
weight_dt = NULL,
conf_lvls = 0.95,
conf_methods = "log"
)

```

Arguments

lexis	[Lexis] (no default) A Lexis dataset ([Epi::Lexis] / [Lexis_dt]).
observation_time_points	[list] (no default) Prevalence observation time points. The list can have only one element, but multiple time points can be supplied, e.g. <code>list(ts_cal = c(2009.999, 2010.999))</code> . The output of this function will have these observation time points as a column with the same name as the time scale.
stratum_breaks	[list] (no default) Breaks to split lexis by. These are passed to [splitMulti]. These breaks create new strata in the output using the time scales in lexis. The last time scale used here is assumed to be the follow-up time time scale. E.g. <code>list(ts_age = seq(0, 100, 10), ts_fut = c(0, 1, 5, Inf))</code> to stratify output by age interval and time since entry interval at a given observation point.
aggre_by	Passed to [lexis_split_merge_aggregate_by_stratum].
subset	[NULL, logical, integer, data.table] (default NULL) This argument is evaluated within the context of dt and the environment where the function with this argument was called. • NULL: Implies no subsetting, i.e. retain all observations. • logical: Keep only observations where this is TRUE. Must be of length <code>nrow(dt)</code>. E.g. <code>subset = my_column == 1</code>. • integer: Keep only observations at these row numbers. E.g. <code>subset = 1:5</code>. • data.table: Each column must also be a column of dt. Only those rows in dt are retained that match a row in subset. E.g. <code>subset = data.table::data.table(my_column = 1)</code>. Due to the non-standard evaluation method used here, you can make use of data in the calling environment as well. E.g. with variable <code>my_value = 1</code> defined and dt containing column <code>my_column</code> you can do <code>subset = my_column == my_value</code> if you want.
first_record_by	Passed to [lexis_split_merge_aggregate_by_stratum].
merge_dt	[NULL, data.table] (default NULL) Table containing survival estimates applicable to lexis. These survival estimates are used in the "projection" of prevalence in those observations which

have been lost to follow-up. See **Functions** for how this works and what you need to have.

- **NULL**: No projection is performed.
- **data.table**: Must contain stratum columns also found in lexis and exactly one value column named S. This table is passed to `[lexis_merge]` and should conform to its requirements. E.g. `data.table(ts_fut = factor(c("[0, 1[", ...))), S = c(0.9, ...))`.
- **list**: We produce a table of survival estimates on the fly and these are arguments passed to `[surv_lexis]`. See **Functions**.

`merge_optional_args`

`[NULL, list]` (default **NULL**)

Each element passed to `[lexis_merge]`. E.g. `list(merge_dt_harmonisers = my_harmonisers)`.

`n_at_risk_dt` `[NULL, data.table]` (default **NULL**)

- **NULL**: No population size data is merged into the output table.
- **data.table**: Merge data from this table. See **Functions** and **Examples**.
 - It can but does not need to contain as a stratifying column the time scale used in argument `observation_time_points`. If it is included, then it must be included containing the same exact values, e.g. with `observation_time_points = list(ts_cal = 2023.9999)` you must have in `n_at_risk_dt` rows with `ts_cal = 2023.9999`.
 - It can but does not have to contain the time scales used in argument `stratum_breaks` as factor columns where the levels must contain the lower and upper bound of each interval, e.g. the levels of `ts_age` might be `c("[0,5[", "[5,10[", ..., "[80,85[", "[85,Inf[")`. The brackets must be included but their directions are ignored. The simple way to form a factor with such levels is `[cut]`. The lower and upper bounds must match the breaks used in argument `stratum_breaks`, so for the above example you should have `stratum_breaks = list(ts_cal = c(0, 5, 10, ..., 80, 85, Inf), ...)`.

`weight_dt` `[NULL, data.table]` (default **NULL**)

- **NULL**: No direct adjusting is performed.
- **data.table**: Passed to `[directadjusting::directly_adjusted_estimates]` after some harmonisation. See **Functions** and **Examples**.

`conf_lvls` Passed as-is to `[directadjusting::directly_adjusted_estimates]`

`conf_methods` Passed as-is to `[directadjusting::directly_adjusted_estimates]`

Value

popEpi::prev_lexis

Returns a `data.table` with

- Stratifying columns defined via `aggre_by` (if any),
- Stratifying time scale columns defined via `stratum_breaks` (if any),
- The observation time scale column defined via `observation_time_points`,

- `n_prev`, the number of subjects in follow-up, and
- `n_prev_eff`, the above plus the "extrapolated" number in follow-up, if this was requested.
- `n_at_risk`, the merged values from `n_at_risk_dt` if it was supplied.
- `p_prev`, `n_prev / n_at_risk`, the proportion of prevalent records out of `n_at_risk`, if `n_at_risk_dt` was supplied.
- `p_prev_eff`, `n_prev_eff / n_at_risk`, the proportion of the effective number of prevalent records out of `n_at_risk`, if `n_at_risk_dt` was supplied and `n_prev_eff` was computed.
- For `p_prev` and `p_prev_eff` the additional columns which indicate standard error and the lower and upper confidence bounds, identified with the suffix `_se`, `_lo`, and `_hi`, respectively.

Functions

popEpi::prev_lexis

`popEpi::prev_lexis` can be used to compute numbers of (potentially effective numbers of) subjects remaining in follow-up at arbitrary time points. It performs the following steps:

- For each observation time point in `observation_time_points[[1]]`:
 - Calls `[lexis_split_merge_aggregate_by_stratum]` with the `lexis`, `subset`, `aggre_by` supplied to `popEpi::prev_lexis` and with `aggre_exprs = list(n_prev = quote(.N))`, and `breaks = stratum_breaks`. This produces a table stratified by `aggre_by` and all time scales used in `stratum_breaks`. The only value column at this point is `n_prev`, the number of subjects in follow-up at the given observation time point.
 - If `!is.null(merge_dt)`, collect subjects in `lexis` who were censored before the current observation time point. This is defined as those in `lexis` who have `lexis[["lex.Cst"]] == lexis[["lex.Xst"]]` and `lexis[[obs_ts_col_nm]] + lexis[["lex.dur"]] < obs_tp`, where `obs_ts_col_nm = names(observation_time_points)` and `obs_tp` is the current observation time point. These collected subjects are the ones we need to extrapolate to the current observation time point.
 - If there are no observations that need to be extrapolated then we end the extrapolation step early and simply use `n_prev_eff = n_prev`. Otherwise proceed as described below.
 - If `inherits(merge_dt, "list")`, collect arguments from `merge_dt` to call `[surv_lexis]`.
 - However, arguments `lexis`, `subset`, and `estimators = "S_ch"` are assigned internally to the respective arguments supplied to `prev_lexis` and cannot be overridden.
 - Note that `lexis` is the input argument `lexis` and not only those whose follow-up we need to "extrapolate". So estimation of survival is based on all subjects (in the `subset`).
 - We set as overrideable defaults
 - * `aggre_by` as-is,
 - * `breaks` as monthly intervals up to 5 years of follow-up and thereafter as annual intervals. Note that this default only works correctly if your time scales are specified in fractional years so that e.g. `lex.dur = 5.1` means 5.1 years of follow-up. The last interval ends where the longest "extrapolation" ends, e.g. for observation time point 2021.99 and the earliest extrapolatable observation starting at `ts_cal = 1995.53` the last interval is `[26, 27]`. From `ts_cal = 2020.13` the last interval is `[1.83333, 1.91667]`.

- * `split_merge_aggregate_optional_args` contains `element_breaks_collapse_args` with value `list(test_expr = quote(sum(lex.dur)))`. Therefore by default we re-aggregate survival intervals to avoid empty ones.
- * `surv_estimate_optional_args` as `list(empty_interval_action = "surv_collapse_ts_1d")`.
- If `inherits(merge_dt, "list")`, we also interpolate survival estimates in the table we have just produced so that there are always 1000 intervals starting from zero and ending where the last survival estimate is available. This is performed to ensure that the survival estimates we merge will be available at a sufficient "resolution". For instance, even if the survival estimates are for one-year intervals due to sparsity, we get a reasonable survival estimate for a subject who was censored at 1.01 — from somewhere close to 1.01 instead from 2.0.
- If you passed `breaks` via `merge_dt` with multiple time scales (e.g. for period analysis), the time scales other than the survival time one are removed from the table of survival estimates if the time scale only defined one interval. For instance if you used `ts_cal = c(2020, 2022.9999)` then the table of survival estimates contains only one period and we discard the `ts_cal_start` and `ts_cal_stop` columns. This is to enable computing e.g. period analysis survival and then allowing those results to be applied to observations outside of the period. Otherwise e.g. period analysis survival estimates for `ts_cal = c(2020, 2022.9999)` would not be merged to an extrapolatable observation at `ts_cal = 2019.5`. But if multiple intervals are defined then those are kept, e.g. `ts_cal = c(2020, 2023, 2026)` leads to the period estimates 2020–2022 and 2023–2025. If you do period analysis here, you must make sure to only use one period or ensure that the estimated multiple periods cover all the extrapolatable observations.
- Merge (for the first time) `merge_dt` with the collected subjects at the original exit time of each subject. This yields the survival probability for each subject at exit, in math we name this $S(t_e)$. E.g. subject was censored at $t_e = 2.1$ means $S(t_e) = S(2.1)$.
- Merge `merge_dt` for the second time, this time at the current prevalence observation time point such as at `ts_cal = 2009.999`. In math this is $S(t_p)$ where t_p is the prevalence observation time point. For instance for one subject t_p might be $t_p = 5.7$ because they were censored 5.7 years before the prevalence observation time point. Then we use $S(t_p) = S(5.7)$.
- With both $S(t_e)$ and $S(t_p)$ available, our "extrapolated" or "effective" for one subject for being in follow-up is between zero and one for each subject and defined simply as the conditional survival up to t_p starting from t_e , $S(t_p|t_e) = S(t_p) / S(t_e)$. E.g. $S(t_p) / S(t_e) = S(5.7) / S(2.1) = 0.8 / 0.9 \sim 0.8888889$.
- Call `[lexis_split_merge_aggregate_by_stratum]` for the second time, this time with the subjects collected for extrapolation, and sum the number of extrapolated subjects in follow-up into a table with the identical stratification as the one created before.
- Add column `n_prev_eff` as the sum of the number of extrapolated subjects and the original `n_prev` into the first table we created.
- Add column `n_cens` as the number of records which we had to extrapolate for.
- Collect the observation time point-specific results into one big table.
- If supplied, merge data from `n_at_risk_dt` into the big table. To accomplish this, any stratifying time scale columns defined via `stratum_breaks` and found also in `n_at_risk_dt` (the rather usual one being age, e.g. `ts_age`) is assumed to be a factor column with levels that specify the lower and upper bound of each interval in brackets, e.g. `[0, 5[`, though the brackets are ignored. The lower and upper bounds are made use of when merging the data from

`n_at_risk_dt` into the big table, as the latter contains the lower and upper bounds as separate columns. E.g. column `ts_age` in `n_at_risk_dt` with levels `c("[0,5[", "[5,10[", ..., "[80,85[", "[85,Inf[")` is turned into the two columns `ts_age_start = c(0, 5, 10, ..., 80, 85)` and `ts_age_stop = c(5, 10, 15, ..., 85, Inf)`. The big output table contains the very same columns if one has supplied something like `stratum_breaks = list(ts_age = c(0, 5, 10, ..., 80, 85, Inf), ...)`. Then the merge is straightforward and the big table gains the additional column `n_at_risk`.

- If `weight_dt` was supplied, we harmonise it with the current big output table. Similarly to `n_at_risk_dt`, it is possible to include time scales from `stratum_breaks` in `weight_dt` as a factor with levels which are bracketed intervals, e.g. `c("[0,5[", "[5,10[", ..., "[80,85[", "[85,Inf[")`, following the format produced by `[cut]`. The lower and upper bounds are extracted and matched with the corresponding columns in the current big output table, e.g. with `ts_age_start` and `ts_age_stop`.
- `[directadjusting::directly_adjusted_estimates]` is called to produce adjusted estimates of prevalence proportions if `weight_dt` was supplied or non-adjusted ones of not. Both `p_prev` and `p_prev_eff` are passed to `[directadjusting::directly_adjusted_estimates]`, if the latter was estimated.
- Even if adjusting is performed, the output table will contain "raw" sums of `n_at_risk`, `n_prev`, and `n_prev_eff` if it was computed.

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)

# popEpi::prev_lexis
make_sire <- function() {
  sire <- popEpi::sire
  sire <- sire[
    sire[["dg_date"]] < sire[["ex_date"]] &
    sire[["ex_date"]] >= as.Date("1999-01-01") &
    (get.yrs(sire[["ex_date"]]) - get.yrs(sire[["bi_date"]])) < 100
  ]
  # censor a few cases for testing
  set.seed(1337)
  censor_idx <- sample(
    which(sire[["ex_date"]] == max(sire[["ex_date"]])),
    size = 100
  )
  sire[
    i = censor_idx,
    j = "ex_date" := dg_date + as.integer((ex_date - dg_date) * runif(n = .N))
  ]
  # you can also use popEpi::Lexis_dt
  sire <- Epi::Lexis(
    data = sire,
    entry = list(
```

```

        ts_cal = popEpi::get.yrs(dg_date),
        ts_age = popEpi::get.yrs(dg_date) - popEpi::get.yrs(bi_date),
        ts_fut = 0.0
    ),
    duration = popEpi::get.yrs(ex_date) - popEpi::get.yrs(dg_date),
    entry.status = 0L,
    exit.status = status
  )
  return(sire[])
}
sire <- make_sire()

# observed counts only
pdt <- popEpi::prev_lexis(
  lexis = sire,
  aggre_by = "sex",
  observation_time_points = list(ts_cal = 2011.99),
  stratum_breaks = list(
    ts_age = c(seq(0, 85, 5), Inf),
    ts_fut = c(0, 1, 5, 10, Inf)
  )
)
stopifnot(
  inherits(pdt, "data.table"),
  "sex" %in% names(pdt),
  "n_prev" %in% names(pdt)
)

# + effective counts with survival table for computing expected counts
sdt <- popEpi::surv_lexis(
  lexis = sire,
  breaks = list(
    # it is best to have smaller intervals initially but they can be rather
    # wide after a while
    ts_fut = c(seq(0, 1, 1 / 12), seq(2, 50, 1))
  ),
  aggre_by = "sex",
  first_record_by = "lex.id",
  split_merge_aggregate_optional_args = list(
    # we allow for collapsing intervals to ensure none are totally empty
    breaks_collapse_args = list(
      test_expr = quote(sum(lex.dur))
    )
  )
)
# sdt goes ultimately to popEpi::lexis_merge, so we massage it a bit first
data.table::set(
  x = sdt,
  j = "ts_fut",
  value = data.table::fctr(paste0(
    "]",
    sdt[["ts_fut_start"]],
    ",",

```

```

      sdt[["ts_fut_stop"]],
      "]"
    ))
  )
  data.table::setnames(x = sdt, "S_ch", "S")
  sdt <- data.table::setDT(as.list(sdt)[c(
    "sex",
    "ts_fut",
    "S"
  )])
  pdt <- popEpi::prev_lexis(
    lexis = sire,
    aggre_by = "sex",
    observation_time_points = list(ts_cal = 2011.99),
    stratum_breaks = list(
      ts_age = c(seq(0, 85, 5), Inf),
      ts_fut = c(0, 1, 5, 10, Inf)
    ),
    merge_dt = sdt
  )
  stopifnot(
    inherits(pdt, "data.table"),
    "sex" %in% names(pdt),
    c("n_prev", "n_prev_eff", "n_cens") %in% names(pdt),
    !is.na(pdt[["n_prev_eff"]]),
    pdt[["n_prev_eff"]] <= pdt[["n_prev"]] + pdt[["n_cens"]]
  )

  # with survival estimated on the fly
  surv_lexis_arg_list <- list(
    breaks = list(
      # optional period analysis estimation approach
      ts_cal = c(2001.99, 2011.99),
      # it is best to have smaller intervals initially but they can be rather
      # wide after a while
      ts_fut = c(seq(0, 1, 1 / 12), seq(2, 50, 1))
    ),
    split_merge_aggregate_optional_args = list(
      # we allow for collapsing intervals to ensure none are totally empty
      breaks_collapse_args = list(
        test_expr = quote(sum(lex.dur))
      )
    ),
    estimators = "S_ch"
  )
  pdt <- popEpi::prev_lexis(
    lexis = sire,
    aggre_by = "sex",
    observation_time_points = list(ts_cal = 2011.99),
    stratum_breaks = list(
      ts_age = c(seq(0, 85, 5), Inf),
      ts_fut = c(0, 1, 5, 10, Inf)
    ),

```

```

    merge_dt = surv_lexis_arg_list
  )
  stopifnot(
    inherits(pdt, "data.table"),
    "sex" %in% names(pdt),
    c("n_prev", "n_prev_eff", "n_cens") %in% names(pdt),
    pdt[["n_prev_eff"]] <= pdt[["n_prev"]] + pdt[["n_cens"]]
  )

  # + prevalence proportions
  # first: this is the finnish population size on 2011-12-31
  my_n_at_risk_dt <- data.table::data.table(
    sex = 1L,
    ts_age = data.table::fctr(paste0(
      "[",
      seq(0, 85, 5),
      ",",
      c(seq(5, 85, 5), Inf),
      "]"
    )),
    ts_cal = 2011.99,
    n_at_risk = c(
      148096L,
      143394L,
      143320L,
      160535L,
      162625L,
      167897L,
      165536L,
      155863L,
      166939L,
      185618L,
      185742L,
      193930L,
      201272L,
      154642L,
      130644L,
      104590L,
      91239L,
      86851L
    )
  )
  pdt <- popEpi::prev_lexis(
    lexis = sire,
    aggre_by = "sex",
    observation_time_points = list(ts_cal = 2011.99),
    stratum_breaks = list(
      ts_age = c(seq(0, 85, 5), Inf),
      ts_fut = c(0, 1, 5, 10, Inf)
    ),
    merge_dt = surv_lexis_arg_list,
    n_at_risk_dt = my_n_at_risk_dt
  )

```

```

stopifnot(
  inherits(pdt, "data.table"),
  "sex" %in% names(pdt),
  c("n_prev", "n_prev_eff", "n_cens") %in% names(pdt),
  paste0("p_prev", c("", "_se", "_lo", "_hi")) %in% names(pdt),
  paste0("p_prev_eff", c("", "_se", "_lo", "_hi")) %in% names(pdt),
  pdt[["n_prev_eff"]] <= pdt[["n_prev"]] + pdt[["n_cens"]]
)

# same without the effective numbers
pdt <- popEpi::prev_lexis(
  lexis = sire,
  aggre_by = "sex",
  observation_time_points = list(ts_cal = 2011.99),
  stratum_breaks = list(
    ts_age = c(seq(0, 85, 5), Inf),
    ts_fut = c(0, 1, 5, 10, Inf)
  ),
  n_at_risk_dt = my_n_at_risk_dt
)
stopifnot(
  inherits(pdt, "data.table"),
  "sex" %in% names(pdt),
  "n_prev" %in% names(pdt),
  paste0("p_prev", c("", "_se", "_lo", "_hi")) %in% names(pdt),
  nrow(pdt) == 72
)

# + direct adjusting
# first make a table of weights for direct adjusting.
# this is the finnish population on 2023-12-31.
wdt <- data.table::data.table(
  ts_age = data.table::fctr(paste0(
    "[",
    seq(0, 85, 5),
    ",",
    c(seq(5, 85, 5), Inf),
    "]"
  )),
  weight_fi_2023 = c(
    115211L,
    136012L,
    155573L,
    153424L,
    148987L,
    165738L,
    182972L,
    178335L,
    177751L,
    169327L,
    155810L,
    180155L,
    182260L,

```

```

    179869L,
    178732L,
    161722L,
    99450L,
    108625L
  )
)
pdt <- popEpi::prev_lexis(
  lexis = sire,
  aggre_by = "sex",
  observation_time_points = list(ts_cal = 2011.99),
  stratum_breaks = list(
    ts_age = c(seq(0, 85, 5), Inf),
    ts_fut = c(0, 1, 5, 10, Inf)
  ),
  n_at_risk_dt = my_n_at_risk_dt,
  weight_dt = wdt
)
stopifnot(
  inherits(pdt, "data.table"),
  "sex" %in% names(pdt),
  "n_prev" %in% names(pdt),
  paste0("p_prev_fi_2023", c("", "_se", "_lo", "_hi")) %in% names(pdt),
  nrow(pdt) == 4
)

# + effective numbers
pdt <- popEpi::prev_lexis(
  lexis = sire,
  aggre_by = "sex",
  observation_time_points = list(ts_cal = 2011.99),
  stratum_breaks = list(
    ts_age = c(seq(0, 85, 5), Inf),
    ts_fut = c(0, 1, 5, 10, Inf)
  ),
  merge_dt = surv_lexis_arg_list,
  n_at_risk_dt = my_n_at_risk_dt,
  weight_dt = wdt
)
stopifnot(
  inherits(pdt, "data.table"),
  "sex" %in% names(pdt),
  "n_prev" %in% names(pdt),
  paste0("p_prev_fi_2023", c("", "_se", "_lo", "_hi")) %in% names(pdt),
  paste0("p_prev_eff_fi_2023", c("", "_se", "_lo", "_hi")) %in% names(pdt),
  nrow(pdt) == 4
)

# + multiple weights
data.table::set(
  x = wdt,
  j = c("weight", "weight_eu_1976"),

```

```

value = list(
  runif(18),
  c(
    # from Waterhouse et al. 1976
    8000L,
    rep(7000L, 10L),
    6000L,
    5000L,
    4000L,
    3000L,
    2000L,
    1000L,
    1000L
  )
)
)
)
pdt <- popEpi::prev_lexis(
  lexis = sire,
  aggre_by = "sex",
  observation_time_points = list(ts_cal = 2011.99),
  stratum_breaks = list(
    ts_age = c(seq(0, 85, 5), Inf),
    ts_fut = c(0, 1, 5, 10, Inf)
  ),
  merge_dt = surv_lexis_arg_list,
  n_at_risk_dt = my_n_at_risk_dt,
  weight_dt = wdt
)
stopifnot(
  inherits(pdt, "data.table"),
  "sex" %in% names(pdt),
  "n_prev" %in% names(pdt),
  paste0("p_prev_fi_2023", c("", "_se", "_lo", "_hi")) %in% names(pdt),
  paste0("p_prev_eff_fi_2023", c("", "_se", "_lo", "_hi")) %in% names(pdt),
  paste0("p_prev_eu_1976", c("", "_se", "_lo", "_hi")) %in% names(pdt),
  paste0("p_prev_eff_eu_1976", c("", "_se", "_lo", "_hi")) %in% names(pdt),
  # based on `wdt$weight`
  paste0("p_prev_da", c("", "_se", "_lo", "_hi")) %in% names(pdt),
  paste0("p_prev_eff_da", c("", "_se", "_lo", "_hi")) %in% names(pdt),
  nrow(pdt) == 4
)

```

print.aggre

Print an aggre Object

Description

Print method function for aggre objects; see [as.aggre] and [aggre].

Usage

```
## S3 method for class 'aggre'
print(x, subset = NULL, ...)
```

Arguments

x an aggre object

subset a logical condition to subset results table by before printing; use this to limit to a certain stratum. E.g. `subset = sex == "male"`

... arguments passed to `print.data.table`; try e.g. `top = 2` for numbers of rows in head and tail printed if the table is large, `nrow = 100` for number of rows to print, etc.

Value

Always returns NULL invisibly. This function is called for its side effects.

