Package 'intervalaverage'

May 8, 2026

Title Time-Weighted Averaging for Interval Data

Version 0.8.0

Description Perform fast and memory efficient time-weighted averaging of values measured over intervals into new arbitrary intervals.
This package is useful in the context of data measured or represented as constant values over intervals on a one-dimensional discrete axis (e.g. time-integrated averages of a curve over defined periods).
This package was written specifically to deal with air pollution data recorded or predicted as averages over sampling periods.
Data in this format often needs to be shifted to non-aligned periods or averaged up to periods of longer duration (e.g. averaging data measured over sequential non-overlapping periods to calendar years).

License GPL-3

Encoding UTF-8

LazyData true

Depends data.table (>= 1.12.8)

RoxygenNote 7.1.0

Suggests knitr, rmarkdown, testthat

VignetteBuilder knitr

LinkingTo Rcpp

Imports Rcpp

NeedsCompilation yes

Author Michael Young [aut, cre]

Maintainer Michael Young <myoung3@uw.edu>

Repository CRAN

Date/Publication 2020-07-23 16:00:10 UTC

Contents

`CJ.dt` *grid expand an arbitrary number of data.tables*

Description

similar to data.table::CJ and base::expand.grid except for rows of data.tables.

Usage

```
CJ.dt(..., groups = NULL)
```

Arguments

`...`	data.tables
`groups`	a character vector corresponding to column names of grouping vars in all of the data.tables

Details

CJ.dt computes successive cartesian join over rows of each table paying no attention to whatever the tables are keyed on.

Examples

```
#' CJ.dt(data.table(c(1,2,2),c(1,1,1)),data.table(c("a","b"),c("c","d")))
#If you want to expand x to unique values of a non-unique columns in y
x <- data.table(c(1,2,3),c("a","b","b"))
y <- data.table(id=c(1,2,2,1,3),value=c(2,4,1,7,3))
z <- CJ.dt(x, y[,list(id=unique(id))])
#if you want to merge this back to y
y[z,on="id",allow.cartesian=TRUE] #or z[y,on="id",allow.cartesian=TRUE]
```

`intervalaverage` *time-weighted average of values measured over intervals*

Description

`intervalaverage` takes values recorded over non-overlapping intervals and averages them to defined intervals, possibly within groups (individuals/monitors/locations/etc). This function could be used to take averages over long intervals of values measured over short intervals and/or to take short "averages" of values measured over longer intervals (ie, downsample without smoothing). Measurement intervals and averaging intervals need not align. In the event that an averaging interval contains more than one measurement interval, a weighted average is calculated (ie each measurement is weighted on the duration of its interval's overlap with the averaging period interval).

Usage

```
intervalaverage(
  x,
  y,
  interval_vars,
  value_vars,
  group_vars = NULL,
  required_percentage = 100,
  skip_overlap_check = FALSE,
  verbose = FALSE
)
```

Arguments

`x`	a data.table containing values measured over intervals. see `interval_vars` parameter for how to specify interval columns and `value_vars` for how to specify value columns. intervals in `x` must must be completely non-overlapping within groups defined by group_vars. if `group_vars` is specified (non-`NULL`), `x` must also contain columns specified in `group_vars`.
`y`	a data.table object containing intervals over which averages of `x` values should be computed. averaging intervals in `y`, unlike measurement intervals in `x`, may be overlapping within groups. if `group_vars` is specified (non-`NULL`), `y` must contains those `group_vars` column names (and this would allow different averaging periods for each group)
`interval_vars`	a length-2 character vector of column names in both `x` and `y`. These column names specify columns in x and y that define closed (inclusive) starting and ending intervals. The column name specifying the lower-bound column must be specified first. these columns in x and y must all be of the same class and either be integer or IDate. The interval_vars character vector cannot be named. This is reserved for future use allowing different interval_vars column names in x and y.

value_vars — a character vector of column names in x. This specifies the columns to be averaged.

