Package 'rem'

June 23, 2017

Type Package

Title Relational Event Models (REM)

Version 1.2.8

Date 2017-06-23

Author Laurence Brandenberger

Maintainer Laurence Brandenberger <laurence.brandenberger@eawag.ch>

Description Calculate endogenous network effects in event sequences and fit relational event models (REM): Using network event sequences (where each tie between a sender and a target in a network is time-stamped), REMs can measure how networks form and evolve over time. Endogenous patterns such as popularity effects, inertia, similarities, cycles or triads can be calculated and analyzed over time.

License GPL (>= 2)

Depends R (>= 2.14.0)

Imports Rcpp, foreach, doParallel

LinkingTo Rcpp

Suggests texreg, statnet, ggplot2

NeedsCompilation yes

Repository CRAN

Date/Publication 2017-06-23 10:37:59 UTC

R topics documented:

`rem-package` *Fit Relational Event Models (REM)*

Description

The **rem** package uses a combination of event history and network analysis to test network dependencies in event sequences. If events in an event sequence depend on each other, network structures and patterns can be calculated and estimated using relational event models. The `rem`-package includes functions to calculate endogenous network statistics in (signed) one-, two- and multi-mode network event sequences. The statistics include inertia (inertiaStat), reciprocity (reciprocityStat), in- or outdegree statistics (degreeStat), closing triads (triadStat), closing four-cycles (fourCycleStat) or endogenous similarity statistics (similarityStat). The rate of event occurrence can then be tested using standard models of event history analysis, such as a stratified Cox model (or a conditional logistic regression). createRemDataset can be used to create counting process data sets with dynamic risk sets.

Details

Package:	rem
Type:	Package
Version:	1.2.8
Date:	2017-06-23

Author(s)

Laurence Brandenberger <`laurence.brandenberger@eawag.ch`>

References

Lerner, J., Bussmann, M., Snijders, T. A., & Brandes, U. (2013). Modeling frequency and type of interaction in event networks. Corvinus Journal of Sociology and Social Policy, (1), 3-32.

`createRemDataset` *Create REM data set with dynamic risk sets*

Description

The function creates counting process data sets with dynamic risk sets for relational event models. For each event in the event sequence, null-events are generated and represent possible events that could have happened at that time but did not. A data set with true and null-events is returned with an event dummy for whether the event occurred or was simply possible (variable `eventdummy`). The returned data set also includes a variable `eventTime` which represents the true time of the reported event.

Usage

```
createRemDataset(data, sender, target, eventSequence,
eventAttribute = NULL, time = NULL,
start = NULL, startDate = NULL,
end = NULL, endDate = NULL,
timeformat = NULL,
atEventTimesOnly = TRUE, untilEventOccurrs = TRUE,
includeAllPossibleEvents = FALSE, possibleEvents = NULL,
returnInputData = FALSE)
```

Arguments

`data`	A data frame containing all the events.
`sender`	A string (or factor or numeric) variable that represents the sender of the event.
`target`	A string (or factor or numeric) variable that represents the target of the event.
`eventSequence`	Numeric variable that represents the event sequence. The variable has to be sorted in ascending order.
`eventAttribute`	An optional variable that represents an attribute to an event. Repeated events affect the construction of the counting process data set. Use the `eventAttribute`-variable to specify the uniqueness of an event. If `eventAttribute = NULL`, events are defines as sender-target nodes only.
`time`	An optional date variable that represents the date an event took place. The variable is used if `startDate` or `endDate` are specified. `timeformat` should be used to specify which format the date variable is in, in case it was not yet converted to a Date-variable.
`start`	An optional numeric variable that indicates at which point in the event sequence a specific event was at risk. The variable has to be numerical and correspond to the variable `eventSequence`. If this option is used, each event in the event data set will be considered at risk from the specified value onwards. If it is not specified, `start` is defined as the first value in the event sequence. In case of repeated events, the start-value for each duplicated event is one event-unit after the last such event.
`startDate`	An optional date variable that represents the date an event started being at risk. `timeformat` should be used to specify which format the date variable is in, incase it was not yet converted to a Date-variable.
`end`	An optional numeric variable that indicates at which point in the event sequence a specific event stopped being at risk. The variable has to be numerical and correspond to the variable `eventSequence`. If this option is used, each event in the event data set will be considered at risk until the specified value.
`endDate`	An optional date variable that represents the date an event stoped being at risk. `timeformat` should be used to specify which format the date variable is in, incase it was not yet converted to a Date-variable.

`timeformat` A character string indicating the format of the `datevar`. see `as.Date`

`atEventTimesOnly`
: `TRUE/FALSE`. Boolean option for continuous event sequences. If `atEventTimesOnly = TRUE`, null-events are only created at times, when an event occurred. If `atEventTimesOnly = FALSE`, null-events are created on each event-unit from `min(eventSequence):max(eventSequence)`. For instance: Given an event sequence with three events at `c(1, 4, 6)`: If `atEventTimesOnly = TRUE` null events are created for events 1, 4 and 6. If `atEventTimesOnly = FALSE` null-events are also created for days 2, 3 and 5.