Author(s)

Joonas Miettinen

print.rate	<i>Print an rate object</i>
------------	-----------------------------

Description

Print method function for rate objects; see `[rate]`.

Usage

```
## S3 method for class 'rate'
print(x, subset = NULL, ...)
```

Arguments

x an rate object

subset a logical condition to subset results table by before printing; use this to limit to a certain stratum. E.g. `subset = sex == "female"`

... arguments for `data.tables` print method, e.g. `row.names = FALSE` suppresses row numbers.

Value

Always returns NULL invisibly. This function is called for its side effects.

Author(s)

Matti Rantanen

print.survtab	<i>Print a survtab Object</i>
---------------	-------------------------------

Description

Print method function for survtab objects; see [survtab_ag].

Usage

```
## S3 method for class 'survtab'  
print(x, subset = NULL, ...)
```

Arguments

- x a survtab object
- subset a logical condition to subset results table by before printing; use this to limit to a certain stratum. E.g. subset = sex == "male"
- ... arguments passed to print.data.table; try e.g. top = 2 for numbers of rows in head and tail printed if the table is large, nrow = 100 for number of rows to print, etc.

Value

Always returns NULL invisibly. This function is called for its side effects.

Author(s)

Joonas Miettinen

See Also

Other survtab functions: [Surv\(\)](#), [lines.survtab\(\)](#), [plot.survtab\(\)](#), [summary.survtab\(\)](#), [survtab\(\)](#), [survtab_ag\(\)](#)

rate	<i>Direct-Standardised Incidence/Mortality Rates</i>
------	--

Description

rate calculates adjusted rates using preloaded weights data or user specified weights.

Usage

```
rate(
  data,
  obs = NULL,
  pyrs = NULL,
  print = NULL,
  adjust = NULL,
  weights = NULL,
  subset = NULL
)
```

Arguments

data	aggregated data (see e.g. <code>[lexpand]</code> , <code>[aggre]</code> if you have subject-level data)
obs	observations variable name in data. Flexible input , typically e.g. <code>obs = obs</code> .
pyrs	person-years variable name in data. Flexible input , typically e.g. <code>pyrs = pyrs</code> .
print	variable name to stratify the rates. Flexible input , typically e.g. <code>print = sex</code> or <code>print = list(sex, area)</code> .
adjust	variable for adjusting the rates. Flexible input , typically e.g. <code>adjust = agegroup</code> .
weights	typically a list of weights or a character string specifying an age group standardization scheme; see the dedicated help page and examples.
subset	a logical expression to subset data.

Details

Input data needs to be in aggregated format with observations and person-years. For individual data use `[lexpand]`, or `[ltable]` and merge person-years manually.

The confidence intervals are based on the normal approximation of the logarithm of the rate. The variance of the log rate that is used to derive the confidence intervals is derived using the delta method.

Value

Returns a `data.table` with observations, person-years, rates and adjusted rates, if available. Results are stratified by `print`. Adjusted rates are identified with suffix `.adj` and `.lo` and `.hi` are for confidence intervals lower and upper 95% bounds, respectively. The prefix `SE.` stands for standard error.

Author(s)

Matti Rantanen, Joonas Miettinen

See Also

`[lexpand]`, `[ltable]`

Other main functions: [Surv\(\)](#), [relpois\(\)](#), [relpois_ag\(\)](#), [sir\(\)](#), [sirspline\(\)](#), [survmean\(\)](#), [survtab\(\)](#), [survtab_ag\(\)](#)

Other rate functions: [rate_ratio\(\)](#)

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)
## Prepare data with lexpand and then reformat agegroup.
data(sibr)
x <- lexpand(sibr, birth = bi_date, entry = dg_date, exit = ex_date,
             breaks = list(per = c(1990,2000,2010,2020), age = c(0:17*5,Inf)),
             aggre = list(agegroup = age, year.cat = per),
             status = status != 0)

x$agegroup <- cut(x$agegroup, c(0:17*5,Inf), right = FALSE)

## calculate rates for selected periods with Nordic 2000 weights:
r1 <- rate( data = x, obs = from0to1, pyrs = pyrs, print = year.cat,
            adjust = agegroup, weights = 'nordic')

r1

## use total person-years by stratum as weights (some have zero)
w <- ltable(x, by.vars = "agegroup", expr = sum(pyrs))
w[is.na(w$V1),]$V1 <- 0

r2 <- rate( data = x, obs = from0to1, pyrs = pyrs, print = year.cat,
            adjust = agegroup,
            weights = w$V1)

r2

## use data.frame of weights:
names(w) <- c("agegroup", "weights")
r2 <- rate( data = x, obs = from0to1, pyrs = pyrs, print = year.cat,
            adjust = agegroup,
            weights = w)

r2

## internal weights (same result as above)
r3 <- rate( data = x, obs = from0to1, pyrs = pyrs, print = year.cat,
            adjust = agegroup,
            weights = "internal")

r3
```

rate_estimate

Rate Estimates

Description

Produce estimates of rates, optionally directly adjusted.

Usage

```
rate_estimate(
  dt,
  stratum_col_nms,
  count_col_nms,
  t_at_risk_dt,
  weight_dt = NULL,
  conf_lvls = 0.95,
  conf_methods = "identity"
)
```

Arguments

<code>dt</code>	[data.frame, data.table] (no default) A data.frame or data.table containing columns of counts (count_col_nms) by stratifying columns stratum_col_nms. This object must contain every possible stratum combination of your stratifying columns. For instance every age group must be included in dt even if their count column values are zero. This is necessary for the correct merging of t_at_risk and direct adjusting.
<code>stratum_col_nms</code>	[character] (no default) Names of stratifying columns in dt. E.g. c("sex", "age_group", "year").
<code>count_col_nms</code>	[character] (no default) Names of count columns in dt. E.g. c("n_cancer", "n_dead").
<code>t_at_risk_dt</code>	[data.table] (no default) data.table object containing exactly one value column "t_at_risk" and one or more stratifying columns.
<code>weight_dt</code>	[NULL, data.table] (default NULL) Optional data.table of direct adjusting weights passed to [directadjusting::directly_adjusted_e]. The stratifying columns of this table must be found also in dt and will be used for adjusting. E.g. data.table::data.table(age_group = 1:18, weight = some_values).
<code>conf_lvls</code>	Passed to [directadjusting::directly_adjusted_estimates].
<code>conf_methods</code>	Passed to [directadjusting::directly_adjusted_estimates].

Details

popEpi::rate_estimate performs the following steps:

- Add column t_at_risk from t_at_risk_dt to (copy of) dt.
- The name of the rate columns will usually be paste0("r_", count_col_nms), e.g. r_my_count_column. But for those count columns that start with n_, this will be replaced, e.g. n_cancer -> r_cancer. Also, n -> r. The standard error column names simply have the suffix _se at the end, e.g. r_cancer_se.
- Compute a rate for each count_col_nms element as $dt[[count_col_nm]] / dt[["t_at_risk"]]$.
- Compute the variance of the rate as the rate divided by $dt[["t_at_risk"]]$. The formula is $n / (t_at_risk^2) = r / t_at_risk$.

- Call `[directadjusting::directly_adjusted_estimates]`. Note that direct adjusting is not performed if `weight_dt = NULL`.
- Turn the variance columns produced by `[directadjusting::directly_adjusted_estimates]` into standard error columns using `sqrt`.
- Add the attribute `directly_adjusted_estimates_meta` from the output of the `[directadjusting::directly_adju]` into the output of this function.
- Return a `data.table` with columns of strata, counts, rates, and `t_at_risk`.

Value

Returns a `data.table` with columns of strata, counts, rates, and `t_at_risk`.

Examples

```
# popEpi::rate_estimate

# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)

dt <- data.table::CJ(
  cancer_type = c("brain", "lung"),
  sex = 0:1,
  year = 2001:2002,
  age_group = 1:18
)
dt[j = "n" := rpois(n = .N, lambda = 100)]

tar_dt <- data.table::copy(dt)
tar_dt[j = "n" := n * runif(n = .N, min = 10.0, max = 100.0)]
data.table::setnames(tar_dt, "n", "t_at_risk")

rdt_1 <- popEpi::rate_estimate(
  dt = dt,
  stratum_col_nms = c("cancer_type", "sex", "year", "age_group"),
  count_col_nms = "n",
  t_at_risk_dt = tar_dt,
  weight_dt = NULL
)
stopifnot(
  c("cancer_type", "sex", "year", "age_group") %in% names(rdt_1),
  "t_at_risk" %in% names(rdt_1)
)

# world standard population, 2000 edition and 1966 edition, in 5-year age groups
wdt_world <- data.table::data.table(
  age_group = 1:18,
  weight_world_1966 = c(
    0.12,
```

```

    0.1,
    0.09,
    0.09,
    0.08,
    0.08,
    0.06,
    0.06,
    0.06,
    0.06,
    0.05,
    0.04,
    0.04,
    0.03,
    0.02,
    0.01,
    0.005,
    0.005
  ),
  weight_world_2000 = c(
    0.088569,
    0.08687,
    0.08597,
    0.08467,
    0.082171,
    0.079272,
    0.076073,
    0.071475,
    0.065877,
    0.060379,
    0.053681,
    0.045484,
    0.037187,
    0.02959,
    0.022092,
    0.015195,
    0.009097,
    0.006348
  )
)
)

# one weight column only
rdt_2 <- popEpi::rate_estimate(
  dt = dt,
  stratum_col_nms = c("cancer_type", "sex", "year", "age_group"),
  count_col_nms = "n",
  t_at_risk_dt = tar_dt,
  weight_dt = data.table::data.table(
    age_group = wdt_world[["age_group"]],
    weight = wdt_world[["weight_world_2000"]]
  )
)
)
stopifnot(
  c("cancer_type", "sex", "year") %in% names(rdt_2),

```

```

    !"age_group" %in% names(rdt_2),
    "t_at_risk" %in% names(rdt_2)
  )

  # multiple weight columns
  rdt_3 <- popEpi::rate_estimate(
    dt = dt,
    stratum_col_nms = c("cancer_type", "sex", "year", "age_group"),
    count_col_nms = "n",
    t_at_risk_dt = tar_dt,
    weight_dt = wdt_world
  )
  stopifnot(
    c("cancer_type", "sex", "year") %in% names(rdt_3),
    !"age_group" %in% names(rdt_3),
    "t_at_risk" %in% names(rdt_3)
  )

```

rate_ratio

Confidence intervals for the rate ratios

Description

Estimate the rate ratio.

Usage

```
rate_ratio(x, y, crude = FALSE, SE.method = FALSE)
```

Arguments

x	[rate, integer, numeric, list] (no default) Rate data. - rate: An object as produced by [rate]. If one of x and y is a rate object then both have to be or an error is raised. - integer / numeric: A vector of length two. For what the elements should be, see the documentation for argument SE.method. - list: A list of length two. Each element is a vector of of numbers, whose intended contents depend on argument SE.method.
y	[rate, integer, numeric] (no default) See the documentation for argument x.
crude	[logical] (default NULL) This argument only used when x and y are rate objects as produced by [rate]. - TRUE causes column rate to be used from the output object of [rate]. - FALSE causes column rate.adj to be used.

- SE.method [logical, character] (default FALSE)
- Ignored when x and y are objects of class rate as produced by [rate].
- FALSE / "poisson.test": The first element of xandy is the number of events and the second the standard error. Confidence intervals are produced using the poisson.test method with the log-transform.
 - TRUE / "se_delta_method": The first element is the rate estimate and second the standard error. Confidence intervals are produced using the delta method with the log-transform.
 - SE.method = "glm": x and y contain event numbers and at-risk times. We fit a tiny Poisson regression model using [stats::glm] to compute exact confidence intervals. This is the most efficient and accurate method.

Details

rate_ratio was originally written for the purpose of estimating the rate ratio when only the rates and their standard errors are known (SE.method = TRUE). For instance one could estimate the rate ratio of two age-adjusted rates in this manner. However, we have also implemented the possibility of estimating the rate ratio from count data.

Value

A vector length of three: rate_ratio, and lower and upper confidence intervals.

Author(s)

Matti Rantanen

See Also

[rate]

Other rate functions: [rate\(\)](#)

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)

# two rate ratios; silly example with female rectal / breast cancer
## mortality rates
data("sire", package = "popEpi")
data("sibr", package = "popEpi")

BL <- list(per = 2000:2005)

re <- lexpand(sire, birth = "bi_date", entry = "dg_date", exit = "ex_date",
             status = status == 1, breaks = BL, aggre = list(per))
br <- lexpand(sibr, birth = "bi_date", entry = "dg_date", exit = "ex_date",
             status = status == 1, breaks = BL, aggre = list(per))
```

```

r_re <- rate(re, obs = "from0to1", pyrs = "pyrs")
r_br <- rate(br, obs = "from0to1", pyrs = "pyrs")

rate_ratio(r_re, r_br, crude = TRUE, SE.method = TRUE)

# using rates (0.003 and 0.005) and their SEs (0.001 and 0.002)
# the CIs come from the delta method
rate_ratio(x = c(0.003, 0.001), y = c(0.005, 0.002),
           SE.method = TRUE)
rate_ratio(x = c(0.003, 0.001), y = c(0.005, 0.002),
           SE.method = "se_delta_method")
rate_ratio(
  x = list(c(0.003, 0.004), c(0.001, 0.001)),
  y = list(c(0.005, 0.006), c(0.002, 0.002)),
  SE.method = "se_delta_method"
)

# using event numbers (10 and 20) and person-years (30000 and 40000)
rate_ratio(x = c(10, 30000), y = c(20, 40000), SE.method = FALSE)
rate_ratio(x = c(10, 30000), y = c(20, 40000), SE.method = "poisson.test")
rate_ratio(
  x = list(c(10, 20), c(30000, 30000)),
  y = list(c(20, 30), c(40000, 40000)),
  SE.method = "poisson.test"
)

# using event numbers (10 and 20) and person-years (30000 and 40000)
# and exact confidence intervals
rate_ratio(x = c(10, 30000), y = c(20, 40000), SE.method = "glm")
rate_ratio(
  x = list(c(10, 20), c(30000, 30000)),
  y = list(c(20, 30), c(40000, 40000)),
  SE.method = "glm"
)

```

relpois

Excess hazard Poisson model

Description

Estimate a Poisson piecewise constant excess hazards model

Usage

```
relpois(data, formula, fot.breaks = NULL, subset = NULL, check = TRUE, ...)
```

Arguments

<code>data</code>	a dataset split with e.g. <code>[lexpand]</code> ; must have expected hazard merged within
<code>formula</code>	a formula which is passed on to <code>glm</code> ; see Details
<code>fot.breaks</code>	optional; a numeric vector of <code>[a,b)</code> breaks to specify survival intervals over the follow-up time; if <code>NULL</code> , the existing breaks along the mandatory <code>fot</code> time scale in data are used (e.g. the breaks for <code>fot</code> supplied to <code>lexpand</code>)
<code>subset</code>	a logical vector or condition; e.g. <code>subset = sex == 1</code> ; limits the data before estimation
<code>check</code>	logical; if <code>TRUE</code> , tabulates excess cases by all factor variables in the formula to check for negative / NA excess cases before fitting the GLM
<code>...</code>	any argument passed on to <code>glm</code>

Details**Basics**

`relpois` employs a custom link function of the Poisson variety to estimate piecewise constant parametric excess hazards. The pieces are determined by `fot.breaks`. A `log(person-years)` offset is passed automatically to the `glm` call.

Formula usage

The formula can be used like any ordinary `glm` formula. The user must define the outcome in some manner, which is usually `lex.Xst` after splitting with e.g. `lexpand`. The exception is the possibility of including the baseline excess hazard terms by including the reserved term `FOT` in the formula.

For example, `lex.Xst != 0 ~ FOT + agegr` estimates a model with constant excess hazards at the follow-up intervals as specified by the pertinent breaks used in splitting data, as well as for the different age groups. `FOT` is created ad hoc if it is used in the formula. If you leave out `FOT`, the hazard is effectively assumed to be constant across the whole follow-up time.

You can also simply use your own follow-up time interval variable that you have created before calling `relpois`. However, when using `FOT`, `relpois` automatically checks for e.g. negative excess cases in follow-up intervals, allowing for quickly finding splitting breaks where model estimation is possible. It also drops any data outside the follow-up time window.

Splitting and merging population hazard

The easiest way to both split and to include population hazard information is by using `[lexpand]`. You may also fairly easily do it by hand by splitting first and then merging in your population hazard information.

Data requirements

The population hazard information must be available for each record and named `pop.haz`. The follow-up time variable must be named `"fot"` e.g. as a result of using `lexpand`. The `lex.dur` variable must also be present, containing person-year information.

Value

A `glm` object created using a custom Poisson family construct. Some `glm` methods are applicable.

Author(s)

Joonas Miettinen, Karri Seppa

References

Paul W Dickman, Andy Sloggett, Michael Hills, and Timo Hakulinen. Regression models for relative survival. Stat Med. 2004 Jan 15;23(1):51-64. doi:10.1002/sim.1597

See Also

[lexpand], [stats::poisson], [stats::glm]

Other main functions: [Surv\(\)](#), [rate\(\)](#), [relpois_ag\(\)](#), [sir\(\)](#), [sirspline\(\)](#), [survmean\(\)](#), [survtab\(\)](#), [survtab_ag\(\)](#)

Other relpois functions: [RPL](#), [relpois_ag\(\)](#), [rpcurve\(\)](#)

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)
## use the simulated rectal cancer cohort
data("sire", package = "popEpi")
sire$agegr <- cut(sire$dg_age, c(0,45,60,Inf), right=FALSE)

## usable straight away after splitting
fb <- c(0,3/12,6/12,1,2,3,4,5)
x <- lexpand(sire, birth = bi_date, entry = dg_date,
            exit = ex_date, status=status,
            breaks = list(fot=fb), pophaz=popmort)
rpm <- relpois(x, formula = lex.Xst %in% 1:2 ~ FOT + agegr)

## some methods for glm work. e.g. test for interaction

rpm2 <- relpois(x, formula = lex.Xst %in% 1:2 ~ FOT*agegr)
anova(rpm, rpm2, test="LRT")
AIC(rpm, rpm2)
## update() won't work currently
```

relpois_ag

Excess hazard Poisson model

Description

Estimate a Poisson Piecewise Constant Excess Hazards Model

Usage

```
relpois_ag(
  formula,
  data,
  d.exp,
  offset = NULL,
  breaks = NULL,
  subset = NULL,
  piecewise = TRUE,
  check = TRUE,
  ...
)
```

Arguments

formula	a formula with the counts of events as the response. Passed on to <code>glm</code> . May contain usage of the <code>offset()</code> function instead of supplying the offset for the Poisson model via the argument <code>offset</code> .
data	an <code>aggre</code> object (an aggregated data set; see <code>[as.aggre]</code> and <code>[aggre]</code>)
d.exp	the counts of expected cases. Mandatory. E.g. <code>d.exp = EXC_CASES</code> , where <code>EXC_CASES</code> is a column in <code>data</code> .
offset	the offset for the Poisson model, supplied as e.g. <code>offset = log(PTIME)</code> , where <code>PTIME</code> is a subject-time variable in <code>data</code> . Not mandatory, but almost always should be supplied.
breaks	optional; a numeric vector of <code>[a,b)</code> breaks to specify survival intervals over the follow-up time; if <code>NULL</code> , the existing breaks along the mandatory time scale mentioned in <code>formula</code> are used
subset	a logical vector or condition; e.g. <code>subset = sex == 1</code> ; limits the data before estimation
piecewise	logical; if <code>TRUE</code> , and if any time scale from <code>data</code> is used (mentioned) in the formula, the time scale is transformed into a factor variable indicating intervals on the time scale. Otherwise the time scale left as it is, usually a numeric variable. E.g. if <code>formula = counts ~ TS1*VAR1</code> , <code>TS1</code> is transformed into a factor before fitting model.
check	logical; if <code>TRUE</code> , performs check on the negativity excess cases by factor-like covariates in formula - negative excess cases will very likely lead to non-converging model
...	any other argument passed on to <code>[stats::glm]</code> such as <code>control</code> or <code>weights</code>

Value

A `relpois` object created using a custom Poisson family construct.

Author(s)

Joonas Miettinen, Karri Seppa

See Also

[lexpand], [poisson], [glm]

Other main functions: [Surv\(\)](#), [rate\(\)](#), [relpois\(\)](#), [sir\(\)](#), [sirspline\(\)](#), [survmean\(\)](#), [survtab\(\)](#), [survtab_ag\(\)](#)

Other relpois functions: [RPL](#), [relpois\(\)](#), [rpcurve\(\)](#)

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)
## use the simulated rectal cancer cohort
data(sire, package = "popEpi")
sire$agegr <- cut(sire$dg_age, c(0,45,60,Inf), right=FALSE)

## create aggregated example data
fb <- c(0,3/12,6/12,1,2,3,4,5)
x <- lexpand(sire, birth = bi_date, entry = dg_date,
            exit = ex_date, status=status %in% 1:2,
            breaks = list(fot=fb),
            pophaz=popmort, pp = FALSE,
            aggre = list(agegr, fot))

## fit model using aggregated data
rpm <- relpois_ag(formula = from0to1 ~ fot + agegr, data = x,
                 d.exp = d.exp, offset = log(pyrs))
summary(rpm)

## the usual functions for handling glm models work
rpm2 <- update(rpm, . ~ fot*agegr)
anova(rpm, rpm2, test="LRT")
AIC(rpm, rpm2)

## other features such as residuals or predicting are not guaranteed
## to work as intended.
```

robust_values

Convert values to numeric robustly

Description

Brute force solution for ensuring a variable is numeric by coercing a variable of any type first to factor and then to numeric

Usage

```
robust_values(num.values, force = FALSE, messages = TRUE)
```

Arguments

num.values	values to convert to numeric
force	logical; if TRUE, returns a vector of values where values that cannot be interpreted as numeric are set to NA; if FALSE, returns the original vector and gives a warning if any value cannot be interpreted as numeric.
messages	logical; if TRUE, returns a message of what was done with the num.values

Value

A numeric vector.

Note

Returns NULL if given num.values is NULL.

Author(s)

Joonas Miettinen

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)
## this works
values <- c("1", "3", "5")
values <- robust_values(values)

## this works
values <- c("1", "3", "5", NA)
values <- robust_values(values)

## this returns originals and throws warnings
values <- c("1", "3", "5", "a")
suppressWarnings(
  values <- robust_values(values)
)

## this forces "a" to NA and works otherwise; throws warning about NAs
values <- c("1", "3", "5", "a")
suppressWarnings(
  values <- robust_values(values, force=TRUE)
)
```

 rpcurve

Marginal piecewise parametric relative survival curve

Description

Fit a marginal relative survival curve based on a relpois fit

Usage

```
rpcurve(object)
```

Arguments

object a relpois object

Details

Estimates a marginal curve, i.e. the average of all possible individual curves.

Only supported when the reserved FOT variable was used in relpois. Computes a curve for each unique combination of covariates (e.g. 4 sets) and returns a weighted average curve based on the counts of subjects for each combination (e.g. 1000, 125, 50, 25 respectively). Fairly fast when only categorical variables have been used, otherwise go get a cup of coffee.

If delayed entry is present in data due to period analysis limiting, the marginal curve is constructed only for those whose follow-up started in the respective period.

Value

A data.table of relative survival curves.

Author(s)

Joonas Miettinen

See Also

Other relpois functions: [RPL](#), [relpois\(\)](#), [relpois_ag\(\)](#)

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)

## use the simulated rectal cancer cohort
data("sire", package = "popEpi")
ab <- c(0,45,55,65,70,Inf)
```

```

sire$agegr <- cut(sire$dg_age, breaks = ab, right = FALSE)

BL <- list(fot= seq(0,10,1/12))
pm <- data.frame(popEpi::popmort)
x <- lexpand(sire, breaks=BL, pophaz=pm,
             birth = bi_date,
             entry = dg_date, exit = ex_date,
             status = status %in% 1:2)

rpm <- relpois(x, formula = lex.Xst %in% 1:2 ~ -1+ FOT + agegr,
              fot.breaks=c(0,0.25,0.5,1:8,10))
pmc <- rpcurve(rpm)

## compare with non-parametric estimates
names(pm) <- c("sex", "per", "age", "haz")
x$agegr <- cut(x$dg_age, c(0,45,55,65,75,Inf), right = FALSE)
st <- survtab(fot ~ adjust(agegr), data = x, weights = "internal",
              pophaz = pm)

plot(st, y = "r.e2.as")
lines(y = pmc$est, x = pmc$Tstop, col="red")

```

RPL

Relative Poisson family object

Description

A family object for GLM fitting of relative Poisson models

Usage

RPL

Format

A list very similar to that created by `poisson()`.