group_vars — A character vector of column names in both x and y. The interaction of these variables define groups in which averages of x values will be taken. specifying subjects/monitors/locations within which to take averages. By default this is NULL, in which case averages are taken over the entire x dataset for each y period. The group_vars character vector cannot be named. This is reserved for future use allowing different interval_vars column names in x and y.

required_percentage — This percentage of the duration of each (possibly group-specific) y interval must be observed and nonmissing for a specific value_var in x in order for the return table to contain a nonmissing average of the value_var for that y interval. If the percentage of the nonmissing value_var observations is less than required_percentage an NA will be be returned for that average. The default is 100, meaning that if *any* portion of a y interval is either not recorded or missing in x, then the corresponding return row will contain a an NA for the average of that value_var.

skip_overlap_check — by default, FALSE. setting this to TRUE will skip internal checks to make sure x intervals are non-overlapping within groups defined by group_vars. intervals in x must be non-overlapping, but you may want to skip this check if you've already checked this because it is computationally intensive for large datasets.

verbose — include printed timing information? by default, FALSE

Details

All intervals are treated as closed (ie inclusive of the start and end values in interval_vars)

x and y are not copied but rather passed by reference to function internals but the order of these data.tables is restored on function completion or error,

When required_percentage is less than 100, xminstart and xmaxend may be useful to determine whether an average meets specified coverage requirements in terms of not just percent of missingness but whether values are represented through the range of the y interval

Value

returns a data.table object. Rows of the return data.table correspond to intervals from y. i.e, the number of rows of the return will be the number of rows of y. Columns of the returned data.table are as follows:

- grouping variables as specified in group_vars
- interval columns corresponding to intervals in y. These columns are named the same they were in x and y and as specified in interval_vars
- value variable columns from x, averaged to periods in y. named the same as they were in x

- yduration: the length of the interval (ie as a count) specified in y
- xduration: the total length of the intervals (ie as a count) from x that fall into this interval from y. this will be equal to yduration if x is comprehensive for (ie, fully covers) this interval from y.
- nobs_<value_vars>: for each value_var specified, this is the count of non-missing values from x that fall into this interval from y. this will be equal to xduration if the value_var contains no NA values over the y interval. If there are NAs in value variables, then nobs_<value_vars> will be different from xduration and won't necessarily be all the same for each value_var.
- xminstart: For each returned interval (ie the intervals from Y) the minimum of the start intervals represented in x. If the start of the earliest x interval is less than the start of the y interval, the minimum of the y interval is returned. Note, this is the minimum start time in x matching with the y interval whether or not any value_vars were missing or not for that start time. If you need non-missing minimum start times, you could remove NA intervals from x prior to calling intervalaverage (this would need to be done separately for each value_var).
- xmaxend: similar to xminstart but the maximum of the end intervals represented in x. Again, this does not pay attention to whether the interval in x had non-missing value_vars.

Examples

```
x <- data.table(start=seq(1L,by=7L,length=6),
                end=seq(7L,by=7L,length=6),
                pm25=c(10,12,8,14,22,18))

y <- data.table(start=seq(3L,by=7L,length=6),
                end=seq(9L,by=7L,length=6))

z <- intervalaverage(x,y,interval_vars=c("start","end"),
                     value_vars=c("pm25"))

#also see vignette for more extensive examples
```

---

intervalintersect *Intersect intervals within groups*

Description

Given two tables each containing a set of intervals, find all interval intersections within groups. Returns a data.table containing all columns from both tables. One use of this function is to take a table containing an address history (a table containing the intervals when study participants lived at past addresses) and join it to an exposure history table (a complete set of exposure predictions for each address, where the exposures are stored as the average value over a set of intervals) returning the set of exposure intervals at addresses clipped to exactly when the participant lived at that address.