`untilEventOccurrs`
: `TRUE/FALSE`. Boolean option to define whether null events should be an option even after an event takes place. If `untilEventOccurrs = TRUE` a conditional logisitc logic is applied in that events are only at risk as long as they have not taken place yet. If `untilEventOccurrs = FALSE` events continue to be at risk after they have occurred. Note that `untilEventOccurrs = TRUE` overwrites the end-Variable, if specified.

`includeAllPossibleEvents`
: `TRUE/FALSE`. Boolean option to allow a more dynamic and specified creation of the risk set. If `includeAllPossibleEvents = TRUE`, a data set has to be provided to `possibleEvents`.

`possibleEvents` An optional data set with the form: column 1 = sender, column 2 = target, 3 = start, 4 = end, 5 = event attribute, 6... . The data set provides all possible events for the entire event sequence and gives each possible event a start and end value to determine when each event could have been possible. This is useful if the risk set follows a complex pattern that cannot be resolved with the above options. E.g., providing a `startDate`-variable and setting `atEventTimesOnly == FALSE` will result in an error since in a continuous time setting the start variable will be matched to the closest date, rather than to the exact value of said date in the event sequence. Manually coding the possible events is neccessary.

`returnInputData`
: `TRUE/FALSE`. Boolean option to check the original data set (handed over in `data`) against the created start and stop variables. If `returnInputData = TRUE`, a list of two data sets is returned. The first data set is the counting process data set with null-events, the second the modified `data`.

Details

To follow.

Author(s)