Author(s)

Karri Seppa

See Also

Other relpois functions: [relpois\(\)](#), [relpois_ag\(\)](#), [rpcurve\(\)](#)

setaggre

Set aggre attributes to an object by modifying in place

Description

Coerces an R object to an aggre object, identifying the object as one containing aggregated counts, person-years and other information. `setaggre` modifies in place without taking any copies. Retains all other attributes.

Usage

```
setaggre(x, values = NULL, by = NULL, breaks = NULL)
```

Arguments

<code>x</code>	a <code>data.frame</code> or <code>data.table</code>
<code>values</code>	a character string vector; the names of value variables
<code>by</code>	a character string vector; the names of variables by which values have been tabulated
<code>breaks</code>	a list of breaks, where each element is a breaks vector as usually passed to e.g. <code>[splitLexisDT]</code> . The list must be fully named, with the names corresponding to time scales at the aggregate level in your data. Every unique value in a time scale variable in data must also exist in the corresponding vector in the breaks list.

Details

`setaggre` sets `x` to the `aggre` class in place without taking a copy as e.g. `as.data.frame.XXX` functions do; see e.g. `[data.table::setDT]`.

Value

Returns `x` invisibly after setting attributes to it without taking a copy. This function is called for its side effects.

Author(s)

Joonas Miettinen

See Also

Other aggregation functions: [aggre\(\)](#), [as.aggre\(\)](#), [lexpand\(\)](#), [summary.aggre\(\)](#)

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)
df <- data.frame(sex = rep(c("male", "female"), each = 5),
                 obs = rpois(10, rep(7,5, each=5)),
                 pyrs = rpois(10, lambda = 10000))
## without any breaks
setaggre(df, values = c("obs", "pyrs"), by = "sex")
df <- data.frame(df)
df$FUT <- 0:4
## with breaks list
setaggre(df, values = c("obs", "pyrs"), by = "sex", breaks = list(FUT = 0:5))
```

setclass	<i>Set the class of an object (convenience function for setattr(obj, "class", CLASS)); can add instead of replace</i>
----------	---

Description

Sets the class of an object in place to `cl` by replacing or adding

Usage

```
setclass(obj, cl, add = FALSE, add.place = "first")
```

Arguments

<code>obj</code>	and object for which to set class
<code>cl</code>	class to set
<code>add</code>	if TRUE, adds <code>cl</code> to the classes of the <code>obj</code> ; otherwise replaces the class information
<code>add.place</code>	"first" or "last"; adds <code>cl</code> to the front or to the back of the <code>obj</code> 's class vector

Author(s)

Joonas Miettinen

setcolsnull

Delete data.table columns if there

Description

Deletes columns in a `data.table` conveniently. May only delete columns that are found silently. Sometimes useful in e.g. `on.exit` expressions.

Usage

```
setcolsnull(  
  DT = NULL,  
  delete = NULL,  
  keep = NULL,  
  colorder = FALSE,  
  soft = TRUE  
)
```

Arguments

<code>DT</code>	a <code>data.table</code>
<code>delete</code>	a character vector of column names to be deleted
<code>keep</code>	a character vector of column names to keep; the rest will be removed; <code>keep</code> overrides <code>delete</code>
<code>colorder</code>	logical; if <code>TRUE</code> , also does <code>setcolorder</code> using <code>keep</code>
<code>soft</code>	logical; if <code>TRUE</code> , does not cause an error if any variable name in <code>keep</code> or <code>delete</code> is missing; <code>soft = FALSE</code> useful for programming sometimes

Value

Always returns `NULL` invisibly. This function is called for its side effects.

Author(s)

Joonas Miettinen

sibr*sibr - a simulated cohort of Finnish female breast cancer patients*

Description

sibr is a simulated cohort pertaining female Finnish breast cancer patients diagnosed between 1993-2012. Instead of actual original dates, the dates are masked via modest randomization within several time windows. The dataset is additionally a random sample of 10 000 cases from the pertaining time window.

Format

data.table with columns

- sex - gender of the patient (1 = female)
- bi_date - date of birth
- dg_date - date of cancer diagnosis
- ex_date - date of exit from follow-up (death or censoring)
- status - status of the person at exit; 0 alive; 1 dead due to pertinent cancer; 2 dead due to other causes
- dg_age - age at diagnosis expressed as fractional years

Details

The closing date for the pertinent data was 2012-12-31, meaning status information was available only up to that point — hence the maximum possible ex_date is 2012-12-31.

Author(s)

Karri Seppa

Source

The Finnish Cancer Registry

See Also

Other popEpi data: [ICSS](#), [meanpop_fi](#), [popmort](#), [sire](#), [stdpop101](#), [stdpop18](#)

Other survival data: [sire](#)

sir

*Calculate SIR or SMR***Description**

Poisson modelled standardised incidence or mortality ratios (SIRs / SMRs) i.e. indirect method for calculating standardised rates. SIR is a ratio of observed and expected cases. Expected cases are derived by multiplying the strata-specific population rate with the corresponding person-years of the cohort.

Usage

```
sir(
  coh.data,
  coh.obs,
  coh.pyrs,
  ref.data = NULL,
  ref.obs = NULL,
  ref.pyrs = NULL,
  ref.rate = NULL,
  subset = NULL,
  print = NULL,
  adjust = NULL,
  mstate = NULL,
  test.type = "homogeneity",
  conf.type = "profile",
  conf.level = 0.95,
  EAR = FALSE
)
```

Arguments

coh.data	aggregated cohort data, see e.g. [lexpand]
coh.obs	variable name for observed cases; quoted or unquoted. A vector when using mstate.
coh.pyrs	variable name for person years in cohort data; quoted (as a string 'myvar') or unquoted (AKA as a name; myvar)
ref.data	population data. Can be left NULL if coh.data is stratified in print. See [pophaz] for details.
ref.obs	variable name for observed cases; quoted or unquoted
ref.pyrs	variable name for person-years in population data; quoted or unquoted
ref.rate	population rate variable (cases/person-years). Overwrites arguments ref.pyrs and ref.obs. Quoted or unquoted
subset	logical condition to select data from coh.data before any computations

<code>print</code>	variable names to stratify results; quoted vector or unquoted named list with functions
<code>adjust</code>	variable names for adjusting without stratifying output; quoted vector or unquoted list
<code>mstate</code>	set column names for cause specific observations; quoted or unquoted. Relevant only when <code>coh.obs</code> length is two or more. See details.
<code>test.type</code>	Test for equal SIRs. Test available are 'homogeneity' and 'trend'.
<code>conf.type</code>	Confidence interval type: 'profile' (=default), 'wald' or 'univariate'.
<code>conf.level</code>	Level of type-I error in confidence intervals, default 0.05 is 95% CI.
<code>EAR</code>	logical; TRUE calculates Excess Absolute Risks for univariate SIRs. (see details)

Details

`sir` is a comprehensive tool for modelling SIRs/SMRs with flexible options to adjust and print SIRs, test homogeneity and utilize multi-state data. The cohort data and the variable names for observation counts and person-years are required. The reference data is optional, since the cohort data can be stratified (`print`) and compared to total.

Adjust and print

A SIR can be adjusted or standardised using the covariates found in both `coh.data` and `ref.data`. Variable to adjust are given in `adjust`. Variable names needs to match in both `coh.data` and `ref.data`. Typical variables to adjust by are gender, age group and calendar period.

`print` is used to stratify the SIR output. In other words, the variables assigned to `print` are the covariates of the Poisson model. Variable levels are treated as categorical. Variables can be assigned in both `print` and `adjust`. This means the output it adjusted and printed by these variables.

`print` can also be a list of expressions. This enables changing variable names or transforming variables with functions such as `cut` and `round`. For example, `agegroup` can be transformed on-the-go with

```
print = list(my_ag = cut(agegroup, my_ag_breaks))
```

ref.rate or ref.obs & ref.pyrs

The population rate variable can be given to the `ref.rate` parameter. That is, when using e.g. the `popmort` or a comparable data file, one may supply `ref.rate` instead of `ref.obs` and `ref.pyrs`, which will be ignored if `ref.rate` is supplied.

Note that if all the stratifying variables in `ref.data` are not listed in `adjust`, or when the categories are otherwise combined, the (unweighted) mean of rates is used for computing expected cases. This might incur a small bias in comparison to when exact numbers of observations and person-years are available.

mstate

E.g. using `lexpand` it's possible to compute counts for several outcomes so that the population at risk is same for each outcome such as a certain kind of cancer. The transition counts are in wide data format, and the relevant columns can be supplied to `sir` in a vector via the `coh.obs` argument. The name of the corresponding new column in `ref.data` is given in `mstate`. It's recommended to include the `mstate` variable in `adjust`, so the corresponding information should also be available in `ref.data`. More examples in `sir-vignette`.

This approach is analogous to where SIRs are calculated separately their own function calls.

Other parameters

univariate confidence intervals are calculated using exact Poisson intervals (`poisson.ci`). The options `profile` and `wald` are based on a Poisson regression model: `profile-likelihood` confidence intervals or Wald's normal-approximation. P-value is Poisson model based `conf.type` or calculated using the method described by Breslow and Day. Function automatically switches to another `conf.type` if calculation is not possible with a message. Usually model fit fails if there is print stratum with zero expected values.

The LRT p-value tests the levels of `print`. The test can be either "homogeneity", a likelihood ratio test where the model variables defined in `print` (factor) is compared to the constant model. Option "trend" tests if the linear trend of the continuous variable in `print` is significant (using model comparison).

EAR: Excess Absolute Risk

Excess Absolute Risk is a simple way to quantify the absolute difference between cohort risk and population risk. Make sure that the person-years are calculated accordingly before using EAR. (when using `mstate`)

Formula for EAR:

$$EAR = \frac{observed - expected}{personyears} \times 1000.$$

Data format

The data should be given in tabulated format. That is the number of observations and person-years are represented for each stratum. Note that also individual data is allowed as long as each observations, person-years, and `print` and `adjust` variables are presented in columns. The extra variables and levels are reduced automatically before estimating SIRs. Example of data format:

sex	age	period	obs	pyrs
0	1	2010	0	390
0	2	2010	5	385
1	1	2010	3	308
1	2	2010	12	315

Value

A `sir`-object that is a `data.table` with meta information in the attributes.

Author(s)

Matti Rantanen, Joonas Miettinen

See Also

[`lexpand`] [A SIR calculation vignette](#)

Other `sir` functions: `lines.sirspline()`, `plot.sirspline()`, `sir_exp()`, `sir_ratio()`, `sirspline()`

Other main functions: `Surv()`, `rate()`, `relpois()`, `relpois_ag()`, `sirspline()`, `survmean()`, `survtab()`, `survtab_ag()`

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)
data(popmort)
data(sire)
c <- lexpand( sire, status = status, birth = bi_date, exit = ex_date, entry = dg_date,
             breaks = list(per = 1950:2013, age = 1:100, fot = c(0,10,20,Inf)),
             aggre = list(fot, agegroup = age, year = per, sex) )
## SMR due other causes: status = 2
se <- sir( coh.data = c, coh.obs = 'from0to2', coh.pyrs = 'pyrs',
          ref.data = popmort, ref.rate = 'haz',
          adjust = c('agegroup', 'year', 'sex'), print = 'fot')
se
## for examples see: vignette('sir')
```

sir_exp

Calculate SMR

Description

Calculate Standardized Mortality Ratios (SMRs) using a single data set that includes observed and expected cases and additionally person-years.

sir_lex solves SMR from an [Epi::Lexis] object calculated with [lexpand].

sir_ag solves SMR from a [aggre] object calculated using [lexpand].

Usage

```
sir_exp(
  x,
  obs,
  exp,
  pyrs = NULL,
  print = NULL,
  conf.type = "profile",
  test.type = "homogeneity",
  conf.level = 0.95,
  subset = NULL
)

sir_lex(x, print = NULL, breaks = NULL, ...)

sir_ag(
```

```

    x,
    obs = "from0to1",
    print = attr(x, "aggre.meta")$by,
    exp = "d.exp",
    pyrs = "pyrs",
    ...
  )

```

Arguments

x	Data set e.g. aggre or Lexis object (see: [lexpand])
obs	Variable name of the observed cases in the data set
exp	Variable name or expression for expected cases
pyrs	Variable name for person-years (optional)
print	Variables or expression to stratify the results
conf.type	select confidence interval type: (default=) profile, wald, univariate
test.type	Test for equal SIRs. Test available are 'homogeneity' and 'trend'
conf.level	Level of type-I error in confidence intervals, default 0.05 is 95% CI
subset	a logical vector for subsetting data
breaks	a named list to split age group (age), period (per) or follow-up (fot).
...	pass arguments to sir_exp

Details

These functions are intended to calculate SMRs from a single data set that includes both observed and expected number of cases. For example utilizing the argument `pop.haz` of the [lexpand].

`sir_lex` automatically exports the transition from X to Y using the first state in `lex.Str` as 0 and all other as 1. No missing values is allowed in observed, `pop.haz` or person-years.

Value

A sir object

Author(s)

Matti Rantanen

See Also

[lexpand] [A SIR calculation vignette](#)

Other sir functions: [lines.sirspline\(\)](#), [plot.sirspline\(\)](#), [sir\(\)](#), [sir_ratio\(\)](#), [sirspline\(\)](#)

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)

BL <- list(fot = 0:5, per = c("2003-01-01", "2008-01-01", "2013-01-01"))

## Aggregated data
x1 <- leexpand(sire, breaks = BL, status = status != 0,
               birth = bi_date, entry = dg_date, exit = ex_date,
               pophaz=popmort,
               aggre=list(sex, period = per, surv.int = fot))
sir_ag(x1, print = 'period')

# no aggregate or breaks
x2 <- leexpand(sire, status = status != 0,
               birth = bi_date, entry = dg_date, exit = ex_date,
               pophaz=popmort)
sir_lex(x2, breaks = BL, print = 'per')
```

sir_ratio

Confidence intervals for the ratio of two SIRs/SMRs

Description

Calculate ratio of two SIRs/SMRs and the confidence intervals of the ratio.

Usage

```
sir_ratio(
  x,
  y,
  digits = 3,
  alternative = "two.sided",
  conf.level = 0.95,
  type = "exact"
)
```

Arguments

x a sir-object or a vector of two; observed and expected cases.
y a sir-object or a vector of two; observed and expected cases.

digits	number of digits in the output
alternative	The null-hypothesis test: (default:) two.sided, less, greater
conf.level	the type-I error in confidence intervals, default 0.95 for 95% CI.
type	How the binomial confidence intervals are calculated (default:) exact or asymptotic.

Details

Function works with pooled sir-objects i.e. the print argument in sir is ignored. Also x and y can be a vector of two where first index is the observed cases and second is expected cases (see examples). Note that the ratio of two SIR's is only applicable when the age distributions are similar in both populations.

Formula

The observed number of first sir O1 is considered as a Binomial variable with sample size of O1+O2. The confidence intervals for Binomial proportion A is solved using exact or asymptotic method. Now the CI for ratio O1/O2 is $B = A / (1 - A)$. And further the CI for SIR/SMR is $B * E2 / E1$. (Ederer and Mantel)

Value

A vector length of three: sir_ratio, and lower and upper confidence intervals.

Note

Parameter alternative is always two.sided when parameter type is set to asymptotic.

Author(s)

Matti Rantanen

References

Statistics with Confidence: Confidence Intervals and Statistical Guidelines, Douglas Altman, 2000. ISBN: 978-0-727-91375-3

See Also

[sir] [A SIR calculation vignette](#)

Other sir functions: `lines.sirspline()`, `plot.sirspline()`, `sir()`, `sir_exp()`, `sirspline()`

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)
## Ratio for sir-object and the same values given manually:
```

```
## create example dataset
dt1 <- data.frame(obs = rep(c(5,7), 10),
                  pyrs = rep(c(250,300,350,400), 5),
                  var = 1:20)
Ref <- data.frame(obs = rep(c(50,70,80,100), 5),
                  pyrs = rep(c(2500,3000,3500,4000), 5),
                  var = 1:20)
## sir using the function
s1 <- sir(coh.data = dt1, coh.obs = obs, coh.pyrs = pyrs,
          ref.data = Ref, ref.obs = obs, ref.pyrs = pyrs,
          adjust = var)

## Ratio is simply 1:
sir_ratio(s1, c(120, 150))
```

sire

sire - a simulated cohort of Finnish female rectal cancer patients

Description

sire is a simulated cohort pertaining female Finnish rectal cancer patients diagnosed between 1993-2012. Instead of actual original dates, the dates are masked via modest randomization within several time windows.

Format

data.table with columns

- sex - gender of the patient (1 = female)
- bi_date - date of birth
- dg_date - date of cancer diagnosis
- ex_date - date of exit from follow-up (death or censoring)
- status - status of the person at exit; 0 alive; 1 dead due to pertinent cancer; 2 dead due to other causes
- dg_age - age at diagnosis expressed as fractional years

Details

The closing date for the pertinent data was 2012-12-31, meaning status information was available only up to that point — hence the maximum possible ex_date is 2012-12-31.

Author(s)

Karri Seppa

Source

The Finnish Cancer Registry

See Also

Other popEpi data: [ICSS](#), [meanpop_fi](#), [popmort](#), [sibr](#), [stdpop101](#), [stdpop18](#)

Other survival data: [sibr](#)

sirspline	<i>Estimate splines for SIR or SMR</i>
-----------	--

Description

Splines for standardised incidence or mortality ratio. A useful tool to e.g. check whether a constant SIR can be assumed for all calendar periods, age groups or follow-up intervals. Splines can be fitted for these time dimensions separately or in the same model.

Usage

```
sirspline(  
  coh.data,  
  coh.obs,  
  coh.pyrs,  
  ref.data = NULL,  
  ref.obs = NULL,  
  ref.pyrs = NULL,  
  ref.rate = NULL,  
  subset = NULL,  
  print = NULL,  
  adjust = NULL,  
  mstate = NULL,  
  spline,  
  knots = NULL,  
  reference.points = NULL,  
  dependent.splines = TRUE  
)
```

Arguments

<code>coh.data</code>	cohort data with observations and at risk time variables
<code>coh.obs</code>	variable name for observed cases
<code>coh.pyrs</code>	variable name for person-years in cohort data
<code>ref.data</code>	aggregated population data
<code>ref.obs</code>	variable name for observed cases
<code>ref.pyrs</code>	variable name for person-years in population data

<code>ref.rate</code>	population rate observed/expected. This overwrites the parameters <code>ref.pyrs</code> and <code>ref.obs</code> .
<code>subset</code>	logical condition to subset <code>coh.data</code> before any computations
<code>print</code>	variable names for which to estimate SIRs/SMRs and associated splines separately
<code>adjust</code>	variable names for adjusting the expected cases
<code>mstate</code>	set column names for cause specific observations. Relevant only when <code>coh.obs</code> length is two or more. See help for <code>sir</code> .
<code>spline</code>	variable name(s) for the splines
<code>knots</code>	number knots (vector), pre-defined knots (list of vectors) or for optimal number of knots left <code>NULL</code>
<code>reference.points</code>	fixed reference values for rate ratios. If left <code>NULL</code> the smallest value is the reference point (where <code>SIR = 1</code>). Ignored if <code>dependent.splines = FALSE</code>
<code>dependent.splines</code>	logical; if <code>TRUE</code> , all splines are fitted in same model.

Details

See [`sir`] for help on SIR/SMR estimation in general; usage of splines is discussed below.

The spline variables

The model can include one, two or three splines variables. Variables can be included in the same model selecting `dependent.splines = TRUE` and SIR ratios are calculated (first one is the SIR, others SIR ratios). Reference points vector can be set via `reference.points` where first element of the vector is the reference point for first ratio.

Variable(s) to fit splines are given as a vector in argument `spline`. Order will affect the results.

dependent.splines

By default `dependent.splines` is `FALSE` and all splines are fitted in separate models. If `TRUE`, the first variable in `spline` is a function of a SIR and other(s) are ratios.

knots

There are three options to set knots to splines:

Set the number of knots for each spline variable with a **vector**. The knots are automatically placed to the quantiles of observed cases in cohort data. The first and last knots are always the maximum and minimum values, so knot value needs to be at least two.

Predefined knot places can be set with a **list** of vectors. The vector for each spline in the list specifies the knot places. The lowest and the largest values are the boundary knots and these should be checked beforehand.

If `knots` is left `NULL`, the model searches the optimal number of knots by model AIC by fitting models iteratively from 2 to 15 knots and the one with smallest AIC is selected. If `dependent.splines = TRUE`, the number of knots is searched by fitting each spline variable separately.

print

Splines can be stratified by the levels of variable given in `print`. If `print` is a vector, only the first variable is accounted for. The knots are placed globally for all levels of `print`. This also ensures

that the likelihood ratio test is valid. Splines are also fitted independently for each level of print. This allows for searching interactions, e.g. by fitting spline for period (splines='period') for each age group (print = 'agegroup').

p-values

The output p-value is a test of whether the splines are equal (homogenous) at different levels of print. The test is based on the likelihood ratio test, where the full model includes print and is compared to a null model without it. When (dependent.splines = TRUE) the p-value returned is a global p-value. Otherwise the p-value is spline-specific.