Usage

```
intervalintersect(
  x,
  y,
  interval_vars,
  group_vars = NULL,
  interval_vars_out = c("start", "end"),
  verbose = FALSE
)
```

Arguments

x	A data.table with two columns defining closed intervals (see also interval_vars parameter)
y	A data.table with two columns defining closed intervals (see also interval_vars parameter)
interval_vars	Either a length-2 character vector denoting column names in both x and y or a named length-2 character vector where the names are column names in x and the values are column names in y. These column names specify columns in x and y that define closed (inclusive) starting and ending intervals. The column name specifying the lower-bound column must be specified first. these columns in x and y must all be of the same class and either be integer or IDate.
group_vars	NULL, or either a character vector denoting the column name(s) in x and y, or a named character vector where the name is the column name in x and the value is the column name in y. This/these column(s) serve as an additional keying variable in the join (ie in addition to the interval join) such that intervals in x will only be joined to overlapping in intervals in y where the group_vars values are the same.
interval_vars_out	The column names of the interval columns in the return data.table. By default the return table will contain columns `c("start","end")`. If your input tables already contain these columns, you need to either specify `interval_vars_out` to be non-conflicting names with columns in x and y. Or you rename columns in x and y to not contain columns named `c("start","end")`.
verbose	Prints additional information about the function processing.

Details

All intervals are treated as closed (ie inclusive of the start and end values in the columns specified by interval_vars)

x and y are not copied but rather passed by reference to function internals but the order of these data.tables is restored on function completion or error,

Technically speaking this is just an inner cartesian join where the last two join variables are doing a non-equi join for partial overlaps. Then each interval intersect is calculated using max and min.

If there are columns with the same names in both x and y (including interval_vars but excepting group_vars), the return value will still return both columns. The column in y will be names as it

was originally and the column in x will be prepended with the letter i followed with a dot: `i.`

Note that the function returns the same result if you switch x and y (with the exception of switched column names in the case of column name conflicts as just discussed)

Value

A data.table with columns `interval_vars_out` which denote the start and stop period for each interval. This return table also contains columns in x and y. See details for how naming conflicts are dealt with.

See Also

`is.overlapping` To test if a table contains overlapping intervals within values of `group_vars`

Examples

```
set.seed(42)
y <- data.table(addr_id=c(1,2,2,3,5),
ppt_id=c(1,1,1,2,2),
addr_start=c(1L,10L,12L,1L,1L),
addr_end=c(9L,11L,14L,17L,10L))
x <- data.table(addr_id=rep(1:4,each=3),
exposure_start=rep(c(1L,8L,15L),times=4),
exposure_end=rep(c(7L,14L,21L),times=4),
exposure_value=c(rnorm(12))
)
intervalintersect(x,y,
interval_vars=c(exposure_start="addr_start",exposure_end="addr_end"),
"addr_id")
y2 <- data.table(addr_id=c(1,2,2,2,3),
ppt_id=c(1,1,1,1,2),
addr_start=c(1L,2L,3L,4L,1L),
addr_end=c(9L,12L,13L,8L,10L))

#intervalintersect will still work when there are overlapping intervals within a table:
is.overlapping(y2,interval_vars =c("addr_start","addr_end") ,group_vars="addr_id")

intervalintersect(x,y2,
                  interval_vars=c(exposure_start="addr_start",exposure_end="addr_end"),
"addr_id")


x2 <- data.table(addr_id=rep(1:4,each=3),
exposure_start=rep(c(1L,7L,14L),times=4),
                exposure_end=rep(c(7L,14L,21L),times=4),
exposure_value=c(rnorm(12))
)
is.overlapping(x2,interval_vars =c("exposure_start","exposure_end") ,group_vars="addr_id")

intervalintersect(x2,y2,
interval_vars=c(exposure_start="addr_start",exposure_end="addr_end"),
```

```
"addr_id")
#however, it may be meaningful isolate intervals of partial-overlap within
#each table and deal with them
#prior to intersecting the tables together:

x2z <- isolateoverlaps(x2,interval_vars=c("exposure_start","exposure_end"),group_vars=c("addr_id"),
interval_vars_out=c("exposure_start2","exposure_end2"))
x2b <- x2z[, list(exposure_value=mean(exposure_value)),
 by=c("addr_id","exposure_start2","exposure_end2")]
data.table::setnames(x2b, c("exposure_start2","exposure_end2"),c("exposure_start","exposure_end"))

y2z <- isolateoverlaps(y2,interval_vars=c("addr_start","addr_end"),group_vars=c("addr_id"),
interval_vars_out = c("addr_start2","addr_end2"))
y2b <- unique(y2z[, list(addr_id, ppt_id,addr_start2,addr_end2)])
data.table::setnames(y2b, c("addr_start2","addr_end2"), c("addr_start","addr_end"))

intervalintersect(x2b,y2b,
interval_vars=c(exposure_start="addr_start",exposure_end="addr_end"),
"addr_id")
```