Laurence Brandenberger <`laurence.brandenberger@eawag.ch`>

See Also

rem-package

Examples

```
## Example 1: standard conditional logistic set-up
dt <- data.frame(
  sender = c('a', 'c', 'd', 'a', 'a', 'f', 'c'),
  target = c('b', 'd', 'd', 'b', 'b', 'a', 'd'),
  eventSequence = c(1, 2, 2, 3, 3, 4, 6)
)
count.data <- createRemDataset(
  data = dt, sender = dt$sender,
  target = dt$target, eventSequence = dt$eventSequence,
  eventAttribute = NULL, time = NULL,
  start = NULL, startDate = NULL,
  end = NULL, endDate = NULL,
  timeformat = NULL,
  atEventTimesOnly = TRUE, untilEventOccurrs = TRUE,
  includeAllPossibleEvents = FALSE, possibleEvents = NULL,
  returnInputData = FALSE)

## Example 2: add 2 attributes to the event-classification
dt <- data.frame(
  sender = c('a', 'c', 'd', 'a', 'a', 'f', 'c'),
  target = c('b', 'd', 'd', 'b', 'b', 'a', 'd'),
  pro.con = c('pro', 'pro', 'con', 'pro', 'con', 'pro', 'pro'),
  attack = c('yes', 'no', 'no', 'yes', 'yes', 'no', 'yes'),
  eventSequence = c(1, 2, 2, 3, 3, 4, 6)
)
count.data <- createRemDataset(
  data = dt, sender = dt$sender,
  target = dt$target, eventSequence = dt$eventSequence,
  eventAttribute = paste0(dt$pro.con, dt$attack), time = NULL,
  start = NULL, startDate = NULL,
  end = NULL, endDate = NULL,
  timeformat = NULL,
  atEventTimesOnly = TRUE, untilEventOccurrs = TRUE,
  includeAllPossibleEvents = FALSE, possibleEvents = NULL,
  returnInputData = FALSE)

## Example 3: adding start and end variables
# Note: the start and end variables will be overwritten
# if there are duplicate events. If you want to
# keep the strict start and stop values that you set, use
# includeAllPossibleEvents = TRUE and specify a
# possibleEvents-data set.
# Note 2: if untilEventOccurrs = TRUE and an end
# variable is provided, this end variable is
# overwritten. Set untilEventOccurrs 0 FALSE and
# provide the end variable if you want the events
# possibilities to stop at these exact event times.
dt <- data.frame(
  sender = c('a', 'c', 'd', 'a', 'a', 'f', 'c'),
  target = c('b', 'd', 'd', 'b', 'b', 'a', 'd'),
  eventSequence = c(1, 2, 2, 3, 3, 4, 6),
  start = c(0, 0, 1, 1, 1, 3, 3),
  end = rep(6, 7)
)
count.data <- createRemDataset(
  data = dt, sender = dt$sender,
  target = dt$target, eventSequence = dt$eventSequence,
  eventAttribute = NULL, time = NULL,
  start = dt$start, startDate = NULL,
  end = dt$end, endDate = NULL,
  timeformat = NULL,
  atEventTimesOnly = TRUE, untilEventOccurrs = TRUE,
  includeAllPossibleEvents = FALSE, possibleEvents = NULL,
  returnInputData = FALSE)

## Example 4: using start (and stop) dates
dt <- data.frame(
  sender = c('a', 'c', 'd', 'a', 'a', 'f', 'c'),
  target = c('b', 'd', 'd', 'b', 'b', 'a', 'd'),
  eventSequence = c(1, 2, 2, 3, 3, 4, 6),
  date = c('01.02.1971', rep('02.02.1971', 2),
rep('03.02.1971', 2), '04.02.1971', '06.02.1971'),
  dateAtRisk = c(rep('21.01.1971', 2), rep('01.02.1971', 5)),
  dateRiskEnds = rep('01.03.1971', 7)
)
count.data <- createRemDataset(
  data = dt, sender = dt$sender, target = dt$target,
  eventSequence = dt$eventSequence,
  eventAttribute = NULL, time = dt$date,
  start = NULL, startDate = dt$dateAtRisk,
  end = NULL, endDate = NULL,
  timeformat = '%d.%m.%Y',
  atEventTimesOnly = TRUE, untilEventOccurrs = TRUE,
  includeAllPossibleEvents = FALSE, possibleEvents = NULL,
  returnInputData = FALSE)
# if you want to include null-events at times when no event happened,
# either see Example 5 or create a start-variable by yourself
# by using the eventSequence()-command with the option
# 'returnDateSequenceData = TRUE' in this package. With the
# generated sequence, dates from startDate can be matched
# to the event sequence values (using the match()-command).

## Example 5: using start and stop dates and including
# possible events whenever no event occurred.
possible.events <- data.frame(
  sender = c('a', 'c', 'd', 'f'),
  target = c('b', 'd', 'd', 'a'),
  start = c(0, 0, 1, 1),
  end = c(rep(8, 4)))
count.data <- createRemDataset(
  data = dt, sender = dt$sender, target = dt$target,
  eventSequence = dt$eventSequence,
  eventAttribute = NULL, time = NULL,
  start = NULL, startDate = NULL,
  end = NULL, endDate = NULL,
  timeformat = NULL,
  atEventTimesOnly = TRUE, untilEventOccurrs = TRUE,
  includeAllPossibleEvents = TRUE, possibleEvents = possible.events,
  returnInputData = FALSE)
# now you can set 'atEventTimesOnly = FALSE' to include
# null-events where none occurred until the events happened
count.data <- createRemDataset(
  data = dt, sender = dt$sender, target = dt$target,
  eventSequence = dt$eventSequence,
  eventAttribute = NULL, time = NULL,
  start = NULL, startDate = NULL,
  end = NULL, endDate = NULL,
  timeformat = NULL,
  atEventTimesOnly = FALSE, untilEventOccurrs = TRUE,
  includeAllPossibleEvents = TRUE, possibleEvents = possible.events,
  returnInputData = FALSE)
# plus you can set  to get the full range of the events
# (bounded by max(possible.events$end))
count.data <- createRemDataset(
  data = dt, sender = dt$sender, target = dt$target,
  eventSequence = dt$eventSequence,
  eventAttribute = NULL, time = NULL,
  start = NULL, startDate = NULL,
  end = NULL, endDate = NULL,
  timeformat = NULL,
  atEventTimesOnly = FALSE, untilEventOccurrs = FALSE,
  includeAllPossibleEvents = TRUE, possibleEvents = possible.events,
  returnInputData = FALSE)
```

degreeStat *Calculate (in/out)-degree statistics*

Description

Calculate the endogenous network statistic `indegree/outdegree` for relational event models. `indegree/outdegree` measures the senders' tendency to be involved in events (sender activity, sender out- or indegree) or the tendency of events to surround a specific target (target popularity, target in- or outdegree)

Usage

```
degreeStat(data, time, degreevar, halflife,
    weight = NULL,
    eventtypevar = NULL,
    eventtypevalue = "valuematch",
    eventfiltervar = NULL,
    eventfiltervalue = NULL,
    eventvar = NULL,
    degreeOnOtherVar = NULL,
    variablename = "degree",
    returnData = FALSE,
    dataPastEvents = NULL,
    showprogressbar = FALSE,
    inParallel = FALSE, cluster = NULL)
```