Value

A list of data.frames and vectors. Three spline estimates are named as spline.est.A/B/C and the corresponding values in spline.seq.A/B/C for manual plotting

Author(s)

Matti Rantanen, Joonas Miettinen

See Also

[splitMulti] [A SIR calculation vignette](#)

Other sir functions: [lines.sirspline\(\)](#), [plot.sirspline\(\)](#), [sir\(\)](#), [sir_exp\(\)](#), [sir_ratio\(\)](#)

Other main functions: [Surv\(\)](#), [rate\(\)](#), [relpois\(\)](#), [relpois_ag\(\)](#), [sir\(\)](#), [survmean\(\)](#), [survtab\(\)](#), [survtab_ag\(\)](#)

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)
## for examples see: vignette('sir')
```

splitLexisDT

Split case-level observations

Description

Split a Lexis object along one time scale (as [Epi::splitLexis]) with speed

Usage

```
splitLexisDT(lex, breaks, timeScale, merge = TRUE, drop = TRUE)
```

Arguments

lex	a Lexis object, split or not
breaks	a vector of [a,b) breaks to split data by
timeScale	a character string; name of the time scale to split by
merge	logical; if TRUE, retains all variables from the original data - i.e. original variables are repeated for all the rows by original subject
drop	logical; if TRUE, drops all resulting rows after expansion that reside outside the time window defined by the given breaks

Details

splitLexisDT is in essence a **data.table** version of splitLexis or survSplit for splitting along a single time scale. It requires a Lexis object as input, which may have already been split along some time scale.

Unlike splitLexis, splitLexisDT drops observed time outside the roof and floor of breaks by default - with drop = FALSE the functions have identical behaviour.

The Lexis time scale variables can be of any arbitrary format, e.g. Date, fractional years (see [Epi::cal.yr]) and [get.yrs].

Value

A data.table or data.frame (depending on options("popEpi.datatable"); see ?popEpi) object expanded to accommodate split observations.

Author(s)

Joonas Miettinen

See Also

Other splitting functions: [lexexpand\(\)](#), [splitMulti\(\)](#)

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)
library(Epi)
data("sire", package = "popEpi")
x <- Lexis(data=sire[1000:1100, ],
           entry = list(fot=0, per=get.yrs(dg_date), age=dg_age),
           exit=list(per=get.yrs(ex_date)), exit.status=status)
BL <- list(fot=seq(0, 5, by = 3/12), per=c(2008, 2013))

x2 <- splitMulti(x, breaks = BL, drop = FALSE)
```

```

x3 <- splitLexisDT(x, breaks = BL$fot, timeScale = "fot", drop = FALSE)
x3 <- splitLexisDT(x3, breaks = BL$per, timeScale = "per", drop = FALSE)

x4 <- splitLexis(x, breaks = BL$fot, time.scale = "fot")
x4 <- splitLexis(x4, breaks = BL$per, time.scale = "per")
## all produce identical results

## using Date variables
x <- Lexis(
  data=sire[1000:1100, ],
  entry = list(fot = 0L, per = dg_date, age = as.integer(dg_date - bi_date)),
  duration = as.integer(ex_date - dg_date),
  entry.status = 0L,
  exit.status = status
)
BL <- list(fot = 0:5*365.25, per = as.Date(c("2008-01-01", "2013-01-01")))

x2 <- splitMulti(x, breaks = BL, drop = FALSE)

x3 <- splitLexisDT(x, breaks = BL$fot, timeScale = "fot", drop = FALSE)
x3 <- splitLexisDT(x3, breaks = BL$per, timeScale = "per", drop = FALSE)

## splitLexis may not work when using Dates

```

splitMulti

Split case-level observations

Description

Split a Lexis object along multiple time scales with speed and ease

Usage

```

splitMulti(
  data,
  breaks = NULL,
  ...,
  drop = TRUE,
  merge = TRUE,
  verbose = FALSE
)

```

Arguments

data	a Lexis object with event cases as rows
breaks	a list of named numeric vectors of breaks; see Details and Examples

...	alternate way of supplying breaks as named vectors; e.g. <code>fot = 0:5</code> instead of <code>breaks = list(fot = 0:5)</code> ; if <code>breaks</code> is not <code>NULL</code> , <code>breaks</code> is used and any breaks passed through ... are NOT used; note also that due to partial matching of argument names in R, if you supply e.g. <code>dat = my_breaks</code> and you do not pass argument data explicitly (<code>data = my_data</code>), then R interprets this as <code>data = my_breaks</code> — so choose the names of your time scales wisely
drop	logical; if <code>TRUE</code> , drops all resulting rows after expansion that reside outside the time window defined by the given breaks
merge	logical; if <code>TRUE</code> , retains all variables from the original data - i.e. original variables are repeated for all the rows by original subject
verbose	logical; if <code>TRUE</code> , the function is chatty and returns some messages along the way

Details

`splitMulti` is in essence a **data.table** version of `splitLexis` or `survSplit` for splitting along multiple time scales. It requires a Lexis object as input.

The breaks must be a list of named vectors of the appropriate type. The breaks are fully explicit and left-inclusive and right exclusive, e.g. `fot=c(0,5)` forces the data to only include time between `[0,5)` for each original row (unless `drop = FALSE`). Use `Inf` or `-Inf` for open-ended intervals, e.g. `per=c(1990,1995,Inf)` creates the intervals `[1990,1995)`, `[1995, Inf)`.

Instead of specifying breaks, one may make use of the ... argument to pass breaks: e.g.

```
splitMulti(x, breaks = list(fot = 0:5))
```

is equivalent to

```
splitMulti(x, fot = 0:5).
```

Multiple breaks can be supplied in the same manner. However, if both `breaks` and ... are used, only the breaks in `breaks` are utilized within the function.

The Lexis time scale variables can be of any arbitrary format, e.g. `Date`, fractional years (see `[Epi::cal.yr]`) and `[get.yrs]`, or other.

Value

A `data.table` or `data.frame` (depending on options(`"popEpi.datatable"`); see `?popEpi`) object expanded to accommodate split observations.

News for version 0.5.0

Fixed `popEpi::splitMulti` when `merge = FALSE`. Did not include `lex.dur` previously which caused an error.

Author(s)

Joonas Miettinen

See Also

`[Epi::splitLexis]`, `[Epi::Lexis]`, `[survival::survSplit]`

Other splitting functions: [lexexpand\(\)](#), [splitLexisDT\(\)](#)

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)
#### let's prepare data for computing period method survivals
#### in case there are problems with dates, we first
#### convert to fractional years.

library("Epi")
library("data.table")
data("sire", package = "popEpi")
x <- Lexis(data=sire[dg_date < ex_date, ],
           entry = list(fot=0, per=get.yrs(dg_date), age=dg_age),
           exit=list(per=get.yrs(ex_date)), exit.status=status)
x2 <- splitMulti(x, breaks = list(fot=seq(0, 5, by = 3/12), per=c(2008, 2013)))
# equivalently:
x2 <- splitMulti(x, fot=seq(0, 5, by = 3/12), per=c(2008, 2013))

## using dates; note: breaks must be expressed as dates or days!
x <- Lexis(
  data = sire[dg_date < ex_date, ],
  entry = list(fot = 0L, per = dg_date, age = as.integer(dg_date - bi_date)),
  duration = as.integer(ex_date - dg_date),
  entry.status = 0L,
  exit.status = status
)
BL <- list(fot = seq(0, 5, by = 3/12)*365.242199,
           per = as.Date(paste0(c(1980:2014), "-01-01")),
           age = c(0,45,85,Inf)*365.242199)
x2 <- splitMulti(x, breaks = BL, verbose=TRUE)

## multistate example (healty - sick - dead)
sire2 <- data.frame(sire)
sire2 <- sire2[sire2$dg_date < sire2$ex_date, ]

set.seed(1L)
not_sick <- sample.int(nrow(sire2), 6000L, replace = FALSE)
sire2$dg_date[not_sick] <- NA
sire2$status[!is.na(sire2$dg_date) & sire2$status == 0] <- -1

sire2$status[sire2$status==2] <- 1
sire2$status <- factor(sire2$status, levels = c(0, -1, 1),
                      labels = c("healthy", "sick", "dead"))

xm <- Lexis(data = sire2,
           entry = list(fot=0, per=get.yrs(bi_date), age=0),
           exit = list(per=get.yrs(ex_date)), exit.status=status)
xm2 <- cutLexis(xm, cut = get.yrs(xm$dg_date),
               timescale = "per",
```

```

      new.state = "sick")
xm2[xm2$lex.id == 6L, ]

xm2 <- splitMulti(xm2, breaks = list(fot = seq(0,150,25)))
xm2[xm2$lex.id == 6L, ]

```

stdpop101	<i>World standard population by 1 year age groups from 1 to 101. Sums to 100 000.</i>
-----------	---

Description

World standard population by 1 year age groups from 1 to 101. Sums to 100 000.

Format

data.table with columns

- world_std weight that sums to 100000 (numeric)
- agegroup age group from 1 to 101 (numeric)

Source

Standard population is from: [world standard population "101of1"](#)

See Also

Other popEpi data: [ICSS](#), [meanpop_fi](#), [popmort](#), [sibr](#), [sire](#), [stdpop18](#)

Other weights: [ICSS](#), [direct_standardization](#), [stdpop18](#)

stdpop18	<i>Standard populations from 2000: world, Europe and Nordic.</i>
----------	--

Description

World, European, and Nordic standard populations by 18 age categories. Sums to 100000.

Format

data.table with columns

- agegroup, age group in 18 categories (character)
- world, World 2000 standard population (numeric)
- europe, European standard population (numeric)
- nordic, Nordic standard population (numeric)

Source

Nordcan, 2000

See Also

Other popEpi data: [ICSS](#), [meanpop-fi](#), [popmort](#), [sibr](#), [sire](#), [stdpop101](#)
Other weights: [ICSS](#), [direct_standardization](#), [stdpop101](#)

summary.aggre	<i>Summarize an aggre Object</i>
---------------	----------------------------------

Description

summary method function for aggre objects; see [as.aggre] and [aggre].

Usage

```
## S3 method for class 'aggre'
summary(object, by = NULL, subset = NULL, ...)
```

Arguments

object	an aggre object
by	list of columns to summarize by - e.g. list(V1, V2) where V1 and V2 are columns in the data.
subset	a logical condition to subset results table by before summarizing; use this to limit to a certain stratum. E.g. subset = sex == "male"
...	unused

Value

Returns a data.table — a further aggregated version of object.

Author(s)

Joonas Miettinen

See Also

Other aggregation functions: [aggre\(\)](#), [as.aggre\(\)](#), [lexpand\(\)](#), [setaggre\(\)](#)

summary.survtab	<i>Summarize a survtab Object</i>
-----------------	-----------------------------------

Description

Summary method function for survtab objects; see [survtab_ag]. Returns estimates at given time points or all time points if t and q are both NULL.

Usage

```
## S3 method for class 'survtab'
summary(object, t = NULL, subset = NULL, q = NULL, ...)
```

Arguments

object	a survtab object
t	a vector of times at which time points (actually intervals that contain t) to print summary table of survival function estimates by strata; values not existing in any interval cause rows containing only NAs to be returned.
subset	a logical condition to subset results table by before printing; use this to limit to a certain stratum. E.g. subset = sex == "male"
q	a named list of quantiles to include in returned data set, where names must match to estimates in object; returns intervals where the quantiles are reached first; e.g. list(surv.obs = 0.5) finds the interval where surv.obs is 0.45 and 0.55 at the beginning and end of the interval, respectively; returns rows with NA values for quantiles not reached in estimates (e.g. if q = list(surv.obs = 0.5) but lowest estimate is 0.6); see Examples.
...	unused; required for congruence with other summary methods

Details

Note that this function returns the intervals and NOT the time points corresponding to quantiles / estimates corresponding to time points. If you want precise estimates at time points that are not interval breaks, add the time points as breaks and re-estimate the survival time function. In interval-based estimation, the estimates denote e.g. probability of dying *during* the interval, so time points within the intervals are not usually considered at all. See e.g. Seppa, Dyba, and Hakulinen (2015).

Value

A data.table: a slice from object based on t, subset, and q.

Author(s)

Joonas Miettinen

References

Seppa K., Dyba T. and Hakulinen T.: Cancer Survival, Reference Module in Biomedical Sciences. Elsevier. 08-Jan-2015. doi:[10.1016/B9780128012383.027458](https://doi.org/10.1016/B9780128012383.027458)

See Also

Other survtab functions: [Surv\(\)](#), [lines.survtab\(\)](#), [plot.survtab\(\)](#), [print.survtab\(\)](#), [survtab\(\)](#), [survtab_ag\(\)](#)

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)

library(Epi)

## NOTE: recommended to use factor status variable
x <- Lexis(entry = list(FUT = 0, AGE = dg_age, CAL = get.yrs(dg_date)),
          exit = list(CAL = get.yrs(ex_date)),
          data = sire[sire$dg_date < sire$ex_date, ],
          exit.status = factor(status, levels = 0:2,
                              labels = c("alive", "canD", "othD")),
          merge = TRUE)
## pretend some are male
set.seed(1L)
x$sex <- rbinom(nrow(x), 1, 0.5)
## observed survival
st <- survtab(Surv(time = FUT, event = lex.Xst) ~ sex, data = x,
              surv.type = "cif.obs",
              breaks = list(FUT = seq(0, 5, 1/12)))

## estimates at full years of follow-up
summary(st, t = 1:5)

## interval estimate closest to 75th percentile, i.e.
## first interval where surv.obs < 0.75 at end
## (just switch 0.75 to 0.5 for median survival, etc.)
summary(st, q = list(surv.obs = 0.75))
## multiple quantiles
summary(st, q = list(surv.obs = c(0.75, 0.90), CIF_canD = 0.20))

## if you want all estimates in a new data.frame, you can also simply do

x <- as.data.frame(st)
```

Surv

Survival Objects

Description

`popEpi::Surv` simply calls `[survival::Surv]`. This wrapper was written simply to avoid doing e.g. `library(survival)` when `Surv` is used in a formula evaluated by `popEpi`.

Usage

```
Surv(  
  time,  
  time2,  
  event,  
  type = c("right", "left", "interval", "counting", "interval2"),  
  origin = 0  
)
```

Arguments

<code>time</code>	See <code>[survival::Surv]</code>
<code>time2</code>	See <code>[survival::Surv]</code>
<code>event</code>	See <code>[survival::Surv]</code>
<code>type</code>	See <code>[survival::Surv]</code>
<code>origin</code>	See <code>[survival::Surv]</code>

Value

See `[survival::Surv]`.

See Also

Other main functions: [rate\(\)](#), [relpois\(\)](#), [relpois_ag\(\)](#), [sir\(\)](#), [sirspline\(\)](#), [survmean\(\)](#), [survtab\(\)](#), [survtab_ag\(\)](#)

Other `survtab` functions: [lines.survtab\(\)](#), [plot.survtab\(\)](#), [print.survtab\(\)](#), [summary.survtab\(\)](#), [survtab\(\)](#), [survtab_ag\(\)](#)

Other `survmean` functions: [lines.survmean\(\)](#), [plot.survmean\(\)](#), [survmean\(\)](#)

surv_estimate	<i>Survival Time Statistics</i>
---------------	---------------------------------

Description

Functions used for estimation of various survival time statistics. E.g. relative survival.

Usage

```
surv_estimate(
  dt,
  ts_fut_col_nm,
  stratum_col_nms = NULL,
  value_col_nms = NULL,
  estimators = "S_ch",
  adjust_col_nms = NULL,
  weight_dt = NULL,
  conf_methods = "log",
  conf_lvls = 0.95,
  empty_interval_action = "warning"
)

surv_collapse_ts_1d(
  dt,
  ts_col_nm,
  stratum_col_nms = NULL,
  value_col_nms = NULL,
  test_expr = NULL,
  mandatory_breaks = NULL
)

surv_collapse_strata_list(
  dt,
  stratum_col_nms,
  collapse_stratum_col_nms,
  value_col_nms = NULL,
  test_expr = NULL
)

surv_collapse_strata_1d(
  dt,
  stratum_col_nms,
  collapse_stratum_col_nm,
  value_col_nms = NULL,
  test_expr = quote(min(t_at_risk) > 0)
)
```

```

surv_observed_weight_dt(lexis, aggre_by, subset = NULL, breaks = NULL)

surv_brenner_weight_dt(
  standard_weight_dt,
  observed_weight_dt = NULL,
  collapse_stratum_col_nms = NULL
)

surv_individual_weights(
  lexis,
  standard_weight_dt,
  observed_weight_dt = NULL,
  collapse_stratum_col_nms = NULL,
  subset = NULL,
  breaks = NULL
)

lexis_breaks_collapse_1d(
  lexis,
  breaks_1d,
  test_expr = NULL,
  mandatory_breaks = NULL
)

surv_lexis(
  lexis,
  breaks,
  merge_dt = NULL,
  merge_dt_by = NULL,
  aggre_by = NULL,
  subset = NULL,
  first_record_by = NULL,
  split_merge_aggregate_optional_args = NULL,
  estimators = "S_ch",
  conf_methods = "log",
  conf_lvls = 0.95,
  weights = NULL,
  surv_estimate_optional_args = NULL
)

```

Arguments

dt	[data.table] (no default) Dataset containing aggregate statistics which can be used to compute survival function estimates. Must also have column box_id, a running number like the one produced by [lexis_split_merge_aggregate_by_stratum].
ts_fut_col_nm	[character] (no default) Name of time scale column over which survival estimates will be computed.

E.g. "ts_fut". dt must contain columns named `paste0(ts_fut_col_nm, "_", c("id", "start", "stop"))`, e.g. `c("ts_fut_id", "ts_fut_start", "ts_fut_stop")`.

`stratum_col_nms`

[NULL, character] (default NULL)

Stratum column names in dt, if any.

- NULL: If dt was the result of calling `[lexis_split_merge_aggregate_by_stratum]`, then `stratum_col_nms` is taken from the attributes of dt. If not, this causes no stratification of output.
- character: dt is stratified by these columns. `character(0)` is also allowed and causes no stratification of output.

`value_col_nms`

[NULL, character] (default NULL)

Value column names in dt, if any.

- NULL: If dt was the result of calling `[lexis_split_merge_aggregate_by_stratum]`, then `value_col_nms` is taken from the attributes of dt. Else we take double and integer columns whose names match regex `"(^t_)|(^n_)"` as the names of the value columns.
- character: One or more names of columns in dt containing values to be included in the output in addition to the estimate etc. columns. E.g. `value_col_nms = c("n_events", "t_at_risk")`.

`estimators`

[character, list] (default "S_ch")

One or more names of estimators whose estimates will be computed into dt.

- character: Causes formulae defined internally into `popEpi` to be used to compute the estimates and their standard errors. For available options see **Details**.
- list: Each named element must be a quoted (see [quote](#) / [substitute](#) / [str2lang](#) / etc) R expression, which when evaluated produces the estimates (est) and standard errors (se). E.g. `list(est = quote(n_events / t_at_risk), se = quote(...))`.

`adjust_col_nms`

[NULL, character] (default NULL)

Names of stratifying columns in dt by which the estimates should be adjusted. No adjusting is performed if this argument is NULL or `character(0)`, the latter whether supplied or inferred.

- NULL: Take `intersect(names(dt), names(weight_dt))`.
- character: Adjust by these column names. Must be found in `names(dt)`, `names(weight_dt)`, and `stratum_col_nms`. This can actually be a subset of the stratifying columns of `weight_dt`. See below how this works.

`weight_dt`

[NULL, data.table] (default NULL)

Weights for direct adjusting or Brenner weighting. See **Details** to understand how the `weight_dt` argument is used.

- NULL: No direct adjusting nor Brenner weighting is performed.
- data.table: Contains weighting strata and either column "weight" for direct adjusting or column "brenner_weight" for Brenner weighting.

A table of weights must fulfill these requirements:

	• Is a <code>data.table</code>. • Has at least one stratifying column. No duplicate strata are permitted, e.g. the same age group twice in a table stratified by age group only. • Has column "weight". All values must be ≥ 0. No missing values are allowed.
<code>conf_methods</code>	Passed one at a time to <code>[directadjusting::directly_adjusted_estimates]</code> .
<code>conf_lvls</code>	Passed one at a time to <code>[directadjusting::directly_adjusted_estimates]</code> .
<code>empty_interval_action</code>	<code>[character, list]</code> What to do when <code>dt</code> contains an empty survival interval — one where <code>t_at_risk</code> or <code>n_at_risk_eff</code> is empty. Conceptually a survival estimate is not defined for an empty interval — if no subjects are in follow-up, there is no observed survival. Must be one of the following: • <code>list</code>: arguments to pass to either <code>[surv_collapse_ts_1d]</code> or to <code>[surv_collapse_strata_1d]</code>. The function in question is defined with the named element <code>function_name</code> which must be either <code>"surv_collapse_ts_1d"</code> or <code>"surv_collapse_strata_1d"</code>. E.g. <code>list(function_name = "surv_collapse_strata_1d", collapse_stratum_col_nm = "age_group")</code>. • <code>"warning"</code>: Emit a warning with <code>[warning]</code> if any empty intervals were found. • <code>"error"</code>: Raise an error with <code>[stop]</code>. • <code>"message"</code>: Emit a message with <code>[message]</code> • <code>"surv_collapse_ts_1d"</code>: Call <code>popEpi::surv_collapse_ts_1d</code> before estimation silently. • <code>"nothing"</code>: Estimation is performed normally and silently. Some estimates will be NA.
<code>ts_col_nm</code>	<code>[character]</code> (no default) Name of time scale column whose intervals are possibly collapsed. This does not need to be the follow-up time scale itself although it usually is. E.g. <code>ts_col_nm = "ts_fut"</code>.
<code>test_expr</code>	<code>[NULL, call]</code> (default <code>NULL</code>) This expression is evaluated for every interval. If the test does not pass, the interval must be aggregated with one of its neighbours.
<code>mandatory_breaks</code>	<code>[NULL, other]</code> (default <code>NULL</code>) These breaks cannot be aggregated over. E.g. with <code>mandatory_breaks = 1.0</code> this function will not attempt to aggregate <code>]0.9, 1.0]</code> and <code>]1.0, 1.1]</code> into <code>]0.9, 1.1]</code>, not even if it is the only way to produce intervals which all pass <code>test_expr</code>.
<code>collapse_stratum_col_nms</code>	<code>[NULL, character]</code> (default <code>NULL</code>) Optional argument of names of stratum column names in both <code>observed_weight_dt</code> and <code>standard_weight_dt</code>. If this argument is supplied, strata defined by these columns are attempted to be combined with their neighbours in the event of a zero in the corresponding <code>observed_weight_dt\$weight</code> value. For instance,

with `collapse_stratum_col_nms = "agegroup"` and the first age group having an observed weight of zero, we combine the first age group with the second for the purpose of calculating the individual weight. See **Details**.

- `NULL`: No automatic combination of strata is performed.
- `character`: Combine neighbouring strata defined by these column names. E.g. `"agegroup"`.

<code>collapse_stratum_col_nm</code>	[character] As the <code>collapse_stratum_col_nms</code> argument of <code>popEpi::surv_collapse_strata_list</code> — and passed to it, as that argument — but must be of length one.
<code>lexis</code>	[Lexis] (no default) A Lexis dataset (<code>[Epi::Lexis] / [Lexis_dt]</code>).
<code>aggre_by</code>	Passed to <code>[lexis_split_merge_aggregate_by_stratum]</code> .
<code>subset</code>	Passed to <code>[lexis_split_merge_aggregate_by_stratum]</code> .
<code>breaks</code>	Passed to <code>[lexis_split_merge_aggregate_by_stratum]</code> .
<code>standard_weight_dt</code>	[data.table] (no default) Table of standardisation weights, e.g. ICSS weights. A table of weights must fulfill these requirements: • Is a <code>data.table</code>. • Has at least one stratifying column. No duplicate strata are permitted, e.g. the same age group twice in a table stratified by age group only. • Has column <code>"weight"</code>. All values must be ≥ 0. No missing values are allowed.
<code>observed_weight_dt</code>	[NULL, data.table] (default NULL) Table of weights in your dataset. • <code>NULL</code>: Weights are computed using <code>surv_observed_weight_dt</code>. • <code>data.table</code>: Must be a valid table of weights. A table of weights must fulfill these requirements: • Is a <code>data.table</code>. • Has at least one stratifying column. No duplicate strata are permitted, e.g. the same age group twice in a table stratified by age group only. • Has column <code>"weight"</code>. All values must be ≥ 0. No missing values are allowed.
<code>breaks_1d</code>	[list] (no default) List of breaks with one element. These define the initial intervals which are subject to being combined. E.g. <code>list(ts_fut = seq(0, 5, 1 / 12))</code>.
<code>merge_dt</code>	Passed to <code>[lexis_split_merge_aggregate_by_stratum]</code> .
<code>merge_dt_by</code>	Passed to <code>[lexis_split_merge_aggregate_by_stratum]</code> .
<code>first_record_by</code>	Passed to <code>[lexis_split_merge_aggregate_by_stratum]</code> .