`is.overlapping` *Test for self-overlap*

Description

Test whether a data.table contains intervals which partially or completely overlap with other intervals in different rows, possibly within groups

Usage

```
is.overlapping(x, interval_vars, group_vars = NULL, verbose = FALSE)
```

Arguments

`x`	A data.table with two columns defining closed intervals (see also interval_vars).
`interval_vars`	A length-2 character vector corresponding to column names of x which designate the closed (inclusive) starting and ending intervals. The column name specifying the lower-bound column must be specified first.
`group_vars`	NULL or a character vector corresponding to column names of x. overlap checks will occur within groups defined by the columns specified here.
`verbose`	prints additional information, default is FALSE

Value

length-1 logical vector. TRUE if there are overlaps, FALSE otherwise.

Examples

```
x <- data.table(start=c(1,2),end=c(3,4))
is.overlapping(x,c("start","end")) #the interval 1,3 overlaps with the interval 2,4
y <- data.table(start=c(1,3),end=c(2,4))
is.overlapping(y,c("start","end")) #the interval 1,2 doesn't overlap other intervals in y
z <- data.table(start=c(1,3,1,2),end=c(2,4,3,4),id=c(1,1,2,2))
is.overlapping(z,c("start","end"),"id")
```

`isolateoverlaps` *isolate sections of overlapping intervals*

Description

Given a set of intervals in a table, isolate sections of intervals that are overlapping with other in intervals (optionally, within groups). Returns a data.table that contains intervals which are mutually non-overlapping or exactly overlapping with other intervals (ie there are no partially overlapping intervals) (optionally within groups). Note that this doesn't just return the intersects; the original interval data is conserved such that for each interval/row in x, the return table has one or more non-overlapping intervals that together form the union of that original interval.

Usage

```
isolateoverlaps(
  x,
  interval_vars,
  group_vars = NULL,
  interval_vars_out = c("start", "end")
)
```

Arguments

`x`	A data.table containing a set of intervals.
`interval_vars`	A length-2 character vector denoting column names in x. these columns must be of the same class and be integer or IDate. The column name specifying the lower-bound column must be specified first.
`group_vars`	NULL, or a character vector denoting column names in x. These columns serve as grouping variables such that testing for overlaps and subsequent isolation only occur within categories defined by the combination of the group variables.
`interval_vars_out`	The desired column names of the interval columns in the return data.table. By default these columns will be generated to be named `c("start","end")`. If x contains columns with the same name as the desired output column names specified in `interval_vars_out`, the function will return in error to avoid naming confusion in the return table.

Details

All intervals are treated as closed (ie inclusive of the start and end values in the columns specified by interval_vars)

x is not copied but rather passed by reference to function internals but the order of this data.tables is restored on function completion or error.

Value

A data.table with columns `interval_vars_out` which denote the start and stop period for each new interval. This return table also contains columns in x (including the original interval columns).

See Also

`is.overlapping` To test if a table contains overlapping intervals within values of `group_vars`

Examples

```
set.seed(23)
x2 <- data.table(addr_id=rep(1:4,each=3),
                 exposure_start=rep(c(1L,7L,14L),times=4),
exposure_end=rep(c(7L,14L,21L),times=4),
exposure_value=c(rnorm(12))
)
x2z <- isolateoverlaps(x2,interval_vars=c("exposure_start","exposure_end"),group_vars=c("addr_id"))
x2z
#x2b represents x2 when where exposure values in overlapping intervals have been averaged
x2b <- x2z[, list(exposure_value=mean(exposure_value)),by=c("addr_id","start","end")]
```

Index