Arguments

`data`	A data frame containing all the variables.
`time`	Numeric variable that represents the event sequence. The variable has to be sorted in ascending order.
`degreevar`	A string (or factor or numeric) variable that represents the sender or target of the event. The degree statistic will calculate how often in the past, a given sender or target has been active by counting the number of events in the past where the `degreevar` is repeated. See `details` for more information on which variable to chose as `degreevar` for one- and two-mode networks.
`halflife`	A numeric value that is used in the decay function. The vector of past events is weighted by an exponential decay function using the specified halflife. The halflife parameter determines after how long a period the event weight should be halved. E.g. if `halflife = 5`, the weight of an event that occurred 5 units in the past is halved. Smaller halflife values give more importance to more recent events, while larger halflife values should be used if time does not affect the sequence of events that much.
`weight`	An optional numeric variable that represents the weight of each event. If `weight = NULL` each event is given an event weight of `1`.
`eventtypevar`	An optional variable that represents the type of the event. Use `eventtypevalue` to specify how the `eventtypevar` should be used to filter past events.
`eventtypevalue`	An optional value (or set of values) used to specify how paste events should be filtered depending on their type. `eventtypevalue = "valuematch"` indicates that only past events that have the same type should be used to calculate the degree statistic. `eventtypevalue = "valuemix"` indicates that past and present events of specific types should be used for the degree statistic. All the possible combinations of the eventtypevar-values will be used. E.g. if `eventtypevar` contains two unique values "a" and "b", 4 degree statistics will be calculated. The first variable calculates the degree effect where the present event is of type "a" and all the past events are of type "b". The next variable calculates the degree statistic for present events of type "b" and past events of type "a". Additionally, a variable is calculated, where present events as well as past events are of type "a" and a fourth variable calculates the degree statistic for events with type "b" (i.e. valuematch on value "b"). `eventtypevalue = c("..", "..")` is similar to the `"nodemix"`-option, all different combinations of the values specified in `eventtypevalue` are used to create the degree statistics.
`eventfiltervar`	An optional numeric/character/or factor variable for each event. If `eventfiltervar` is specified, `eventfiltervalue` has to be provided as well.
`eventfiltervalue`	An optional character string that represents the value for which past events should be filtered. To filter the current events, use `eventtypevar`.

`eventvar`	An (optional) dummy variable with 0 values for null-events and 1 values for true events. If the `data` is in the form of counting process data, use the `eventvar`-option to specify which variable contains the 0/1-dummy for event occurrence. If this variable is not specified, all events in the past will be considered for the calulation of the degree statistic, regardless if they occurred or not (= are null-events).
`degreeOnOtherVar`	A string (or factor or numeric) variable that represents the sender or target of the event. It can be used to calculate target-outdegree or sender-indegree statistics in one-mode networks. For the sender indegree statistic, fill the sender variable into the `degreevar` and the target variable into the `degree.on.other.var`. For the target-outdegree statistic, fill the target variable into the `degreevar` and the sender variable into the `degree.on.other.var`.
`variablename`	An optional value (or values) with the name the degree statistic variable should be given. Default "degree" is used. To be used if `returnData = TRUE` or multiple degree statistics are calculated.
`returnData`	`TRUE/FALSE`. Set to `FALSE` by default. The new variable(s) are bound directly to the `data.frame` provided and the data frame is returned in full.
`dataPastEvents`	An optional `data.frame` with the following variables: column 1 = time variable, column 2 = degree variable, column 3 = degree on other variable (or all "1"), column 4 = event dummy (or all 1), column 5 = weight variable (or all "1"), column 6 = event type variable (or all "1"), column 7 = event filter variable (or all "1").
`showprogressbar`	`TRUE/FALSE`. Can only be set to TRUE if the function is not run in parallel.
`inParallel`	`TRUE/FALSE`. An optional boolean to specify if the loop should be run in parallel.
`cluster`	An optional numeric or character value that defines the cluster. By specifying a single number, the cluster option uses the provided number of nodes to parallellize. By specifying a cluster using the `makeCluster`-command in the `doParallel`-package, the loop can be run on multiple nodes/cores. E.g., `cluster = makeCluster(12, type="FORK")`.

Details

The `degreeStat()`-function calculates an endogenous statistic that measures whether events have a tendency to include either the same sender or the same target over the entire event sequence.

The effect is calculated as follows.

$$G_t = G_t(E) = (A, B, w_t),$$

G_t represents the network of past events and includes all events E. These events consist each of a sender $a \in A$ and a target $b \in B$ (in one-mode networks $A = B$) and a weight function w_t:

$$w_t(i,j) = \sum_{e:a=i,b=j} |w_e| \cdot e^{-(t-t_e)\cdot \frac{ln(2)}{T_{1/2}}} \cdot \frac{ln(2)}{T_{1/2}},$$

where w_e is the event weight (usually a constant set to 1 for each event), t is the current event time, t_e is the past event time and $T_{1/2}$ is a halflife parameter.

For the degree effect, the past events G_t are filtered to include only events where the senders or targets are identical to the current sender or target.

$$sender - outdegree(G_t, a, b) = \sum_{j \in B} w_t(a, j)$$

$$target - indegree(G_t, a, b) = \sum_{i \in A} w_t(i, b)$$

$$sender - indegree(G_t, a, b) = \sum_{i \in A} w_t(i, a)$$

$$target - outdegree(G_t, a, b) = \sum_{j \in B} w_t(b, j)$$

Depending on whether the degree statistic is measured on the sender variable or the target variable, either activity or popularity effects are calculated.

For one-mode networks: Four distinct statistics can be calculated: sender-indegree, sender-outdegree, target-indegree or target-outdegree. The sender-indegree measures how often the current sender was targeted by other senders in the past (i.e. how popular were current senders). The sender-outedegree measures how often the current sender was involved in an event, where they were also marked as sender (i.e. how active the current sender has been in the past). The target-indegree statistic measures how often the current targets were targeted in the past (i.e. how popular were current targets). And the target-outdegree measures how often the current targets were senders in the past (i.e. how active were current targets in the past).