- split_merge_aggregate_optional_args**
 [NULL, list] (default NULL)
 Optional arguments to pass to [lexis_split_merge_aggregate_by_stratum].
- NULL: No additional arguments.
 - list: Pass these arguments. E.g. list(breaks_collapse_args = list(mandatory_breaks = 0:5)).
- weights**
 [NULL, data.table, character] (default NULL)
 Weights for adjusting estimates.
- NULL: No adjusting is performed.
 - data.table: Passed to [surv_estimate] arg weight_dt.
 - character: Passed to [lexis_split_merge_aggregate_by_stratum] argument weight_col_nm.
- surv_estimate_optional_args**
 [NULL, list]
 Optional additional arguments to pass to [surv_estimate]. Arguments passed via surv_estimate_optional_args override internally defined ones. See section **Functions**.
- NULL: No additional arguments are passed.
 - list: Pass these arguments. E.g. list(empty_interval_action = "surv_collapse_ts_1d").

Value

popEpi::surv_estimate

Returns a data.table containing all columns in dt (unless direct adjusting was performed — then the adjusting column(s) are not present) and additional columns of estimates, standard errors, and confidence intervals.

popEpi::surv_individual_weights

Returns a vector of weights of length nrow(lexis).

popEpi::surv_lexis

Returns a data.table as produced by surv_estimate.

Functions

popEpi::surv_estimate

Compute survival time function estimates. Performs the following steps:

- Handles estimators. Elements of estimators that are character strings cause the corresponding pre-defined formulae to be used. E.g. "S_ch". Pre-defined estimators with their formulae:

Name	Explanation
h_ch	Constant hazards estimate of... hazard. The most basic hazard estimate where the number of events is Poisson.
H_ch	Cumulative hazard function estimate based on the above.
h_exp_e2_ch	Expected constant hazards estimated by the Ederer II method.
h_exc_e2_ch	Excess hazard, observed - expected. Based on Ederer II expected hazard.

S_ch	Observed survival, constant hazards estimator. We estimate the hazard first for each survival interval and c
F_ch	The is the cumulative density / distribution function. It is simply $1 - S$. Based on constant hazards.
S_lt	Observed survival, lifetable estimator. We first estimate the (conditional) probabilities of surviving each s
H_lt	This cumulative hazard estimator is derived from S_lt. We don't know the standard error formula for this
h_lt	This hazard estimator is derived from S_lt. We don't know the standard error formula for this one.
F_lt	This is simply $1 - S_{lt}$.
S_exp_e2_lt	Expected Ederer II survival, lifetable estimator.
S_exp_e2_ch	Expected Ederer II survival, constant hazards estimator.
RS_e2_lt	Relative survival, Ederer II, lifetable estimator.
RS_e2_ch	Relative survival, Ederer II, constant hazards estimator.
NS_pp_lt	Net survival, Pohar Perme, constan hazards estimator. The Pohar Perme method is the least biased method
NS_pp_ch	Net survival, Pohar Perme, lifetable estimator. The Pohar Perme method is the least biased method for est
h_lt[x, y]	Cause-specific (transition-specific) hazard, life table estimator, for transitioning from x to y. We don't kno
h_ch[x, y]	Cause-specific (transition-specific) hazard, constant hazards estimator, for transitioning from x to y. Both
H_lt[x, y]	Cumulative version of h_lt[x, y]. We don't know the standard error formula for this one.
H_ch[x, y]	Cumulative version of h_ch[x, y].
S_lt[x, y]	Cause-specific (transition-specific) survival, life table estimator, for transitioning from x to y. Both x and y
S_ch[x, y]	Cause-specific (transition-specific) survival, constant hazards estimator, for transitioning from x to y. Both
F_lt[x, y]	Cause-specific (transition-specific) (absolute) risk, lifetable estimator for transitioning from x to y. Both x
F_ch[x, y]	Cause-specific (transition-specific) (absolute) risk, constant hazards estimator. Here we use the estimator p
RF_e2_lt	""Relative"" absolute risk, lifetable estimator. This was proposed by Karri Seppa at the Finnish Cancer Re
RF_e2_ch	""Relative"" absolute risk, constant hazards estimator. This was proposed by Karri Seppa at the Finnish C
h_ch_bw	Constant hazards estimate of... hazard. The most basic hazard estimate where the number of events is Pois
H_ch_bw	Cumulative hazard function estimate based on the above. The suffix _bw implies that the estimates are bas
h_exp_e2_ch_bw	Expected constant hazards estimated by the Ederer II method. The suffix _bw implies that the estimates ar
h_exc_e2_ch_bw	Excess hazard, observed - expected. Based on Ederer II expected hazard. The suffix _bw implies that the e
S_ch_bw	Observed survival, constant hazards estimator. We estimate the hazard first for each survival interval and c
F_ch_bw	The is the cumulative density / distribution function. It is simply $1 - S$. Based on constant hazards. The su
S_lt_bw	Observed survival, lifetable estimator. We first estimate the (conditional) probabilities of surviving each s
H_lt_bw	This cumulative hazard estimator is derived from S_lt. We don't know the standard error formula for this
h_lt_bw	This hazard estimator is derived from S_lt. We don't know the standard error formula for this one. The s
F_lt_bw	This is simply $1 - S_{lt}$. The suffix _bw implies that the estimates are based on Brenner weighted statistics
S_exp_e2_lt_bw	Expected Ederer II survival, lifetable estimator. The suffix _bw implies that the estimates are based on Bre
S_exp_e2_ch_bw	Expected Ederer II survival, constant hazards estimator. The suffix _bw implies that the estimates are base
RS_e2_lt_bw	Relative survival, Ederer II, lifetable estimator. The suffix _bw implies that the estimates are based on Bre
RS_e2_ch_bw	Relative survival, Ederer II, constant hazards estimator. The suffix _bw implies that the estimates are base
NS_pp_lt_bw	Net survival, Pohar Perme, constan hazards estimator. The Pohar Perme method is the least biased method
NS_pp_ch_bw	Net survival, Pohar Perme, lifetable estimator. The Pohar Perme method is the least biased method for est
h_lt[x, y]_bw	Cause-specific (transition-specific) hazard, life table estimator, for transitioning from x to y. We don't kno
h_ch[x, y]_bw	Cause-specific (transition-specific) hazard, constant hazards estimator, for transitioning from x to y. Both
H_lt[x, y]_bw	Cumulative version of h_lt[x, y]. We don't know the standard error formula for this one. The suffix _b
H_ch[x, y]_bw	Cumulative version of h_ch[x, y]. The suffix _bw implies that the estimates are based on Brenner weighl
S_lt[x, y]_bw	Cause-specific (transition-specific) survival, life table estimator, for transitioning from x to y. Both x and y
S_ch[x, y]_bw	Cause-specific (transition-specific) survival, constant hazards estimator, for transitioning from x to y. Both
F_lt[x, y]_bw	Cause-specific (transition-specific) (absolute) risk, lifetable estimator for transitioning from x to y. Both x
F_ch[x, y]_bw	Cause-specific (transition-specific) (absolute) risk, constant hazards estimator. Here we use the estimator p
RF_e2_lt_bw	""Relative"" absolute risk, lifetable estimator. This was proposed by Karri Seppa at the Finnish Cancer Re
RF_e2_ch_bw	""Relative"" absolute risk, constant hazards estimator. This was proposed by Karri Seppa at the Finnish C

- If `weight_dt` is a `data.table` with column "brenner_weight", we merge the weights into `dt`, multiply all `value_col_nms` with the respective Brenner weight, and sum `value_col_nms` over the adjusting strata in `weight_dt`. E.g. if `dt` is stratified by `sex` and `ag` and `weight_dt` contains columns `ag` and `brenner_weight` then the resulting table will have weighted sums of `value_col_nms` by `sex` (and the intervals). This approach is the same as assigning weights for each individual before splitting and aggregating when the individual weights are based on strata only (and not maybe some continuous variable).
- Check columns `t_at_risk`, `n_at_risk_eff` for zeroes/NAs if they are in `dt`. Intervals with e.g. `t_at_risk == 0` have no survival probability defined for them which causes NA values (i.e. zero divided by zero is not defined). The action taken depends on argument `empty_interval_action`. `t_at_risk_bw` and `n_at_risk_eff_bw` are used instead of the `_bw`-less versions if found.
- Add column `delta_t` as the difference between the `_stop` and `_start` columns of the follow-up time scale, e.g. `ts_fut_stop - ts_fut_start`.
- Armed with a list of expressions based on estimates, called `expressions`, for each `i`:
 - Evaluate each element of `expressions[[i]]` and add the result into `dt`. E.g. `S_ch` and `S_ch_se`.
- Call `[directadjusting::directly_adjusted_estimates]`. If no weights were given for direct adjusting then this step simply produces the confidence intervals.
- If direct adjusting was performed, the summary statistics such as `n_events` are summed over the adjusting strata and will be included in the output. These are not weighted averages/sums but simple sums.
- Return a `data.table` containing all columns in `dt` (unless direct adjusting was performed — then the adjusting column(s) are not present) and additional columns of estimates, standard errors, and confidence intervals.

popEpi::surv_brenner_weight_dt

`popEpi::surv_brenner_weight_dt` performs the following steps:

- Merge `standard_weight_dt` and `observed_weight_dt` into one table. Scale the standard weights and the observed weights to sum into one (they are in separate columns). E.g. one stratum has `weight_standard = 0.5` and `weight_observed = 0.4`.
- At this point the weight table might look like this:

ag	weight_standard	weight_observed
[0,15[	0.00000	0.0000000
[15,20[	0.00164	0.0000000
[20,25[	0.00215	0.0004110
[25,30[	0.00416	0.0030037
[30,35[	0.00842	0.0092639
[35,40[	0.01874	0.0174213
[40,45[	0.03489	0.0317440
[45,50[	0.04906	0.0641520

[50,55[	0.07094	0.0962755
[55,60[	0.09641	0.1094916
[60,65[	0.13359	0.1531554
[65,70[	0.14234	0.1582459
[70,75[	0.14766	0.1212217
[75,80[	0.14131	0.0954218
[80,85[	0.09347	0.0718983
[85,Inf[	0.05522	0.0682939

- If the weight table contains rows where the observed weight is zero and the standard weight is not, the individual weights will not work properly.
 - If `is.null(collapse_stratum_col_nms)`, a warning is thrown.
 - Else we combine strata identified by `collapse_stratum_col_nms` to ensure there are zero problematic rows in the weight table.
- At this point the weight table looks like e.g. (`weight_standard` and `weight_observed` are the same for the 2nd and 3rd rows because they were combined — but we keep the original strata because we need them for the join later)

ag	weight_standard	weight_observed
[0,15[	0.00000	0.0000000
[15,20[	0.00379	0.0004110
[20,25[	0.00379	0.0004110
[25,30[	0.00416	0.0030037
[30,35[	0.00842	0.0092639
[35,40[	0.01874	0.0174213
[40,45[	0.03489	0.0317440
[45,50[	0.04906	0.0641520
[50,55[	0.07094	0.0962755
[55,60[	0.09641	0.1094916
[60,65[	0.13359	0.1531554
[65,70[	0.14234	0.1582459
[70,75[	0.14766	0.1212217
[75,80[	0.14131	0.0954218
[80,85[	0.09347	0.0718983
[85,Inf[	0.05522	0.0682939

- Compute the individual weights as the standard weights divided by the observed weights. E.g. one stratum has $\text{weight_brenner} = 0.5 / 0.4 = 1.2$. If either the standard or observed weight is zero then set the Brenner weight to zero.
- At this point the table looks like e.g. (note: `weight_brenner` was made larger by our combining the strata in rows 2 and 3. In fact it will always be larger when we combine strata because `weight_standard` is summed and becomes greater but `weight_observed` does not become any greater. The observations here serve "double duty" and represent all the strata in the combination.):

ag	weight_standard	weight_observed	weight_brenner
[0,15[	0.00000	0.0000000	0.0000000
[15,20[	0.00379	0.0004110	9.2214110
[20,25[	0.00379	0.0004110	9.2214110
[25,30[	0.00416	0.0030037	1.3849735
[30,35[	0.00842	0.0092639	0.9089002
[35,40[	0.01874	0.0174213	1.0756964
[40,45[	0.03489	0.0317440	1.0991045
[45,50[	0.04906	0.0641520	0.7647460
[50,55[	0.07094	0.0962755	0.7368441
[55,60[	0.09641	0.1094916	0.8805243
[60,65[	0.13359	0.1531554	0.8722511
[65,70[	0.14234	0.1582459	0.8994864
[70,75[	0.14766	0.1212217	1.2180987
[75,80[	0.14131	0.0954218	1.4808988
[80,85[	0.09347	0.0718983	1.3000304
[85,Inf[	0.05522	0.0682939	0.8085640

popEpi::surv_individual_weights

Produce a vector of weights, one weight for each row in lexis. These weights have been called individual weights, Brenner weights, (Brenner et al 2004, <https://doi.org/10.1016/j.ejca.2004.07.007>), pre-weights, and maybe even others. A beloved child has many names.

The idea of these weights is to weigh the individual contribution of each person to the summary statistics from which the survival function estimates themselves are produced. They pronounce the influence of under-represented data and reduce the influence of over-represented data just as the more conventional computation of weighted averages survival function estimates does. However, in this individual weighting approach even a single event can be included in the summary statistics as e.g. 0.80 or 1.20 events for over and under-represented data respectively.

This function makes producing these weights easier. It performs the following steps:

- If `is.null(observed_weight_dt)`, `observed_weight_dt` is computed by calling `surv_observed_weight_dt`. In this case `lexis` is mandatory. We also pass `subset` and `breaks`. `aggre_by` is supplied as the stratum columns in `standard_weight_dt`.
- Call `popEpi::surv_brenner_weight_dt` to form Brenner weights.
- Using left-join, produce a vector of length `nrow(lexis)` where each row in `lexis` gets an individual weight based on its stratum.
- Return a vector of weights of length `nrow(lexis)`.

popEpi::surv_lexis

Compute survival estimates on a Lexis dataset (`[Epi::Lexis] / [Lexis_dt]`).

Performs the following steps:

- `estimators` is analysed and the following will be appended to `aggre_exprs` arg of `[lexis_split_merge_aggregate_by surv_lexis]`:
 - Detect variables used in the estimation expressions with `[all.vars]`.

- Retain only those known to popEpi — see ?lexis_split_merge_aggregate_by_stratum.
This results in a character string vector that the argument `aggre_exprs` of `lexis_split_merge_aggregate_by_stratum` accepts.
 - This results in `aggre_exprs` with both anything that the user defined and also what was added by `surv_lexis`. However, we drop duplicates in `aggre_exprs` based on both `duplicated(names(aggre_exprs))` and `duplicated(aggre_exprs)`. E.g. if you supply `aggre_exprs = list(n_events = quote(sum(lex.Xst != 0)))` and `n_events` is also added by `surv_lexis` then only the one you supplied is retained.
 - Call `lexis_split_merge_aggregate_by_stratum`. The resulting table of aggregated data is stratified by both `aggre_by` and by any stratifying columns found in `weight_dt` if a `data.table` was supplied as that = "sex" and `weight_dt = data.table(ag = 1:3, weight = c(100, 150, 200))`, the statistics table is stratified by `aggre_by = "sex"` and `weights = "individual_weight"` the table is stratified by sex and contains individually weighted statistics.
 - Add `surv_estimate_optional_args` to a list of internally defined arguments. The arguments passed this way override internally defined ones.
 - Call `surv_estimate`.
- If Brenner weighting was used, rename columns containing estimates, standard error, and confidence bounds to contain the identifier `_bw`.

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)
# popEpi::surv_estimate
dt <- data.table::data.table(
  box_id = 1:60,
  ts_fut_id = 1:60,
  ts_fut_start = seq(0, 5 - 1 / 12, 1 / 12),
  ts_fut_stop = seq(1 / 12, 5, 1 / 12),
  n_events = rpois(n = 60, lambda = 60:1),
  t_at_risk = rpois(n = 60, lambda = 300:240)
)
data.table::set(
  x = dt,
  j = "my_h_ch",
  value = dt[["n_events"]] / dt[["t_at_risk"]]
)
sdt <- popEpi::surv_estimate(
  dt = dt,
  ts_fut_col_nm = "ts_fut",
  value_col_nms = c("n_events", "t_at_risk"),
  estimators = c("h_ch", "S_ch"),
  conf_methods = "log-log"
)
stopifnot(
  c("h_ch", "S_ch") %in% names(sdt),
```

```

    dt[["my_h_ch"]] == sdt[["h_ch"]]
  )
  sdt <- popEpi::surv_estimate(
    dt = dt,
    ts_fut_col_nm = "ts_fut",
    value_col_nms = c("n_events", "t_at_risk"),
    estimators = list(
      "h_ch",
      my_surv = list(
        est = quote(
          exp(-cumsum((ts_fut_stop - ts_fut_start) * h_ch))
        ),
        se = quote(rep(0.0, length(ts_fut_start)))
      )
    ),
    conf_methods = c("log", "none")
  )
  stopifnot(
    c("h_ch", "my_surv") %in% names(sdt)
  )
  sdt <- popEpi::surv_estimate(
    dt = dt,
    ts_fut_col_nm = "ts_fut",
    value_col_nms = c("n_events", "t_at_risk"),
    estimators = list(
      "h_ch",
      "S_ch"
    ),
    conf_methods = list(
      "log",
      list(
        g = quote(stats::qnorm(p = theta)),
        g_inv = quote(stats::pnorm(q = g))
      )
    )
  )
  stopifnot(
    c("h_ch", "S_ch") %in% names(sdt)
  )
  # one approach to brenner weighting --- effectively the same as assigning
  # a weight for each individual using strata.
  make_column_ag_icss <- function(age) {
    cut(
      age,
      breaks = c(0, 60, 70, 80, Inf),
      right = FALSE,
      labels = c("0-59", "60-69", "70-79", "80+")
    )
  }
  make_sire <- function() {
    sire <- popEpi::sire
    sire <- sire[
      sire[["dg_date"]] < sire[["ex_date"]] &

```

```

      data.table::between(
        sire[["ex_date"]],
        as.Date("1999-01-01"),
        as.Date("2003-12-31"),
        incbounds = TRUE
      ) &
      (get.yrs(sire[["ex_date"]]) - get.yrs(sire[["bi_date"]])) < 100
    ]
    sire[j = "my_stratum" := sample(2L, size = nrow(sire), replace = TRUE)]
    sire <- sire[
      j = .SD[as.integer(seq(1L, .N, length.out = 50L))],
      keyby = "my_stratum"
    ]
    # you can also use popEpi::Lexis_dt
    sire <- Epi::Lexis(
      entry = list(
        ts_cal = popEpi::get.yrs(dg_date),
        ts_age = popEpi::get.yrs(dg_date) - popEpi::get.yrs(bi_date),
        ts_fut = 0.0
      ),
      duration = popEpi::get.yrs(ex_date) - popEpi::get.yrs(dg_date),
      entry.status = 0L,
      exit.status = status,
      data = sire
    )
    sire[["ag_icss"]] <- make_column_ag_icss(sire[["dg_age"]])
    sire
  }
  sire <- make_sire()
  make_weight_dt <- function() {
    wdt <- popEpi::ICSS[
      j = list(
        standard_weight = as.double(sum(.SD[["ICSS1"]]))
      ),
      keyby = list(
        ag_icss = make_column_ag_icss(popEpi::ICSS[["age"]])
      )
    ]
    wdt[
      j = "standard_weight" := wdt[["standard_weight"]] /
        sum(wdt[["standard_weight"]])
    ]
    owdt <- data.table::setDT(as.list(sire))[
      j = list(observed_weight = .N),
      keyby = "ag_icss"
    ]
    owdt[
      j = "observed_weight" := owdt[["observed_weight"]] /
        sum(owdt[["observed_weight"]])
    ]
    wdt[
      j = "observed_weight" := owdt[["observed_weight"]]
    ]
  }

```

```

    wdt[
      j = "brenner_weight" := wdt[["standard_weight"]] / owdt[["observed_weight"]]
    ]
    wdt[i = is.na(wdt[["brenner_weight"]]), j = "brenner_weight" := 0.0]
    return(wdt[])
  }
  wdt <- make_weight_dt()
  sdt <- popEpi::lexis_split_merge_aggregate_by_stratum(
    lexis = sire,
    breaks = list(ts_fut = seq(0, 5, 1 / 12)),
    aggre_exprs = c("t_at_risk", "n_events"),
    aggre_by = "ag_icss"
  )
  sdt_bw <- popEpi::surv_estimate(
    dt = sdt,
    ts_fut_col_nm = "ts_fut",
    estimators = c("S_ch_bw", "S_ch"),
    weight_dt = data.table::data.table(
      ag_icss = wdt[["ag_icss"]],
      brenner_weight = wdt[["brenner_weight"]]
    ),
    value_col_nms = c("n_events", "t_at_risk")
  )
  stopifnot(
    c("S_ch_bw", "S_ch") %in% names(sdt_bw),
    !"ag_icss" %in% names(sdt_bw)
  )
  # direct adjusting for comparison
  sdt_da <- popEpi::surv_estimate(
    dt = sdt,
    ts_fut_col_nm = "ts_fut",
    estimators = "S_ch",
    weight_dt = data.table::data.table(
      ag_icss = wdt[["ag_icss"]],
      weight = wdt[["standard_weight"]]
    ),
    value_col_nms = c("n_events", "t_at_risk")
  )
  stopifnot("S_ch_da" %in% names(sdt_da), !"ag_icss" %in% names(sdt_da))
  stopifnot(
    max(abs(sdt_bw[["S_ch_bw"]] - sdt_da[["S_ch_da"]])) < 0.01
  )
  # automatic handling of empty intervals
  sdt1 <- popEpi::surv_estimate(
    dt = data.table::data.table(
      box_id = 1:2,
      ts_fut_id = 1:2,
      ts_fut_start = c(0.0, 1.0),
      ts_fut_stop = c(1.0, 2.0),
      t_at_risk = c(10.0, 0.0),
      n_events = c(1, 0)
    ),
    estimators = "S_ch",

```

```

    ts_fut_col_nm = "ts_fut",
    empty_interval_action = "surv_collapse_ts_1d"
  )
  stopifnot(nrow(sdt1) == 1)
  sdt2 <- popEpi::surv_estimate(
    dt = data.table::data.table(
      box_id = 1L,
      ts_fut_id = 1L,
      ts_fut_start = 0.0,
      ts_fut_stop = 2.0,
      t_at_risk = 10.0,
      n_events = 1
    ),
    estimators = "S_ch",
    ts_fut_col_nm = "ts_fut"
  )
  stopifnot(
    nrow(sdt2) == 1,
    all.equal(sdt1[["S_ch"]], sdt2[["S_ch"]]),
    all.equal(sdt1[["S_ch_se"]], sdt2[["S_ch_se"]])
  )

  # this data.table::setDTthreads call is included here only to
  # conform to the CRAN submission requirement to only use at most 2
  # threads. you do not need to set this to use popEpi.
  # however some long calculations may benefit from using more threads.
  data.table::setDTthreads(2L)

  # popEpi::surv_collapse_ts_1d
  sdt <- data.table::data.table(
    box_id = 1:3,
    ts_fut_id = 1:3,
    ts_fut_start = 0:2,
    ts_fut_stop = 1:3,
    t_at_risk = c(1.0, 0.0, 0.5),
    n_events = 0L
  )
  sdt <- surv_collapse_ts_1d(
    dt = sdt,
    ts_col_nm = "ts_fut",
    value_col_nms = c("t_at_risk", "n_events"),
    test_expr = quote(t_at_risk > 0.0)
  )
  stopifnot(
    nrow(sdt) == 2,
    sdt[["ts_fut_start"]] == c(0.0, 1.0),
    sdt[["ts_fut_stop"]] == c(1.0, 3.0),
    sdt[["t_at_risk"]] == c(1.0, 0.5)
  )