For two-mode networks: Two distinct statistics can be calculated: sender-outdegree and target-indegree. Sender-outdegree measures how often the current sender has been involved in an event in the past (i.e. how active the sender has been up until now). The target-indegree statistic measures how often the current target has been involved in an event in the past (i.e. how popular a given target has been before the current event).

An exponential decay function is used to model the effect of time on the endogenous statistics. Each past event that contains the same sender or the same target (depending on the variable specified in `degreevar`) and fulfills additional filtering options (specified via event type or event attributes) is weighted with an exponential decay. The further apart the past event is from the present event, the less weight is given to this event. The halflife parameter in the `degreeStat()`-function determines at which rate the weights of past events should be reduced.

The `eventtypevar`- and `eventattributevar`-options help filter the past events more specifically. How they are filtered depends on the `eventtypevalue`- and `eventattributevalue`-option.

Author(s)

Laurence Brandenberger <`laurence.brandenberger@eawag.ch`>

See Also

rem-package

Examples

```
# create some data with 'sender', 'target' and a 'time'-variable
# (Note: Data used here are random events from the Correlates of War Project)
sender <- c('TUN', 'NIR', 'NIR', 'TUR', 'TUR', 'USA', 'URU',
            'IRQ', 'MOR', 'BEL', 'EEC', 'USA', 'IRN', 'IRN',
            'USA', 'AFG', 'ETH', 'USA', 'SAU', 'IRN', 'IRN',
            'ROM', 'USA', 'USA', 'PAN', 'USA', 'USA', 'YEM',
            'SYR', 'AFG', 'NAT', 'NAT', 'USA')
target <- c('BNG', 'ZAM', 'JAM', 'SAU', 'MOM', 'CHN', 'IRQ',
            'AFG', 'AFG', 'EEC', 'BEL', 'ITA', 'RUS', 'UNK',
            'IRN', 'RUS', 'AFG', 'ISR', 'ARB', 'USA', 'USA',
            'USA', 'AFG', 'IRN', 'IRN', 'IRN', 'AFG', 'PAL',
            'ARB', 'USA', 'EEC', 'BEL', 'PAK')
time <- c('800107', '800107', '800107', '800109', '800109',
          '800109', '800111', '800111', '800111', '800113',
          '800113', '800113', '800114', '800114', '800114',
          '800116', '800116', '800116', '800119', '800119',
          '800119', '800122', '800122', '800122', '800124',
          '800125', '800125', '800127', '800127', '800127',
          '800204', '800204', '800204')
type <- sample(c('cooperation', 'conflict'), 33,
               replace = TRUE)

# combine them into a data.frame
dt <- data.frame(sender, target, time, type)

# create event sequence and order the data
dt <- eventSequence(datevar = dt$time, dateformat = "%y%m%d",
                    data = dt, type = "continuous",
                    byTime = "daily", returnData = TRUE,
                    sortData = TRUE)

# create counting process data set (with null-events) - conditional logit setting
dts <- createRemDataset(dt, dt$sender, dt$target, dt$event.seq.cont,
                          eventAttribute = dt$type,
                          atEventTimesOnly = TRUE, untilEventOccurrs = TRUE,
  returnInputData = TRUE)
## divide up the results: counting process data = 1, original data = 2
dtrem <- dts[[1]]
dt <- dts[[2]]
## merge all necessary event attribute variables back in
dtrem$type <- dt$type[match(dtrem$eventID, dt$eventID)]
dtrem$important <- dt$important[match(dtrem$eventID, dt$eventID)]
# manually sort the data set
dtrem <- dtrem[order(dtrem$eventTime), ]

# calculate sender-outdegree statistic
dtrem$sender.outdegree <- degreeStat(data = dtrem,
                                      time = dtrem$eventTime,
                                      degreevar = dtrem$sender,
                                      halflife = 2,
                                      eventvar = dtrem$eventDummy,
                                      returnData = FALSE)

# plot sender-outdegree over time
library("ggplot2")
ggplot(dtrem, aes(eventTime, sender.outdegree,
                  group = factor(eventDummy), color = factor(eventDummy) ) ) +
  geom_point()+ geom_smooth()

# calculate sender-indegree statistic
dtrem$sender.indegree <- degreeStat(data = dtrem,
                                    time = dtrem$eventTime,
                                    degreevar = dtrem$sender,
                                    halflife = 2,
                                    eventvar = dtrem$eventDummy,
                                    degreeOnOtherVar = dtrem$target,
                                    returnData = FALSE)

# calculate target-indegree statistic
dtrem$target.indegree <- degreeStat(data = dtrem,
                                     time = dtrem$eventTime,
                                     degreevar = dtrem$target,
                                     halflife = 2,
                                     eventvar = dtrem$eventDummy,
                                     returnData = FALSE)

# calculate target-outdegree statistic
dtrem$target.outdegree <- degreeStat(data = dtrem,
                                    time = dtrem$eventTime,
                                    degreevar = dtrem$target,
                                    halflife = 2,
                                    eventvar = dtrem$eventDummy,
                                    degreeOnOtherVar = dtrem$sender,
                                    returnData = FALSE)

# calculate target-indegree with typematch
dtrem$target.indegree.tm <- degreeStat(data = dtrem,
                                    time = dtrem$eventTime,
                                    degreevar = dtrem$target,
                                    halflife = 2,
                                    eventtypevar = dtrem$type,
                                    eventtypevalue = "valuematch",
                                    eventvar = dtrem$eventDummy,
                                    returnData = FALSE)
```

eventSequence *Create event sequence*

Description

Create the event sequence for relational event models. Continuous or ordinal sequences can be created. Various dates may be excluded from the sequence (e.g. special holidays, specific weekdays or longer time spans).

Usage

```
eventSequence(datevar,
    dateformat = NULL, data = NULL,
    type = "continuous", byTime = "daily",
    excludeDate = NULL, excludeTypeOfDay = NULL,
    excludeYear = NULL, excludeFrom = NULL,
    excludeTo = NULL, returnData = FALSE,
    sortData = FALSE,
    returnDateSequenceData = FALSE)
```

Arguments

`datevar`	The variable containing the information on the date and/or time of the event.
`dateformat`	A character string indicating the format of the `datevar`. see `as.Date`
`data`	An optional data frame containing all the variables.
`type`	"‘`continuous`"’ or "‘`ordinal`"’. Specifies whether the event sequence is to be created as a continuous sequence or an ordinal sequence.
`byTime`	String value. Specifies at what interval the event sequence is created. Use "daily", "monthly" or "yearly".
`excludeDate`	An optional string or string vector containing one or more dates that should be excluded from the event.sequence. The dates have to be in the same format as provided in `dateformat`. Only valid for continuous event sequences.
`excludeTypeOfDay`	String value or vector naming the day(s) that should be excluded from the event sequence. Depending on the locale the weekdays may be named differently. Use `Sys.getlocale("LC_TIME")` to find which locale is installed.
`excludeYear`	A string value or vector naming the year(s) that should be excluded from the event sequence.
`excludeFrom`	A string value (or a vector of strings) with the start value of the date from (from-value included) which the event sequence should not be affected. The value has to be in the same format as specified in `dateformat`.
`excludeTo`	A string value (or a vector of strings) with the end value of the date to which time the event sequence should not be affected (to-value included). The value has to be in the same format as specified in `dateformat`.
`returnData`	`TRUE/FALSE`. Default set to `FALSE`. The data frame provided is returned in full, together with the new variable for the event sequence.
`sortData`	`TRUE/FALSE`. Default set to `FALSE`. Should only be used if `returnData = TRUE`. The entire data.frame will be ordered according to the event sequence.

`returnDateSequenceData`
: `TRUE/FALSE`. Boolean option to return the full information on which date matches to which sequence number instead of the event sequence (and corresponding data frame).

Details

In order to estimate relational event models, the events have to be ordered, either according to an ordinal or a continuous event sequence. The ordinal event sequence simply orders the events and gives each event a place in the sequence. The continuous event sequence creates an artificial sequence ranging from `min(datevar)` to `max(datevar)` and matches each event with its place in the artificial event sequence. Dates, years or Weekdays can be excluded from the artificial event sequence. This is useful for excluding specific holidays, weekends etc..

Where two or more events occur at the same time, they are given the same value in the event sequence.

Author(s)