  # table with multiple time scales and we collapse along one of them
  dt <- local({
    dt <- data.table::data.table(

```

```

      ts_age_id = 1:4,
      ts_age_start = c(0, 40, 50, 60),
      ts_age_stop = c(40, 50, 60, Inf)
    )
    dt_fut <- data.table::data.table(
      ts_fut_id = 1:5,
      ts_fut_start = 0:4,
      ts_fut_stop = 1:5
    )
    idx_df <- merge(seq_len(nrow(dt)), seq_len(nrow(dt_fut)))
    dt <- cbind(dt[idx_df[[1]]], dt_fut[idx_df[[2]]])
    data.table::set(
      x = dt,
      j = c("box_id", "t_at_risk"),
      value = list(seq_len(nrow(dt)), c(0.0, rep(100.0, nrow(dt) - 1L)))
    )
    dt[]
  })
  dt_collapsed <- popEpi::surv_collapse_ts_1d(
    dt = dt,
    ts_col_nm = "ts_age",
    stratum_col_nms = c("ts_fut_id", "ts_fut_start", "ts_fut_stop"),
    value_col_nms = "t_at_risk",
    test_expr = quote(sum(t_at_risk) > 10.0)
  )
  stopifnot(
    nrow(dt_collapsed) == nrow(dt) - 1L,
    dt[["ts_age_stop"]][1] == 40,
    dt_collapsed[["ts_age_stop"]][1] == 50
  )
  # this data.table::setDTthreads call is included here only to
  # conform to the CRAN submission requirement to only use at most 2
  # threads. you do not need to set this to use popEpi.
  # however some long calculations may benefit from using more threads.
  data.table::setDTthreads(2L)

  # popEpi::surv_collapse_strata_list
  sdt <- data.table::CJ(
    ag = 1:3,
    box_id = 1:5
  )
  box_dt <- data.table::data.table(
    box_id = 1:5,
    ts_fut_id = 1:5,
    ts_fut_start = as.numeric(0:4),
    ts_fut_stop = as.numeric(1:5)
  )
  sdt <- box_dt[sdt, on = "box_id"]
  sdt[
    i = sdt[["ag"]] == 1,
    j = "t_at_risk" := 0.0
  ]
  sdt[

```

```

    i = sdt[["ag"]] == 2,
    j = "t_at_risk" := c(1, 1, 0, 2, 1)
  ]
  sdt[
    i = sdt[["ag"]] == 3,
    j = "t_at_risk" := c(1, 1, 1, 0, 1)
  ]
  sdt_collapse_data <- popEpi::surv_collapse_strata_list(
    dt = sdt,
    stratum_col_nms = "ag",
    collapse_stratum_col_nms = "ag"
  )
  stopifnot(
    "result" %in% names(sdt_collapse_data),
    inherits(sdt_collapse_data[["result"]][[1]], "list"),
    c("new", "old") %in% names(sdt_collapse_data[["result"]][[1]]),
    sdt_collapse_data[["result"]][[1]][["new"]][["stratum_id"]] == 1L
  )
  sdt[
    i = sdt[["ag"]] == 1,
    j = "t_at_risk" := 0.5
  ]
  sdt_collapse_data <- popEpi::surv_collapse_strata_list(
    dt = sdt,
    stratum_col_nms = "ag",
    collapse_stratum_col_nms = "ag"
  )
  stopifnot(
    "result" %in% names(sdt_collapse_data),
    inherits(sdt_collapse_data[["result"]][[1]], "list"),
    c("new", "old") %in% names(sdt_collapse_data[["result"]][[1]]),
    identical(
      sdt_collapse_data[["result"]][[1]][["new"]][["stratum_id"]],
      c(rep(1L, 5L), rep(2L, 10L))
    )
  )
  # popEpi::surv_collapse_strata_1d
  sdt_collapsed <- popEpi::surv_collapse_strata_1d(
    dt = sdt,
    stratum_col_nms = "ag",
    collapse_stratum_col_nm = "ag"
  )
  stopifnot(
    nrow(sdt_collapsed) == 10,
    names(sdt) %in% names(sdt_collapsed),
    names(sdt_collapsed) %in% names(sdt),
    is.character(sdt_collapsed[["ag"]]),
    identical(unique(sdt_collapsed[["ag"]]), c("1", "2 & 3"))
  )

# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.

```

```

# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)

# popEpi::surv_brenner_weight_dt
make_column_icss_ag <- function(age) {
  cut(
    age,
    breaks = c(seq(15, 85, 5), Inf),
    right = FALSE
  )
}

make_standard_weight_dt <- function() {
  swdt <- popEpi::ICSS[
    j = list(
      weight = as.double(sum(.SD[["ICSS1"]]))
    ),
    keyby = list(
      icss_ag = make_column_icss_ag(popEpi::ICSS[["age"]])
    )
  ][
    swdt <- swdt[!is.na(swdt[["icss_ag"]]), ]
    return(swdt[])
  ]

  swdt <- make_standard_weight_dt()
  owdt <- data.table::setDT(list(
    icss_ag = make_column_icss_ag(c(50.5, 60.5))
  ))[
    i = data.table::setDT(list(icss_ag = swdt[["icss_ag"]])),
    on = "icss_ag",
    j = list(weight = .N),
    keyby = .EACHI
  ]

  # this is what happens when at least one weighting stratum is empty
  warn_env <- new.env()
  warn_env[["w"]] <- NULL
  bwdt_bad <- suppressWarnings(withCallingHandlers(
    popEpi::surv_brenner_weight_dt(
      standard_weight_dt = swdt,
      observed_weight_dt = owdt
    ),
    warning = function(w) warn_env[["w"]] <- w
  ))
  stopifnot(
    inherits(warn_env[["w"]], "warning"),
    bwdt_bad[["weight_brenner"]] >= 0
  )

  # here is an easy way to handle this problem. note that the weights are
  # always larger when we need to aggregate strata --- because the standard
  # weight increases but the observed weight remains the same. the same

```

```

# observations serve "double duty" now for multiple original strata.
bwdt <- popEpi::surv_brenner_weight_dt(
  standard_weight_dt = swdt,
  observed_weight_dt = owdt,
  collapse_stratum_col_nms = "icss_ag"
)
stopifnot(
  nrow(swdt) == nrow(bwdt),
  bwdt[["weight_brenner"]] > 0
)
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)

# popEpi::surv_individual_weights
make_column_icss_ag <- function(age) {
  cut(
    age,
    breaks = c(seq(15, 85, 5), Inf),
    right = FALSE
  )
}

make_standard_weight_dt <- function() {
  swdt <- popEpi::ICSS[
    j = list(
      weight = as.double(sum(.SD[["ICSS1"]]))
    ),
    keyby = list(
      icss_ag = make_column_icss_ag(popEpi::ICSS[["age"]])
    )
  ][
    ][]
  return(swdt[])
}

lexis <- Epi::Lexis(
  entry = list(ts_fut = c(0.0, 0.0)),
  exit = list(ts_fut = c(1.5, 2.5)),
  entry.status = 0L,
  exit.status = 0L
)
lexis[["icss_ag"]] <- make_column_icss_ag(c(50.5, 60.5))
swdt <- make_standard_weight_dt()

# this is what happens when at least one weighting stratum is empty
warn_env <- new.env()
warn_env[["w"]] <- NULL
iw_bad <- suppressWarnings(withCallingHandlers(popEpi::surv_individual_weights(
  lexis = lexis,
  standard_weight_dt = swdt
), warning = function(w) warn_env[["w"]] <- w))

```

```

stopifnot(
  inherits(warn_env[["w"]], "warning"),
  iw_bad >= 0
)

# here is an easy way to handle this problem. note that the weights are
# always larger when we need to aggregate strata --- because the standard
# weight increases but the observed weight remains the same. the same
# observations serve "double duty" now for multiple original strata.
iw <- popEpi::surv_individual_weights(
  lexis = lexis,
  standard_weight_dt = swdt,
  collapse_stratum_col_nms = "icss_ag"
)
stopifnot(
  length(iw) == nrow(lexis),
  iw >= 0.0,
  iw >= iw_bad
)
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)

# popEpi::lexis_breaks_collapse_1d
lexis <- Epi::Lexis(
  entry = list(ts_fut = c(0.0, 3.0)),
  duration = c(2.5, 10.0),
  entry.status = 0L,
  exit.status = c(0L, 1L)
)
bl <- list(ts_fut = seq(0, 5, 0.5))
agdt <- popEpi::lexis_split_merge_aggregate_by_stratum(
  lexis = lexis,
  breaks = bl,
  aggre_exprs = "t_at_risk"
)
# what to do about empty intervals? we have intervals with no subjects in them.
stopifnot(
  agdt[["t_at_risk"]][-6] > 0,
  agdt[["t_at_risk"]][6] == 0.0
)
# combine intervals until there are no empty ones left.
breaks_ts_fut <- popEpi::lexis_breaks_collapse_1d(
  lexis = lexis,
  breaks_1d = bl
)
# this has resulted in the combined interval ]2.5, 3.5].
stopifnot(!3.0 %in% breaks_ts_fut)

# but sometimes we want to retain specific breaks in the output. for instance
# if the goal is to produce survival estimates for every year.

```

```

# we can make use of arg `mandatory_breaks` to achieve this.
breaks_ts_fut <- popEpi::lexis_breaks_collapse_1d(
  lexis = lexis,
  breaks_1d = bl,
  mandatory_breaks = 0:5
)
# this has resulted in the combined interval ]2.0, 3.0].
stopifnot(!2.5 %in% breaks_ts_fut)
agdt <- popEpi::lexis_split_merge_aggregate_by_stratum(
  lexis = lexis,
  breaks = list(ts_fut = breaks_ts_fut),
  aggre_exprs = "t_at_risk"
)
stopifnot(
  agdt[["t_at_risk"]] > 0.0
)

# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)

# popEpi::surv_lexis
make_pm <- function() {
  pm <- data.table::copy(popEpi::popmort)
  data.table::setnames(
    pm,
    c("year", "agegroup", "haz"),
    c("ts_cal", "ts_age", "h_exp")
  )
  data.table::setkeyv(pm, c("sex", "ts_cal", "ts_age"))
  pm <- c(
    list(pm),
    lapply(101:110, function(age) {
      sub_pm <- pm[
        pm[["ts_age"]] == 100
      ]
      sub_pm[
        j = "ts_age" := age
      ]
      sub_pm[]
    })
  )
  pm <- data.table::rbindlist(pm)
  data.table::setkeyv(pm, c("sex", "ts_cal", "ts_age"))
  return(pm[])
}

make_column_icss_ag <- function(age) {
  cut(
    age,
    breaks = c(0, 60, 70, 80, Inf),

```

```

    right = FALSE,
    labels = c("0-59", "60-69", "70-79", "80+")
  )
}

make_standard_weight_dt <- function() {
  return(popEpi::ICSS[
    j = list(
      weight = as.double(sum(.SD[["ICSS1"]]))
    ),
    keyby = list(
      icss_ag = make_column_icss_ag(popEpi::ICSS[["age"]])
    )
  ][])
}

make_sire <- function() {
  sire <- popEpi::sire
  sire <- sire[
    sire[["dg_date"]] < sire[["ex_date"]] &
    sire[["ex_date"]] >= as.Date("1999-01-01") &
    (get.yrs(sire[["ex_date"]]) - get.yrs(sire[["bi_date"]])) < 100
  ]
  set.seed(1337)
  sire[j = "my_stratum" := sample(2L, size = .N, replace = TRUE)]
  # you can also use popEpi::Lexis_dt
  sire <- Epi::Lexis(
    data = sire,
    entry = list(
      ts_cal = popEpi::get.yrs(dg_date),
      ts_age = popEpi::get.yrs(dg_date) - popEpi::get.yrs(bi_date),
      ts_fut = 0.0
    ),
    duration = popEpi::get.yrs(ex_date) - popEpi::get.yrs(dg_date),
    entry.status = 0L,
    exit.status = status
  )
  sire[["icss_ag"]] <- make_column_icss_ag(sire[["dg_age"]])
  return(sire[])
}

pm <- make_pm()
wdt <- make_standard_weight_dt()
sire <- make_sire()

# using breaks on the calendar time causes period analysis to be performed.
# this is done here for demonstration purposes but of course everything works
# also without ts_cal breaks.
bl <- list(
  ts_cal = c(1999, 2004),
  ts_fut = popEpi::lexis_breaks_collapse_1d(
    lexis = sire,
    breaks_1d = list(ts_fut = seq(0, 5, 1 / 12)),

```

```

    mandatory_breaks = 0:5
  )
)

# observed survival
sdt <- popEpi::surv_lexis(
  lexis = sire,
  breaks = bl,
  aggre_by = "my_stratum",
  subset = NULL,
  estimators = "S_ch",
  conf_methods = "log",
  conf_lvls = 0.95
)
stopifnot(
  inherits(sdt, "data.table"),
  "my_stratum" %in% names(sdt),
  c("S_ch", "S_ch_se", "S_ch_lo", "S_ch_hi") %in% names(sdt)
)

# observed survival with direct adjusting
sdt_da <- popEpi::surv_lexis(
  lexis = sire,
  breaks = bl,
  aggre_by = "my_stratum",
  estimators = "S_ch",
  weights = wdt
)
stopifnot(
  inherits(sdt_da, "data.table"),
  "my_stratum" %in% names(sdt_da),
  c("S_ch_da", "S_ch_da_se", "S_ch_da_lo", "S_ch_da_hi") %in% names(sdt_da)
)

# observed survival with individual weighting
sire[["individual_weight"]] <- popEpi::surv_individual_weights(
  lexis = sire,
  standard_weight_dt = wdt,
  breaks = bl["ts_cal"]
)
sdt_iw <- popEpi::surv_lexis(
  lexis = sire,
  breaks = bl,
  aggre_by = "my_stratum",
  estimators = c("S_ch_bw", "S_ch"),
  weights = "individual_weight"
)
stopifnot(
  inherits(sdt_iw, "data.table"),
  "my_stratum" %in% names(sdt_iw),
  c("S_ch_bw", "S_ch_bw_se", "S_ch_bw_lo", "S_ch_bw_hi") %in% names(sdt_iw),

  # direct adjusting and individual weighting produce similar results.

```

```

# the larger the dataset, the smaller the difference between the two.
max(abs(sdt_da[["S_ch_da"]] - sdt_iw[["S_ch_bw"]])) < 0.012,
max(abs(sdt_da[["S_ch_da_se"]] - sdt_iw[["S_ch_bw_se"]])) < 0.001
)

# observed survival with brenner weighting applied on the aggregated data.
# the result is identical to individual weighting.
wdt_obs <- popEpi::surv_observed_weight_dt(
  lexis = sire,
  aggre_by = data.table::setDT(list(icss_ag = wdt[["icss_ag"]])),
  breaks = bl["ts_cal"]
)
wdt_bw <- popEpi::surv_brenner_weight_dt(
  standard_weight_dt = wdt,
  observed_weight_dt = wdt_obs
)
wdt_bw <- data.table::setDT(list(
  icss_ag = wdt_bw[["icss_ag"]],
  brenner_weight = wdt_bw[["weight_brenner"]]
))
sdt_bw <- popEpi::surv_lexis(
  lexis = sire,
  breaks = bl,
  aggre_by = "my_stratum",
  estimators = c("S_ch_bw", "S_ch"),
  weights = wdt_bw
)
stopifnot(
  inherits(sdt_bw, "data.table"),
  "my_stratum" %in% names(sdt_bw),
  c("S_ch_bw", "S_ch_bw_se", "S_ch_bw_lo", "S_ch_bw_hi") %in% names(sdt_bw),

  all.equal(sdt_bw[["t_at_risk_bw"]], sdt_iw[["t_at_risk_bw"]]),
  all.equal(sdt_bw[["n_events_bw"]], sdt_iw[["n_events_bw"]]),
  all.equal(sdt_bw[["S_ch_bw"]], sdt_iw[["S_ch_bw"]]),
  all.equal(sdt_bw[["S_ch_bw_se"]], sdt_iw[["S_ch_bw_se"]])
)

# observed survival with direct adjusting, multiple period estimates
# with the periods as strata in output
bl_multi_period <- bl
bl_multi_period[["ts_cal"]] <- c(bl[["ts_cal"]], max(bl[["ts_cal"]]) + 5)
sdt_da <- popEpi::surv_lexis(
  lexis = sire,
  breaks = bl_multi_period,
  aggre_by = "my_stratum",
  estimators = "S_ch",
  weights = wdt
)
stopifnot(
  inherits(sdt_da, "data.table"),
  c("ts_cal_start", "ts_fut_start") %in% names(sdt_da),
  c("S_ch_da", "S_ch_da_se", "S_ch_da_lo", "S_ch_da_hi") %in% names(sdt_da)
)

```

```

)

# observed survival with individual adjusting, period estimates
sdt_iw <- popEpi::surv_lexis(
  lexis = sire,
  breaks = bl_multi_period,
  aggre_by = "my_stratum",
  estimators = "S_ch_bw",
  weights = "individual_weight"
)
stopifnot(
  inherits(sdt_iw, "data.table"),
  c("ts_cal_start", "ts_fut_start") %in% names(sdt_iw),
  c("S_ch_bw", "S_ch_bw_se", "S_ch_bw_lo", "S_ch_bw_hi") %in% names(sdt_iw),

  # direct adjusting and individual weighting produce similar results.
  # the larger the dataset, the smaller the difference between the two.
  max(abs(sdt_da[["S_ch_da"]] - sdt_iw[["S_ch_bw"]])) < 0.03,
  max(abs(sdt_da[["S_ch_da_se"]] - sdt_iw[["S_ch_bw_se"]])) < 0.006
)

# a few common survival time function estimates
sdt <- popEpi::surv_lexis(
  lexis = sire,
  breaks = bl,
  merge_dt_by = c("sex", "ts_cal", "ts_age"),
  merge_dt = pm,
  aggre_by = "my_stratum",
  estimators = c(
    "S_ch",
    "RS_e2_ch",
    "NS_pp_ch"
  )
)
stopifnot(
  inherits(sdt, "data.table"),
  c("S_ch", "RS_e2_ch", "NS_pp_ch") %in% names(sdt)
)

# your very own estimator (the example `se` is made up)
sdt <- popEpi::surv_lexis(
  lexis = sire,
  breaks = bl,
  merge_dt_by = c("sex", "ts_cal", "ts_age"),
  merge_dt = pm,
  aggre_by = "my_stratum",
  estimators = list(
    my_estimator = list(
      est = quote(n_events / n_events_exp_e2),
      se = quote(sqrt(n_events / (n_events_exp_e2^2)))
    )
  ),
  conf_method = "identity"
)

```

```

)
stopifnot(
  c("my_estimator", "my_estimator_se") %in% names(sdt)
)

# your very own estimator with direct adjusting
sdt <- popEpi::surv_lexis(
  lexis = sire,
  breaks = bl,
  merge_dt_by = c("sex", "ts_cal", "ts_age"),
  merge_dt = pm,
  aggre_by = "my_stratum",
  estimators = list(
    my_estimator = list(
      est = quote(n_events / n_events_exp_e2),
      se = quote(sqrt(n_events / (n_events_exp_e2^2)))
    )
  ),
  conf_method = "identity",
  weights = wdt
)
stopifnot(
  c("my_estimator_da", "my_estimator_da_se") %in% names(sdt)
)

# your very own estimator with individual brenner weighting
sdt <- popEpi::surv_lexis(
  lexis = sire,
  breaks = bl,
  merge_dt_by = c("sex", "ts_cal", "ts_age"),
  merge_dt = pm,
  aggre_by = "my_stratum",
  estimators = list(
    my_estimator = list(
      est = quote(n_events / n_events_exp_e2),
      se = quote(sqrt(n_events / (n_events_exp_e2^2)))
    ),
    my_estimator_bw = list(
      est = quote(n_events_bw / n_events_exp_e2_bw),
      se = quote(sqrt(n_events_bw / (n_events_exp_e2_bw^2)))
    )
  ),
  conf_method = "identity",
  weights = "individual_weight"
)
stopifnot(
  c("my_estimator_bw", "my_estimator_bw_se") %in% names(sdt)
)

# your very own method of confidence interval estimation
sdt <- popEpi::surv_lexis(
  lexis = sire,
  breaks = bl,

```

```

estimators = c("S_ch_bw", "S_lt_bw"),
conf_methods = list(list(
  g = quote(qnorm(theta)),
  g_inv = quote(pnorm(g))
)),
conf_lvls = 0.95,
weights = "individual_weight"
)
stopifnot(
  c("S_ch_bw_lo", "S_ch_bw_hi") %in% names(sdt)
)

# a bunch of estimators, cause-specific ones from state 0 to 1 or 2
# and non-cause-specific ones also
sdt <- popEpi::surv_lexis(
  lexis = sire,
  breaks = bl,
  merge_dt_by = c("sex", "ts_cal", "ts_age"),
  merge_dt = pm,
  aggre_by = NULL,
  subset = NULL,
  estimators = c(
    "h_ch_[0, 1]",
    "h_ch_[0, 2]",
    "H_ch_[0, 1]",
    "H_ch_[0, 2]",
    "F_ch_[0, 1]",
    "F_ch_[0, 2]",
    "S_ch_[0, 1]",
    "S_ch_[0, 2]",
    "S_ch_[0, 1:2]",
    "H_ch",
    "H_lt",
    "h_ch",
    "h_lt",
    "F_ch",
    "F_lt",
    "S_ch",
    "S_lt",
    "NS_pp_ch",
    "RF_e2_ch"
  )
)
stopifnot(
  # hazard is additive.
  all.equal(
    sdt[["h_ch"]],
    sdt[["h_ch_[0, 1]"]] + sdt[["h_ch_[0, 2]"]]
  ),
  # cumulative hazard is additive.
  all.equal(
    sdt[["H_ch"]],
    sdt[["H_ch_[0, 1]"]] + sdt[["H_ch_[0, 2]"]]
  )
)

```

```

),
# competing risks cdf is additive. in epidemiology the cdf is often called
# the cumulative incidence function.
all.equal(
  sdt[["F_ch"]],
  sdt[["F_ch_[0, 1]"]] + sdt[["F_ch_[0, 2]"]]
),

# competing risks survival is NOT additive!
sdt[["S_ch_[0, 1]"]] + sdt[["S_ch_[0, 2]"]] > 1,

# you can define multiple states in the cause-specific transition.
# this one covers all exit statuses and is therefore the same as overall
# survival by definition.
sdt[["S_ch_[0, 1:2]"]] == sdt[["S_ch"]] #,

# the different estimation methods produce about the same results.
# abs(sdt[["S_ch"]] - sdt[["S_lt"]]) < 0.01,
# abs(sdt[["H_ch"]] - sdt[["H_lt"]]) < 0.01,

# net survival is approximately the same as cause-specific survival in this
# rather ideal dataset.
# abs(sdt[["S_ch_[0, 1]"]] - sdt[["NS_pp_ch"]]) < 0.01,
# the ederer 2 based "relative cdf" is approximately the same as
# cause-specific cdf in this rather ideal dataset.
# abs(sdt[["F_ch_[0, 1]"]] - sdt[["RF_e2_ch"]]) < 0.01
)