Laurence Brandenberger <`laurence.brandenberger@eawag.ch`>

See Also

rem-package

Examples

```
# create some data with 'sender', 'target' and a 'time'-variable
# (Note: Data used here are random events from the Correlates of War Project)
sender <- c('TUN', 'NIR', 'NIR', 'TUR', 'TUR', 'USA', 'URU',
            'IRQ', 'MOR', 'BEL', 'EEC', 'USA', 'IRN', 'IRN',
            'USA', 'AFG', 'ETH', 'USA', 'SAU', 'IRN', 'IRN',
            'ROM', 'USA', 'USA', 'PAN', 'USA', 'USA', 'YEM',
            'SYR', 'AFG', 'NAT', 'NAT', 'USA')
target <- c('BNG', 'ZAM', 'JAM', 'SAU', 'MOM', 'CHN', 'IRQ',
            'AFG', 'AFG', 'EEC', 'BEL', 'ITA', 'RUS', 'UNK',
            'IRN', 'RUS', 'AFG', 'ISR', 'ARB', 'USA', 'USA',
            'USA', 'AFG', 'IRN', 'IRN', 'IRN', 'AFG', 'PAL',
            'ARB', 'USA', 'EEC', 'BEL', 'PAK')
time <- c('800107', '800107', '800107', '800109', '800109',
          '800109', '800111', '800111', '800111', '800113',
          '800113', '800113', '800114', '800114', '800114',
          '800116', '800116', '800116', '800119', '800119',
          '800119', '800122', '800122', '800122', '800124',
          '800125', '800125', '800127', '800127', '800127',
          '800204', '800204', '800204')

# combine them into a data.frame
dt <- data.frame(sender, target, time)

# create continuous event sequence: return the data with the
# event sequence and sort the data according to the event sequence.
```

```
dt <- eventSequence(datevar = dt$time, dateformat = '%y%m%d',
                    data = dt, type = 'continuous',
                    byTime = 'daily', returnData = TRUE,
                    sortData = TRUE)

# alternative : create variable with the continuous event
# sequence, unsorted
dt$eventSeq <- eventSequence(datevar = dt$time,
                             dateformat = '%y%m%d',
                             data = dt, type = 'continuous',
                             byTime = 'daily',
                             returnData = FALSE,
                             sortData = FALSE)
# manually sort the data set
dt <- dt[order(dt$eventSeq), ]

# create the sequence by month
dt$eventSeqMonthly <- eventSequence(datevar = dt$time,
                                    dateformat = '%y%m%d',
                                    data = dt,
                                    type = 'continuous',
                                    byTime = 'monthly',
                                    returnData = FALSE,
                                    sortData = FALSE)

# create the sequence by year
dt$eventSeqYearly <- eventSequence(datevar = dt$time,
                                   dateformat = '%y%m%d',
                                   data = dt,
                                   type = 'continuous',
                                   byTime = 'yearly',
                                   returnData = FALSE,
                                   sortData = FALSE)

# create an ordinal event sequence
dt$eventSeqOrdinal <- eventSequence(datevar = dt$time,
                                    dateformat = '%y%m%d',
                                    data = dt,
                                    type = 'ordinal',
                                    byTime = 'daily',
                                    returnData = FALSE,
                                    sortData = FALSE)

# exclude certain dates
dt$eventSeqEx <- eventSequence(datevar = dt$time,
                               dateformat = '%y%m%d',
                               data = dt, type = 'continuous',
                               byTime = 'daily',
                               excludeDate = c('800108', '800112'),
                               returnData = FALSE,
                               sortData = FALSE)

# return the sequence data set, where all values in the event sequence
# correspond to the date of the events. Useful to calculate
# start-variables for the createRemDataset-command.
seq.data <- eventSequence(datevar = dt$time,
                          dateformat = "%y%m%d",
                          data = dt, type = "continuous",
                          byTime = "daily",
                          excludeDate = c("800108", "800112"),
                          returnData = FALSE,
                          sortData = FALSE,
                          returnDateSequenceData = TRUE)
```

---

fourCycleStat *Calculate four cycle statistics*

Description

Calculate the endogenous network statistic `fourCycle` that measures the tendency for events to close four cycles in two-mode event sequences.

Usage

```
fourCycleStat(data, time, sender, target, halflife,
    weight = NULL,
    eventtypevar = NULL,
    eventtypevalue = 'standard',
    eventfiltervar = NULL,
    eventfilterAB = NULL, eventfilterAJ = NULL,
    eventfilterIB = NULL, eventfilterIJ = NULL,
    eventvar = NULL,
    variablename = 'fourCycle',
    returnData = FALSE,
    dataPastEvents = NULL,
    showprogressbar = FALSE,
    inParallel = FALSE, cluster = NULL
)
```

Arguments

`data`	A data frame containing all the variables.
`time`	Numeric variable that represents the event sequence. The variable has to be sorted in ascending order.
`sender`	A string (or factor or numeric) variable that represents the sender of the event.
`target`	A string (or factor or numeric) variable that represents the target of the event.
`halflife`	A numeric value that is used in the decay function. The vector of past events is weighted by an exponential decay function using the specified halflife. The halflife parameter determins after how long a period the event weight should be halved. E.g. if `halflife = 5`, the weight of an event that occured 5 units in