# factor status
data.table::set(
  x = sire,
  j = c("lex.Cst", "lex.Xst"),
  value = list(
    factor(
      x = sire[["lex.Cst"]],
      levels = 0:2,
      labels = c("alive", "dead from cancer", "dead from other")
    ),
    factor(
      x = sire[["lex.Xst"]],
      levels = 0:2,
      labels = c("alive", "dead from cancer", "dead from other")
    )
  )
)

sdt <- popEpi::surv_lexis(
  lexis = sire,
  breaks = bl,
  merge_dt_by = c("sex", "ts_cal", "ts_age"),
  merge_dt = pm,
  aggre_by = NULL,
  subset = NULL,
  estimators = c(

```

```

      "F_ch_['alive', 'dead from cancer']",
      "F_ch_['alive', 'dead from other']"
    )
  )
  stopifnot(
    # produces the estimates as intended. maybe these are not too pretty but
    # you could use shorter factor labels such as c("a", "dc", "do").
    "F_ch_['alive', 'dead from cancer']" %in% names(sdt)
  )

  # TRUE/FALSE status
  data.table::set(
    x = sire,
    j = c("lex.Cst", "lex.Xst"),
    value = list(
      sire[["lex.Cst"]] != "alive",
      sire[["lex.Xst"]] != "alive"
    )
  )
  sdt <- popEpi::surv_lexis(
    lexis = sire,
    breaks = bl,
    merge_dt_by = c("sex", "ts_cal", "ts_age"),
    merge_dt = pm,
    aggre_by = NULL,
    subset = NULL,
    estimators = c(
      "F_ch",
      "F_ch_[F, T]"
    )
  )
  stopifnot(
    "F_ch_[F, T]" %in% names(sdt),

    max(abs(sdt[["F_ch_[F, T]"] - sdt[["F_ch"]])) < 1e-15
  )

  # more arguments to popEpi::lexis_split_merge_aggregate_by_stratum
  sdt <- popEpi::surv_lexis(
    lexis = sire,
    breaks = bl,
    split_merge_aggregate_optional_args = list(
      breaks_collapse_args = list(mandatory_breaks = 0:5)
    ),
    estimators = "S_ch"
  )
  stopifnot(
    1:5 %in% sdt[["ts_fut_stop"]]
  )

```

survmean

*Compute Mean Survival Times Using Extrapolation***Description**

Computes mean survival times based on survival estimation up to a point in follow-up time (e.g. 10 years), after which survival is extrapolated using an appropriate hazard data file (pophaz) to yield the "full" survival curve. The area under the full survival curve is the mean survival.

Usage

```
survmean(
  formula,
  data,
  adjust = NULL,
  weights = NULL,
  breaks = NULL,
  pophaz = NULL,
  e1.breaks = NULL,
  e1.pophaz = pophaz,
  r = "auto",
  surv.method = "hazard",
  subset = NULL,
  verbose = FALSE
)
```

Arguments

formula	a formula, e.g. <code>FUT ~ V1</code> or <code>Surv(FUT, lex.Xst) ~ V1</code> . Supplied in the same way as to <code>[survtab]</code> , see that help for more info.
data	a Lexis data set; see <code>[Epi::Lexis]</code> .
adjust	variables to adjust estimates by, e.g. <code>adjust = "agegr"</code> . Flexible input .
weights	weights to use to adjust mean survival times. See the dedicated help page for more details on weighting. <code>survmean</code> computes curves separately by all variables to adjust by, computes mean survival times, and computes weighted means of the mean survival times. See Examples.
breaks	a list of breaks defining the time window to compute observed survival in, and the intervals used in estimation. E.g. <code>list(FUT = 0:10)</code> when FUT is the follow-up time scale in your data.
pophaz	a data set of population hazards passed to <code>[survtab]</code> (see the dedicated help page and the help page of <code>survtab</code> for more information). Defines the population hazard in the time window where observed survival is estimated.
e1.breaks	NULL or a list of breaks defining the time window to compute expected survival in, and the intervals used in estimation. E.g. <code>list(FUT = 0:100)</code> when FUT is the follow-up time scale in your data to extrapolate up to 100 years from where

	the observed survival curve ends. NOTE: the breaks on the survival time scale MUST include the breaks supplied to argument breaks; see Examples. If NULL, uses decent defaults (maximum follow-up time of 50 years).
e1.pophaz	Same as pophaz, except this defines the population hazard in the time window where expected survival is estimated. By default uses the same data as argument pophaz.
r	either a numeric multiplier such as 0.995, "auto", or "autoX" where X is an integer; used to determine the relative survival ratio (RSR) persisting after where the estimated observed survival curve ends. See Details.
surv.method	passed to survtab; see that help for more info.
subset	a logical condition; e.g. subset = sex == 1; subsets the data before computations
verbose	logical; if TRUE, the function is returns some messages and results along the run, which may be useful in debugging

Details

Basics

survmean computes mean survival times. For median survival times (i.e. where 50 % of subjects have died or met some other event) use [survtab].

The mean survival time is simply the area under the survival curve. However, since full follow-up rarely happens, the observed survival curves are extrapolated using expected survival: E.g. one might compute observed survival till up to 10 years and extrapolate beyond that (till e.g. 50 years) to yield an educated guess on the full observed survival curve.

The area is computed by trapezoidal integration of the area under the curve. This function also computes the "full" expected survival curve from T = 0 till e.g. T = 50 depending on supplied arguments. The expected mean survival time is the area under the mean expected survival curve. This function returns the mean expected survival time to be compared with the mean survival time and for computing years of potential life lost (YPLL).

Results can be formed by strata and adjusted for e.g. age by using the formula argument as in survtab. See also Examples.

Extrapolation tweaks

Argument r controls the relative survival ratio (RSR) assumed to persist beyond the time window where observed survival is computed (defined by argument breaks; e.g. up to FUT = 10). The RSR is simply $RSR_i = p_{oi} / p_{ei}$ for a time interval i, i.e. the observed divided by the expected (conditional, not cumulative) probability of surviving from the beginning of a time interval till its end. The cumulative product of RSR_i over time is the (cumulative) relative survival curve.

If r is numeric, e.g. 0.995, that RSR level is assumed to persist beyond the observed survival curve. Numeric r should be > 0 and expressed at the annual level when using fractional years as the scale of the time variables. E.g. if RSR is known to be 0.95 at the month level, then the annualized RSR is 0.95^{12} . This enables correct usage of the RSR with survival intervals of varying lengths. When using day-level time variables (such as Dates; see as.Date), numeric r should be expressed at the day level, etc.

If r is "auto" or "auto1", this function computes RSR estimates internally and automatically uses the RSR_i in the last survival interval in each stratum (and adjusting group) and assumes that to

persist beyond the observed survival curve. Automatic determination of r is a good starting point, but in situations where the RSR estimate is uncertain it may produce poor results. Using "autoX" such as "auto6" causes `survmean` to use the mean of the estimated RSRs in the last X survival intervals, which may be more stable. Automatic determination will not use values >1 but set them to 1. Visual inspection of the produced curves is always recommended: see Examples.

One may also tweak the accuracy and length of extrapolation and expected survival curve computation by using `e1.breaks`. By default this is whatever was supplied to `breaks` for the survival time scale, to which

```
c(seq(1/12, 1, 1/12), seq(1.2, 1.8, 0.2), 2:19, seq(20, 50, 5))
```

is added after the maximum value, e.g. with `breaks = list(FUT = 0:10)` we have
`..., 10+1/12, ..., 11, 11.2, ..., 2, 3, ..., 19, 20, 25, ... 50`

as the `e1.breaks`. Supplying `e1.breaks` manually requires the breaks over time survival time scale supplied to argument `breaks` to be reiterated in `e1.breaks`; see Examples. **NOTE:** the default extrapolation breaks assume the time scales in the data to be expressed as fractional years, meaning this will work extremely poorly when using e.g. day-level time scales (such as `Date` variables). Set the extrapolation breaks manually in such cases.

Value

Returns a `data.frame` or `data.table` (depending on `getOptions("popEpi.datatable")`; see `?popEpi`) containing the following columns:

- `est`: The estimated mean survival time
- `exp`: The computed expected survival time
- `obs`: Counts of subjects in data
- `YPLL`: Years of Potential Life Lost, computed as $((\text{exp} - \text{est}) * \text{obs})$ — though your time data may be in e.g. days, this column will have the same name regardless. The returned data also has columns named according to the variables supplied to the right-hand-side of the formula.

Author(s)

Joonas Miettinen

See Also

Other `survmean` functions: [Surv\(\)](#), [lines.survmean\(\)](#), [plot.survmean\(\)](#)

Other main functions: [Surv\(\)](#), [rate\(\)](#), [relpois\(\)](#), [relpois_ag\(\)](#), [sir\(\)](#), [sirspline\(\)](#), [survtab\(\)](#), [survtab_ag\(\)](#)

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)

library(Epi)
```

```

## take 500 subjects randomly for demonstration
data(sire)
sire <- sire[sire$dg_date < sire$ex_date, ]
set.seed(1L)
sire <- sire[sample(x = nrow(sire), size = 500),]

## NOTE: recommended to use factor status variable
x <- Lexis(entry = list(FUT = 0, AGE = dg_age, CAL = get.yrs(dg_date)),
          exit = list(CAL = get.yrs(ex_date)),
          data = sire,
          exit.status = factor(status, levels = 0:2,
                              labels = c("alive", "canD", "othD")),
          merge = TRUE)

## phony variable
set.seed(1L)
x$group <- rbinom(nrow(x), 1, 0.5)
## age group
x$agegr <- cut(x$dg_age, c(0,45,60,Inf), right=FALSE)

## population hazards data set
pm <- data.frame(popEpi::popmort)
names(pm) <- c("sex", "CAL", "AGE", "haz")

## breaks to define observed survival estimation
BL <- list(FUT = seq(0, 10, 1/12))

## crude mean survival
sm1 <- survmean(Surv(FUT, lex.Xst != "alive") ~ 1,
               pophaz = pm, data = x, weights = NULL,
               breaks = BL)

sm1 <- survmean(FUT ~ 1,
               pophaz = pm, data = x, weights = NULL,
               breaks = BL)

## mean survival by group
sm2 <- survmean(FUT ~ group,
               pophaz = pm, data = x, weights = NULL,
               breaks = BL)

## ... and adjusted for age using internal weights (counts of subjects)
## note: need also longer extrapolation here so that all curves
## converge to zero in the end.
eBL <- list(FUT = c(BL$FUT, 11:75))
sm3 <- survmean(FUT ~ group + adjust(agegr),
               pophaz = pm, data = x, weights = "internal",
               breaks = BL, e1.breaks = eBL)

## visual inspection of how realistic extrapolation is for each stratum;
## solid lines are observed + extrapolated survivals;
## dashed lines are expected survivals
plot(sm1)

```

```

## plotting object with both stratification and standardization
## plots curves for each strata-std.group combination
plot(sm3)

## for finer control of plotting these curves, you may extract
## from the survmean object using e.g.
attributes(sm3)$survmean.meta$curves

#### using Dates

x <- Lexis(entry = list(FUT = 0L, AGE = dg_date-bi_date, CAL = dg_date),
           exit = list(CAL = ex_date),
           data = sire[sire$dg_date < sire$ex_date, ],
           exit.status = factor(status, levels = 0:2,
                                labels = c("alive", "canD", "othD")),
           merge = TRUE)
## phony group variable
set.seed(1L)
x$group <- rbinom(nrow(x), 1, 0.5)

## NOTE: population hazard should be reported at the same scale
## as time variables in your Lexis data.
data(popmort, package = "popEpi")
pm <- data.frame(popmort)
names(pm) <- c("sex", "CAL", "AGE", "haz")
## from year to day level
pm$haz <- pm$haz/365.25
pm$CAL <- as.Date(paste0(pm$CAL, "-01-01"))
pm$AGE <- pm$AGE*365.25

BL <- list(FUT = seq(0, 8, 1/12)*365.25)
eBL <- list(FUT = c(BL$FUT, c(8.25,8.5,9:60)*365.25))
smd <- survmean(FUT ~ group, data = x,
                pophaz = pm, verbose = TRUE, r = "auto5",
                breaks = BL, e1.breaks = eBL)

plot(smd)

```

survtab

Estimate Survival Time Functions

Description

This function estimates survival time functions: survival, relative/net survival, and crude/absolute risk functions (CIF).

Usage

```
survtab(
  formula,
  data,
  adjust = NULL,
  breaks = NULL,
  pophaz = NULL,
  weights = NULL,
  surv.type = "surv.rel",
  surv.method = "hazard",
  relsurv.method = "e2",
  subset = NULL,
  conf.level = 0.95,
  conf.type = "log-log",
  verbose = FALSE
)
```

Arguments

formula	a formula; e.g. <code>fot ~ sex</code> , where <code>fot</code> is the time scale over which you wish to estimate a survival time function; this assumes that <code>lex.Xst</code> in your data is the status variable in the intended format (almost always right). To be explicit, use <code>[survival::Surv]:</code> e.g. <code>Surv(fot, lex.Xst) ~ sex</code> . Variables on the right-hand side of the formula separated by <code>+</code> are considered stratifying variables, for which estimates are computed separately. May contain usage of <code>adjust()</code> — see Details and Examples.
data	a Lexis object with at least the survival time scale
adjust	can be used as an alternative to passing variables to argument <code>formula</code> within a call to <code>adjust()</code> ; e.g. <code>adjust = "agegr"</code> . Flexible input .
breaks	a named list of breaks, e.g. <code>list(FUT = 0:5)</code> . If data is not split in advance, breaks must at the very least contain a vector of breaks to split the survival time scale (mentioned in argument <code>formula</code>). If data has already been split (using e.g. <code>[splitMulti]</code>) along at least the used survival time scale, this may be <code>NULL</code> . It is generally recommended (and sufficient; see Seppa, Dyban and Hakulinen (2015)) to use monthly intervals where applicable.
pophaz	a <code>data.frame</code> containing expected hazards for the event of interest to occur. See the dedicated help page . Required when <code>surv.type = "surv.rel"</code> or <code>"cif.rel"</code> . <code>pophaz</code> must contain one column named <code>"haz"</code> , and any number of other columns identifying levels of variables to do a merge with split data within <code>survtab</code> . Some columns may be time scales, which will allow for the expected hazard to vary by e.g. calendar time and age.
weights	typically a list of weights or a character string specifying an age group standardization scheme; see the dedicated help page and examples. NOTE: <code>weights = "internal"</code> is based on the counts of persons in follow-up at the start of follow-up (typically <code>T = 0</code>)
surv.type	one of <code>'surv.obs'</code> , <code>'surv.cause'</code> , <code>'surv.rel'</code> , <code>'cif.obs'</code> or <code>'cif.rel'</code> ; defines what kind of survival time function(s) is/are estimated; see Details

<code>surv.method</code>	either 'lifetable' or 'hazard'; determines the method of calculating survival time functions, where the former computes ratios such as $p = d / (n - n.cens)$ and the latter utilizes subject-times (typically person-years) for hazard estimates such as <code>d/pyrs</code> which are used to compute survival time function estimates. The former method requires argument <code>n.cens</code> and the latter argument <code>pyrs</code> to be supplied.
<code>relsurv.method</code>	either 'e2' or 'pp'; defines whether to compute relative survival using the EdererII method or using Pohar-Perme weighting; ignored if <code>surv.type != "surv.rel"</code>
<code>subset</code>	a logical condition; e.g. <code>subset = sex == 1</code> ; subsets the data before computations
<code>conf.level</code>	confidence level used in confidence intervals; e.g. 0.95 for 95 percent confidence intervals
<code>conf.type</code>	character string; must be one of "plain", "log-log" and "log"; defines the transformation used on the survival time function to yield confidence intervals via the delta method
<code>verbose</code>	logical; if TRUE, the function is chatty and returns some messages and timings along the process

Value

Returns a table of life time function values and other information with survival intervals as rows. Returns some of the following estimates of survival time functions:

- `surv.obs` - observed (raw, overall) survival
- `surv.obs.K` - observed cause-specific survival for cause K
- `CIF.k` - cumulative incidence function for cause k
- `CIF.rel` - cumulative incidence function using excess cases
- `r.e2` - relative survival, EdererII
- `r.pp` - relative survival, Pohar-Perme weighted

The suffix `.as` implies adjusted estimates, and `.lo` and `.hi` imply lower and upper confidence limits, respectively. The prefix `SE.` stands for standard error.

Basics

This function computes interval-based estimates of survival time functions, where the intervals are set by the user. For product-limit-based estimation see packages **survival** and **relsurv**.

if `surv.type = 'surv.obs'`, only 'raw' observed survival is estimated over the chosen time intervals. With `surv.type = 'surv.rel'`, also relative survival estimates are supplied in addition to observed survival figures.

`surv.type = 'cif.obs'` requests cumulative incidence functions (CIF) to be estimated. CIFs are estimated for each competing risk based on a survival-interval-specific proportional hazards assumption as described by Chiang (1968). With `surv.type = 'cif.rel'`, a CIF is estimated with using excess cases as the "cause-specific" cases. Finally, with `surv.type = 'surv.cause'`, cause-specific survivals are estimated separately for each separate type of event.

In hazard-based estimation (`surv.method = "hazard"`) survival time functions are transformations of the estimated corresponding hazard in the intervals. The hazard itself is estimated using counts of events (or excess events) and total subject-time in the interval. Life table `surv.method = "lifetable"` estimates are constructed as transformations of probabilities computed using counts of events and counts of subjects at risk.

The vignette [survtab_examples](#) has some practical examples.

Relative survival

When `surv.type = 'surv.rel'`, the user can choose `relsurv.method = 'pp'`, whereupon Pohar-Perme weighting is used. By default `relsurv.method = 'e2'`, i.e. the Ederer II method is used to estimate relative survival.

Adjusted estimates

Adjusted estimates in this context mean computing estimates separately by the levels of adjusting variables and returning weighted averages of the estimates. For example, computing estimates separately by age groups and returning a weighted average estimate (age-adjusted estimate).

Adjusting requires specification of both the adjusting variables and the weights for all the levels of the adjusting variables. The former can be accomplished by using `adjust()` with the argument `formula`, or by supplying variables directly to argument `adjust`. E.g. the following are all equivalent:

```
formula = fot ~ sex + adjust(agegr) + adjust(area)
formula = fot ~ sex + adjust(agegr, area)
formula = fot ~ sex, adjust = c("agegr", "area")
formula = fot ~ sex, adjust = list(agegr, area)
```

The adjusting variables must match with the variable names in the argument `weights`; see the [dedicated help page](#). Typically weights are supplied as a `list` or a `data.frame`. The former can be done by e.g.

```
weights = list(agegr = VEC1, area = VEC2),
```

where `VEC1` and `VEC2` are vectors of weights (which do not have to add up to one). See [survtab_examples](#) for an example of using a `data.frame` to pass weights.

Period analysis and other data selection schemes

To calculate e.g. period analysis (delayed entry) estimates, limit the data when/before supplying to this function. See [survtab_examples](#).

References

- Perme, Maja Pohar, Janez Stare, and Jacques Esteve. "On estimation in relative survival." *Biometrics* 68.1 (2012): 113-120. doi:10.1111/j.15410420.2011.01640.x
- Hakulinen, Timo, Karri Seppa, and Paul C. Lambert. "Choosing the relative survival method for cancer survival estimation." *European Journal of Cancer* 47.14 (2011): 2202-2210. doi:10.1016/j.ejca.2011.03.011
- Seppa, Karri, Timo Hakulinen, and Arun Pokhrel. "Choosing the net survival method for cancer survival estimation." *European Journal of Cancer* (2013). doi:10.1016/j.ejca.2013.09.019

CHIANG, Chin Long. Introduction to stochastic processes in biostatistics. 1968. ISBN-14: 978-0471155003

Seppa K., Dyba T. and Hakulinen T.: Cancer Survival, Reference Module in Biomedical Sciences. Elsevier. 08-Jan-2015. doi:[10.1016/B9780128012383.027458](https://doi.org/10.1016/B9780128012383.027458)

See Also

[splitMulti], [lexpand], [ICSS], [sire] [The survtab_examples vignette](#)

Other main functions: [Surv\(\)](#), [rate\(\)](#), [relpois\(\)](#), [relpois_ag\(\)](#), [sir\(\)](#), [sirspline\(\)](#), [survmean\(\)](#), [survtab_ag\(\)](#)

Other survtab functions: [Surv\(\)](#), [lines.survtab\(\)](#), [plot.survtab\(\)](#), [print.survtab\(\)](#), [summary.survtab\(\)](#), [survtab_ag\(\)](#)

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)

data("sire", package = "popEpi")
library(Epi)

## NOTE: recommended to use factor status variable
x <- Lexis(entry = list(FUT = 0, AGE = dg_age, CAL = get.yrs(dg_date)),
          exit = list(CAL = get.yrs(ex_date)),
          data = sire[sire$dg_date < sire$ex_date, ],
          exit.status = factor(status, levels = 0:2,
                              labels = c("alive", "canD", "othD")),
          merge = TRUE)

## phony group variable
set.seed(1L)
x$group <- rbinom(nrow(x), 1, 0.5)

## observed survival. explicit supplying of status:
st <- survtab(Surv(time = FUT, event = lex.Xst) ~ group, data = x,
              surv.type = "surv.obs",
              breaks = list(FUT = seq(0, 5, 1/12)))
## this assumes the status is lex.Xst (right 99.9 % of the time)
st <- survtab(FUT ~ group, data = x,
              surv.type = "surv.obs",
              breaks = list(FUT = seq(0, 5, 1/12)))

## relative survival (ederer II)
data("popmort", package = "popEpi")
pm <- data.frame(popmort)
names(pm) <- c("sex", "CAL", "AGE", "haz")
st <- survtab(FUT ~ group, data = x,
              surv.type = "surv.rel",
```

```

      pophaz = pm,
      breaks = list(FUT = seq(0, 5, 1/12)))

## ICSS weights usage
data("ICSS", package = "popEpi")
cut <- c(0, 30, 50, 70, Inf)
agegr <- cut(ICSS$age, cut, right = FALSE)
w <- aggregate(ICSS1~agegr, data = ICSS, FUN = sum)
x$agegr <- cut(x$dg_age, cut, right = FALSE)
st <- survtab(FUT ~ group + adjust(agegr), data = x,
              surv.type = "surv.rel",
              pophaz = pm, weights = w$ICSS1,
              breaks = list(FUT = seq(0, 5, 1/12)))

#### using dates with survtab
x <- Lexis(entry = list(FUT = 0L, AGE = dg_date-bi_date, CAL = dg_date),
          exit = list(CAL = ex_date),
          data = sire[sire$dg_date < sire$ex_date, ],
          exit.status = factor(status, levels = 0:2,
                              labels = c("alive", "canD", "othD")),
          merge = TRUE)
## phony group variable
set.seed(1L)
x$group <- rbinom(nrow(x), 1, 0.5)

st <- survtab(Surv(time = FUT, event = lex.Xst) ~ group, data = x,
              surv.type = "surv.obs",
              breaks = list(FUT = seq(0, 5, 1/12)*365.25))

## NOTE: population hazard should be reported at the same scale
## as time variables in your Lexis data.
data(popmort, package = "popEpi")
pm <- data.frame(popmort)
names(pm) <- c("sex", "CAL", "AGE", "haz")
## from year to day level
pm$haz <- pm$haz/365.25
pm$CAL <- as.Date(paste0(pm$CAL, "-01-01"))
pm$AGE <- pm$AGE*365.25

st <- survtab(Surv(time = FUT, event = lex.Xst) ~ group, data = x,
              surv.type = "surv.rel", relsurv.method = "e2",
              pophaz = pm,
              breaks = list(FUT = seq(0, 5, 1/12)*365.25))

```

Description

This function estimates survival time functions: survival, relative/net survival, and crude/absolute risk functions (CIF).

Usage

```
survtab_ag(
  formula = NULL,
  data,
  adjust = NULL,
  weights = NULL,
  surv.breaks = NULL,
  n = "at.risk",
  d = "from0to1",
  n.cens = "from0to0",
  pyrs = "pyrs",
  d.exp = "d.exp",
  n.pp = NULL,
  d.pp = "d.pp",
  d.pp.2 = "d.pp.2",
  n.cens.pp = "n.cens.pp",
  pyrs.pp = "pyrs.pp",
  d.exp.pp = "d.exp.pp",
  surv.type = "surv.rel",
  surv.method = "hazard",
  relsurv.method = "e2",
  subset = NULL,
  conf.level = 0.95,
  conf.type = "log-log",
  verbose = FALSE
)
```

Arguments

formula	a formula; the response must be the time scale to compute survival time function estimates over, e.g. <code>tot ~ sex</code> . Variables on the right-hand side of the formula separated by <code>+</code> are considered stratifying variables, for which estimates are computed separately. May contain usage of <code>adjust()</code> — see Details and Examples .
data	since popEpi 0.4.0, a <code>data.frame</code> containing variables used in formula and other arguments. <code>aggre</code> objects are recommended as they contain information on any time scales and are therefore safer; for creating <code>aggre</code> objects see <code>[as.aggre]</code> when your data is already aggregated and <code>aggre</code> for aggregating split Lexis objects.
adjust	can be used as an alternative to passing variables to argument formula within a call to <code>adjust()</code> ; e.g. <code>adjust = "agegr"</code> . Flexible input .
weights	typically a list of weights or a character string specifying an age group standardization scheme; see the dedicated help page and examples. NOTE: weights

	= "internal" is based on the counts of persons in follow-up at the start of follow-up (typically $T = 0$)
surv.breaks	a vector of breaks on the survival time scale. Optional if data is an <code>aggre</code> object and mandatory otherwise. Must define each intended interval; e.g. <code>surv.breaks = 0:5</code> when data has intervals defined by breaks <code>seq(0, 5, 1/12)</code> will aggregate to wider intervals first. It is generally recommended (and sufficient; see Seppa, Dyban and Hakulinen (2015)) to use monthly intervals where applicable.
n	variable containing counts of subjects at-risk at the start of a time interval; e.g. <code>n = "at.risk"</code> . Required when <code>surv.method = "lifetable"</code> . Flexible input .
d	variable(s) containing counts of subjects experiencing an event. With only one type of event, e.g. <code>d = "deaths"</code> . With multiple types of events (for CIF or cause-specific survival estimation), supply e.g. <code>d = c("canD", "othD")</code> . If the survival time function to be estimated does not use multiple types of events, supplying more than one variable to <code>d</code> simply causes the variables to be added together. Always required. Flexible input .
n.cens	variable containing counts of subjects censored during a survival time interval; E.g. <code>n.cens = "alive"</code> . Required when <code>surv.method = "lifetable"</code> . Flexible input .
pyrs	variable containing total subject-time accumulated within a survival time interval; E.g. <code>pyrs = "pyrs"</code> . Required when <code>surv.method = "hazard"</code> . Flexible input.
d.exp	variable denoting total "expected numbers of events" (typically computed <code>pyrs * pop.haz</code> , where <code>pop.haz</code> is the expected hazard level) accumulated within a survival time interval; E.g. <code>pyrs = "pyrs"</code> . Required when computing EdererII relative survivals or CIFs based on excess counts of events. Flexible input.
n.pp	variable containing total Pohar-Perme weighted counts of subjects at risk in an interval, supplied as argument <code>n</code> is supplied. Computed originally on the subject level as analogous to <code>pp * as.integer(status == "at-risk")</code> . Required when <code>relsurv.method = "pp"</code> . Flexible input.
d.pp	variable(s) containing Pohar-Perme weighted counts of events, supplied as argument <code>d</code> is supplied. Computed originally on the subject level as analogous to <code>pp * as.integer(status == some_event)</code> . Required when <code>relsurv.method = "pp"</code> . Flexible input.
d.pp.2	variable(s) containing total Pohar-Perme "double-weighted" counts of events, supplied as argument <code>d</code> is supplied. Computed originally on the subject level as analogous to <code>pp * pp * as.integer(status == some_event)</code> . Required when <code>relsurv.method = "pp"</code> . Flexible input.
n.cens.pp	variable containing total Pohar-Perme weighted counts censorings, supplied as argument <code>n.cens</code> is supplied. Computed originally on the subject level as analogous to <code>pp * as.integer(status == "censored")</code> . Required when <code>relsurv.method = "pp"</code> . Flexible input.
pyrs.pp	variable containing total Pohar-Perme weighted subject-times, supplied as argument <code>pyrs</code> is supplied. Computed originally on the subject level as analogous to <code>pp * pyrs</code> . Required when <code>relsurv.method = "pp"</code> . Flexible input.

d.exp.pp	variable containing total Pohar-Perme weighted counts of excess events, supplied as argument pyrs is supplied. Computed originally on the subject level as analogous to $pp * d.exp$. Required when <code>relsurv.method = "pp"</code> . Flexible input.
surv.type	one of 'surv.obs', 'surv.cause', 'surv.rel', 'cif.obs' or 'cif.rel'; defines what kind of survival time function(s) is/are estimated; see Details
surv.method	either 'lifetable' or 'hazard'; determines the method of calculating survival time functions, where the former computes ratios such as $p = d / (n - n.cens)$ and the latter utilizes subject-times (typically person-years) for hazard estimates such as $d/pyrs$ which are used to compute survival time function estimates. The former method requires argument <code>n.cens</code> and the latter argument <code>pyrs</code> to be supplied.
relsurv.method	either 'e2' or 'pp'; defines whether to compute relative survival using the EdererII method or using Pohar-Perme weighting; ignored if <code>surv.type != "surv.rel"</code>
subset	a logical condition; e.g. <code>subset = sex == 1</code> ; subsets the data before computations
conf.level	confidence level used in confidence intervals; e.g. 0.95 for 95 percent confidence intervals
conf.type	character string; must be one of "plain", "log-log" and "log"; defines the transformation used on the survival time function to yield confidence intervals via the delta method
verbose	logical; if TRUE, the function is chatty and returns some messages and timings along the process

Value

Returns a table of life time function values and other information with survival intervals as rows. Returns some of the following estimates of survival time functions:

- `surv.obs` - observed (raw, overall) survival
- `surv.obs.K` - observed cause-specific survival for cause K
- `CIF.k` - cumulative incidence function for cause k
- `CIF.rel` - cumulative incidence function using excess cases
- `r.e2` - relative survival, EdererII
- `r.pp` - relative survival, Pohar-Perme weighted

The suffix `.as` implies adjusted estimates, and `.lo` and `.hi` imply lower and upper confidence limits, respectively. The prefix `SE.` stands for standard error.

Basics

This function computes interval-based estimates of survival time functions, where the intervals are set by the user. For product-limit-based estimation see packages **survival** and **relsurv**.

if `surv.type = 'surv.obs'`, only 'raw' observed survival is estimated over the chosen time intervals. With `surv.type = 'surv.rel'`, also relative survival estimates are supplied in addition to observed survival figures.

`surv.type = 'cif.obs'` requests cumulative incidence functions (CIF) to be estimated. CIFs are estimated for each competing risk based on a survival-interval-specific proportional hazards assumption as described by Chiang (1968). With `surv.type = 'cif.rel'`, a CIF is estimated with using excess cases as the "cause-specific" cases. Finally, with `surv.type = 'surv.cause'`, cause-specific survivals are estimated separately for each separate type of event.

In hazard-based estimation (`surv.method = "hazard"`) survival time functions are transformations of the estimated corresponding hazard in the intervals. The hazard itself is estimated using counts of events (or excess events) and total subject-time in the interval. Life table `surv.method = "lifetable"` estimates are constructed as transformations of probabilities computed using counts of events and counts of subjects at risk.

The vignette [survtab_examples](#) has some practical examples.

Relative survival

When `surv.type = 'surv.rel'`, the user can choose `relsurv.method = 'pp'`, whereupon Pohar-Perme weighting is used. By default `relsurv.method = 'e2'`, i.e. the Ederer II method is used to estimate relative survival.

Adjusted estimates

Adjusted estimates in this context mean computing estimates separately by the levels of adjusting variables and returning weighted averages of the estimates. For example, computing estimates separately by age groups and returning a weighted average estimate (age-adjusted estimate).

Adjusting requires specification of both the adjusting variables and the weights for all the levels of the adjusting variables. The former can be accomplished by using `adjust()` with the argument `formula`, or by supplying variables directly to argument `adjust`. E.g. the following are all equivalent:

```
formula = fot ~ sex + adjust(agegr) + adjust(area)
formula = fot ~ sex + adjust(agegr, area)
formula = fot ~ sex, adjust = c("agegr", "area")
formula = fot ~ sex, adjust = list(agegr, area)
```

The adjusting variables must match with the variable names in the argument `weights`; see the [dedicated help page](#). Typically weights are supplied as a `list` or a `data.frame`. The former can be done by e.g.

```
weights = list(agegr = VEC1, area = VEC2),
```

where `VEC1` and `VEC2` are vectors of weights (which do not have to add up to one). See [survtab_examples](#) for an example of using a `data.frame` to pass weights.

Period analysis and other data selection schemes

To calculate e.g. period analysis (delayed entry) estimates, limit the data when/before supplying to this function. See [survtab_examples](#).

Data requirements

survtab_ag computes estimates of survival time functions using pre-aggregated data. For using subject-level data directly, use [survtab]. For aggregating data, see [lexpand] and [aggre].

By default, and if data is an aggre object (not mandatory), survtab_ag makes use of the exact same breaks that were used in splitting the original data (with e.g. lexpand), so it is not necessary to specify any surv.breaks. If specified, the surv.breaks must be a subset of the pertinent pre-existing breaks. When data is not an aggre object, breaks must always be specified. Interval lengths (delta in output) are also calculated based on whichever breaks are used, so the upper limit of the breaks should therefore be meaningful and never e.g. Inf.

References

Perme, Maja Pohar, Janez Stare, and Jacques Esteve. "On estimation in relative survival." *Biometrics* 68.1 (2012): 113-120. doi:10.1111/j.15410420.2011.01640.x

Hakulinen, Timo, Karri Seppa, and Paul C. Lambert. "Choosing the relative survival method for cancer survival estimation." *European Journal of Cancer* 47.14 (2011): 2202-2210. doi:10.1016/j.ejca.2011.03.011

Seppa, Karri, Timo Hakulinen, and Arun Pokhrel. "Choosing the net survival method for cancer survival estimation." *European Journal of Cancer* (2013). doi:10.1016/j.ejca.2013.09.019

CHIANG, Chin Long. *Introduction to stochastic processes in biostatistics*. 1968. ISBN-14: 978-0471155003

Seppa K., Dyba T. and Hakulinen T.: *Cancer Survival, Reference Module in Biomedical Sciences*. Elsevier. 08-Jan-2015. doi:10.1016/B9780128012383.027458

See Also

[splitMulti], [lexpand], [ICSS], [sire] [The survtab_examples vignette](#)

Other main functions: [Surv\(\)](#), [rate\(\)](#), [relpois\(\)](#), [relpois_ag\(\)](#), [sir\(\)](#), [sirspline\(\)](#), [survmean\(\)](#), [survtab\(\)](#)

Other survtab functions: [Surv\(\)](#), [lines.survtab\(\)](#), [plot.survtab\(\)](#), [print.survtab\(\)](#), [summary.survtab\(\)](#), [survtab\(\)](#)

Examples

```
# this data.table::setDTthreads call is included here only to
# conform to the CRAN submission requirement to only use at most 2
# threads. you do not need to set this to use popEpi.
# however some long calculations may benefit from using more threads.
data.table::setDTthreads(2L)
## see more examples with explanations in vignette("survtab_examples")

#### survtab_ag usage

data("sire", package = "popEpi")
## prepare data for e.g. 5-year "period analysis" for 2008-2012
## note: sire is a simulated cohort integrated into popEpi.
BL <- list(fot=seq(0, 5, by = 1/12),
           per = c("2008-01-01", "2013-01-01"))
```

```

x <- leexpand(sire, birth = bi_date, entry = dg_date, exit = ex_date,
              status = status %in% 1:2,
              breaks = BL,
              pophaz = popmort,
              aggre = list(fot))

## calculate relative EdererII period method
## NOTE: x is an aggre object here, so surv.breaks are deduced
## automatically
st <- survtab_ag(fot ~ 1, data = x)

summary(st, t = 1:5) ## annual estimates
summary(st, q = list(r.e2 = 0.75)) ## 1st interval where r.e2 < 0.75 at end

plot(st)

## non-aggre data: first call to survtab_ag would fail
df <- data.frame(x)
# st <- survtab_ag(fot ~ 1, data = x)
st <- survtab_ag(fot ~ 1, data = x, surv.breaks = BL$fot)

## calculate age-standardised 5-year relative survival ratio using
## Ederer II method and period approach

sire$agegr <- cut(sire$dg_age, c(0, 45, 55, 65, 75, Inf), right = FALSE)
BL <- list(fot = seq(0, 5, by = 1/12),
           per = c("2008-01-01", "2013-01-01"))
x <- leexpand(sire, birth = bi_date, entry = dg_date, exit = ex_date,
              status = status %in% 1:2,
              breaks = BL,
              pophaz = popmort,
              aggre = list(agegr, fot))

## age standardisation using internal weights (age distribution of
## patients diagnosed within the period window)
## (NOTE: what is done here is equivalent to using weights = "internal")
w <- aggregate(at.risk ~ agegr, data = x[x$fot == 0], FUN = sum)
names(w) <- c("agegr", "weights")

st <- survtab_ag(fot ~ adjust(agegr), data = x, weights = w)
plot(st, y = "r.e2.as", col = c("blue"))

## age standardisation using ICSS1 weights
data(ICSS)
cut <- c(0, 45, 55, 65, 75, Inf)
agegr <- cut(ICSS$age, cut, right = FALSE)
w <- aggregate(ICSS1~agegr, data = ICSS, FUN = sum)
names(w) <- c("agegr", "weights")

st <- survtab_ag(fot ~ adjust(agegr), data = x, weights = w)
lines(st, y = "r.e2.as", col = c("red"))

```

```
## cause-specific survival
sire$stat <- factor(sire$status, 0:2, c("alive", "canD", "othD"))
x <- lextand(sire, birth = bi_date, entry = dg_date, exit = ex_date,
            status = stat,
            breaks = BL,
            pophaz = popmort,
            aggre = list(agegr, fot))
st <- survtab_ag(fot ~ adjust(agegr), data = x, weights = w,
                d = c("fromalivetocanD", "fromalivetoothD"),
                surv.type = "surv.cause")
plot(st, y = "surv.obs.fromalivetocanD.as")
lines(st, y = "surv.obs.fromalivetoothD.as", col = "red")
```

try2int

*Attempt coercion to integer***Description**

Attempts to convert a numeric object to integer, but won't if loss of information is imminent (if values after decimal are not zero for even one value in obj)

Usage

```
try2int(obj, tol = .Machine$double.eps^0.5)
```

Arguments

obj	a numeric vector
tol	tolerance; if each numeric value in obj deviate from the corresponding integers at most the value of tol, they are considered to be integers; e.g. by default $1 + .Machine$double.eps$ is considered to be an integer but $1 + .Machine$double.eps^{0.49}$ is not.

Value

An integer vector if no information is lost in coercion; else numeric vector.

Author(s)

James Arnold

Source

[Stackoverflow thread](#)

Index

- * **Lexis_functions**
 - lexis_funs, [25](#)
 - lexis_merge, [28](#)
 - lexis_split_merge_aggregate_by_stratum, [33](#)
 - * **aggregation functions**
 - aggre, [4](#)
 - as.aggre, [10](#)
 - lexpand, [45](#)
 - setaggre, [98](#)
 - summary.aggre, [118](#)
 - * **main functions**
 - rate, [82](#)
 - relpois, [90](#)
 - relpois_ag, [92](#)
 - sir, [102](#)
 - sirspline, [110](#)
 - Surv, [121](#)
 - survmean, [151](#)
 - survtab, [155](#)
 - survtab_ag, [160](#)
 - * **popEpi argument evaluation docs**
 - direct_standardization, [15](#)
 - flexible_argument, [18](#)
 - * **popEpi data**
 - ICSS, [22](#)
 - meanpop_fi, [58](#)
 - popmort, [67](#)
 - sibr, [101](#)
 - sire, [109](#)
 - stdpop101, [117](#)
 - stdpop18, [117](#)
 - * **rate functions**
 - rate, [82](#)
 - rate_ratio, [88](#)
 - * **relpois functions**
 - relpois, [90](#)
 - relpois_ag, [92](#)
 - rpcurve, [96](#)
 - RPL, [97](#)
 - * **sir functions**
 - lines.sirspline, [51](#)
 - plot.sirspline, [61](#)
 - sir, [102](#)
 - sir_exp, [105](#)
 - sir_ratio, [107](#)
 - sirspline, [110](#)
 - * **splitting functions**
 - lexpand, [45](#)
 - splitLexisDT, [112](#)
 - splitMulti, [114](#)
 - * **survival data**
 - sibr, [101](#)
 - sire, [109](#)
 - * **survmean functions**
 - lines.survmean, [52](#)
 - plot.survmean, [62](#)
 - Surv, [121](#)
 - survmean, [151](#)
 - * **survtab functions**
 - lines.survtab, [53](#)
 - plot.survtab, [63](#)
 - print.survtab, [82](#)
 - summary.survtab, [119](#)
 - Surv, [121](#)
 - survtab, [155](#)
 - survtab_ag, [160](#)
 - * **weights**
 - direct_standardization, [15](#)
 - ICSS, [22](#)
 - stdpop101, [117](#)
 - stdpop18, [117](#)
- adjust, [3](#)
aggre, [4](#)
aggre(), [11](#), [50](#), [98](#), [118](#)
all_names_present, [7](#)
array_df_ratetable_utils, [8](#)

- array_to_long_df
 - (array_df_ratetable_utils), 8
- array_to_long_dt
 - (array_df_ratetable_utils), 8
- array_to_ratetable
 - (array_df_ratetable_utils), 8
- as.aggre, 10
- as.aggre(), 6, 50, 98, 118
- as.Date.yrs, 12
- cast_simple, 13
- cut_bound, 14
- dedicated help page, 83, 151, 156, 158, 161, 164
- dimnames, 9
- direct_adjusting
 - (direct_standardization), 15
- direct_standardization, 15, 20, 23, 117, 118
- expr.by.cj(ltable), 55
- fac2num, 17
- Flexible input, 4, 25, 83, 151, 156, 161, 162
- flexible_argument, 17, 18
- get.yrs, 21
- ICSS, 17, 22, 58, 67, 101, 110, 117, 118
- is.Date, 23
- is_leap_year, 24
- lexis_breaks_collapse_1d
 - (surv_estimate), 122
- Lexis_dt(lexis_funs), 25
- Lexis_fpa(lexis_funs), 25
- lexis_funs, 25, 31, 38
- lexis_merge, 28
- lexis_merge(), 26, 38
- lexis_split_merge_aggregate_by_stratum, 33
- lexis_split_merge_aggregate_by_stratum(), 26, 31
- lexpand, 45
- lexpand(), 6, 11, 98, 113, 115, 118
- lines.sirspline, 51
- lines.sirspline(), 62, 104, 106, 108, 112
- lines.survmean, 52
- lines.survmean(), 63, 121, 153
- lines.survtab, 53
- lines.survtab(), 64, 82, 120, 121, 159, 165
- long_df_to_array
 - (array_df_ratetable_utils), 8
- long_df_to_ratetable
 - (array_df_ratetable_utils), 8
- long_dt_to_array
 - (array_df_ratetable_utils), 8
- long_dt_to_ratetable
 - (array_df_ratetable_utils), 8
- lower_bound, 54
- ltable, 55
- meanpop_fi, 23, 58, 67, 101, 110, 117, 118
- na2zero, 58
- plot.rate, 59
- plot.sir, 60
- plot.sirspline, 61
- plot.sirspline(), 51, 104, 106, 108, 112
- plot.survmean, 62
- plot.survmean(), 52, 121, 153
- plot.survtab, 63
- plot.survtab(), 53, 82, 120, 121, 159, 165
- poisson.ci, 65
- pophaz, 66
- popmort, 23, 58, 67, 101, 110, 117, 118
- prepExpo, 67
- prev_functions, 69
- prev_lexis(prev_functions), 69
- print.aggre, 80
- print.rate, 81
- print.survtab, 82
- print.survtab(), 53, 64, 120, 121, 159, 165
- quote, 124
- rate, 82
- rate(), 89, 92, 94, 104, 112, 121, 153, 159, 165
- rate_estimate, 84
- rate_ratio, 88
- rate_ratio(), 83
- ratetable_to_array
 - (array_df_ratetable_utils), 8
- ratetable_to_long_df
 - (array_df_ratetable_utils), 8
- ratetable_to_long_dt
 - (array_df_ratetable_utils), 8

- relpois, 90
- relpois(), 83, 94, 96, 97, 104, 112, 121, 153, 159, 165
- relpois_ag, 92
- relpois_ag(), 83, 92, 96, 97, 104, 112, 121, 153, 159, 165
- robust_values, 94
- rpcurve, 96
- rpcurve(), 92, 94, 97
- RPL, 92, 94, 96, 97
- setaggre, 98
- setaggre(), 6, 11, 50, 118
- setclass, 99
- setcolsnul, 100
- sibr, 23, 58, 67, 101, 110, 117, 118
- sir, 102
- sir(), 51, 62, 83, 92, 94, 106, 108, 112, 121, 153, 159, 165
- sir_ag(sir_exp), 105
- sir_exp, 105
- sir_exp(), 51, 62, 104, 108, 112
- sir_lex(sir_exp), 105
- sir_ratio, 107
- sir_ratio(), 51, 62, 104, 106, 112
- sire, 23, 58, 67, 101, 109, 117, 118
- sirspline, 110
- sirspline(), 51, 62, 83, 92, 94, 104, 106, 108, 121, 153, 159, 165
- splitLexisDT, 112
- splitLexisDT(), 50, 115
- splitMulti, 114
- splitMulti(), 50, 113
- stats::poisson.test, 89
- stdpop101, 17, 23, 58, 67, 101, 110, 117, 118
- stdpop18, 17, 23, 58, 67, 101, 110, 117, 117
- str2lang, 124
- substitute, 124
- summary.aggre, 118
- summary.aggre(), 6, 11, 50, 98
- summary.survtab, 119
- summary.survtab(), 53, 64, 82, 121, 159, 165
- Surv, 121
- Surv(), 52, 53, 63, 64, 82, 83, 92, 94, 104, 112, 120, 153, 159, 165
- surv_brenner_weight_dt(surv_estimate), 122
- surv_collapse_strata_1d(surv_estimate), 122
- surv_collapse_strata_list(surv_estimate), 122
- surv_collapse_ts_1d(surv_estimate), 122
- surv_estimate, 122
- surv_functions(surv_estimate), 122
- surv_individual_weights(surv_estimate), 122
- surv_lexis(surv_estimate), 122
- surv_observed_weight_dt(surv_estimate), 122
- survival::ratetable, 8–10
- survmean, 151
- survmean(), 52, 63, 83, 92, 94, 104, 112, 121, 159, 165
- survtab, 155
- survtab(), 53, 64, 82, 83, 92, 94, 104, 112, 120, 121, 153, 165
- survtab_ag, 160
- survtab_ag(), 53, 64, 82, 83, 92, 94, 104, 112, 120, 121, 153, 159
- try2int, 167