	the past is halved. Smaller halflife values give more importance to more recent events, while larger halflife values should be used if time does not affect the sequence of events that much.
`weight`	An optional numeric variable that represents the weight of each event. If `weight = NULL` each event is given an event weight of `1`.
`eventtypevar`	An optional variable that represents the type of the event. Use `eventtypevalue` to specify how the `eventtypevar` should be used to filter past events.
`eventtypevalue`	An optional value (or set of values) used to specify how paste events should be filtered depending on their type. `'standard'`, `'positive'` or `'negative'` may be used. Default set to `'standard'`. `'standard'` referrs to closing four cylces where the type of the events is irrelevant. `'positive'` closing four cycles can be classified as reciprocity via the second mode. It indicates whether senders have a tendency to reciprocate or show support by engaging in targets that close a four cycle between two senders. `'negative'` closing four cycles represent opposition between two senders, where the current event is more likely if the two senders have opposed each other in the past. Support or opposition is represented by the `eventtypevar` value for each event.
`eventfiltervar`	An optinoal variable that allows filtering of past events using an event attribute. It can be a sender attribute, a target attribute, time or dyad attribute. Use `eventfilterAB`, `eventfilterAJ`, `eventfilterIB` or `eventfilterIJ` to specify how the `eventfiltervar` should be used.
`eventfilterAB`	An optional value used to specify how paste events should be filtered depending on their attribute. Each distinct edge that form a four cycle can be filtered. `eventfilterAB` refers to the current event. `eventfilterAJ` refers to the event involving the current sender and target `j` that has been used by the current as well as the second actor in the past. `eventfilterIB` refers to the event involving the second sender and the current target. `eventfilterIJ` filters events that involve the second sender and the second target. See the four cycle formula in the `details` section for more information.
`eventfilterAJ`	see `eventfilterAB`.
`eventfilterIB`	see `eventfilterAB`.
`eventfilterIJ`	see `eventfilterAB`.
`eventvar`	An optional dummy variable with 0 values for null-events and 1 values for true events. If the `data` is in the form of counting process data, use the `eventvar`-option to specify which variable contains the 0/1-dummy for event occurrence. If this variable is not specified, all events in the past will be considered for the calulation of the four cycle statistic, regardless if they occurred or not (= are null-events). Misspecification could result in grievous errors in the calculation of the network statistic.
`variablename`	An optional value (or values) with the name the four cycle statistic variable should be given. To be used if `returnData = TRUE`.
`returnData`	`TRUE`/`FALSE`. Set to `FALSE` by default. The new variable(s) are bound directly to the `data.frame` provided and the data frame is returned in full.
`dataPastEvents`	An optional `data.frame` with the following variables: column 1 = time variable, column 2 = sender variable, column 3 = target on other variable (or all "1"),

column 4 = weight variable (or all "1"), column 5 = event type variable (or all "1"), column 6 = event filter variable (or all "1"). Make sure that the data frame does not contain null events. Filter it out for true events only.

`showprogressbar`
: `TRUE/FALSE`. To be implemented.

`inParallel`
: `TRUE/FALSE`. An optional boolean to specify if the loop should be run in parallel.

`cluster`
: An optional numeric or character value that defines the cluster. By specifying a single number, the cluster option uses the provided number of nodes to parallellize. By specifying a cluster using the `makeCluster`-command in the `doParallel`-package, the loop can be run on multiple nodes/cores. E.g., `cluster = makeCluster(12, type="FORK")`.

Details

The `fourCycleStat()`-function calculates an endogenous statistic that measures whether events have a tendency to form four cycles.

The effect is calculated as follows:

$$G_t = G_t(E) = (A, B, w_t),$$

G_t represents the network of past events and includes all events E. These events consist each of a sender $a \in A$ and a target $b \in B$ and a weight function w_t:

$$w_t(i,j) = \sum_{e:a=i,b=j} |w_e| \cdot e^{-(t-t_e)\cdot\frac{ln(2)}{T_{1/2}}} \cdot \frac{ln(2)}{T_{1/2}},$$

where w_e is the event weight (usually a constant set to 1 for each event), t is the current event time, t_e is the past event time and $T_{1/2}$ is a halflife parameter.

For the four-cylce effect, the past events G_t are filtered to include only events where the current event closes an open four-cycle in the past.

$$fourCycle(G_t, a, b) = \sqrt[3]{\sum_{i \in A \& j \in B} w_t(a,j) \cdot w_t(i,b) \cdot w_t(i,j)}$$

An exponential decay function is used to model the effect of time on the endogenous statistics. The further apart the past event is from the present event, the less weight is given to this event. The halflife parameter in the `fourCycleStat()`-function determins at which rate the weights of past events should be reduced. Therefore, if the one (or more) of the three events in the four cycle have ocurred further in the past, less weight is given to this four cycle because it becomes less likely that the two senders reacted to each other in the way the four cycle assumes.

The `eventtypevar`- and `eventfiltervar`-options help filter the past events more specifically. How they are filtered depends on the `eventtypevalue`- and `eventfilter__`-option.

Author(s)

Laurence Brandenberger <`laurence.brandenberger@eawag.ch`>

See Also

rem-package

Examples

```
# create some data two-mode network event sequence data with
# a 'sender', 'target' and a 'time'-variable
sender <- c('A', 'B', 'A', 'C', 'A', 'D', 'F', 'G', 'A', 'B',
            'B', 'C', 'D', 'E', 'F', 'B', 'C', 'D', 'E', 'C',
            'A', 'F', 'E', 'B', 'C', 'E', 'D', 'G', 'A', 'G',
            'F', 'B', 'C')
target <- c('T1', 'T2', 'T3', 'T2', 'T1', 'T4', 'T6', 'T2',
            'T4', 'T5', 'T5', 'T5', 'T1', 'T6', 'T7', 'T2',
            'T3', 'T1', 'T1', 'T4', 'T5', 'T6', 'T8', 'T2',
            'T7', 'T1', 'T6', 'T7', 'T3', 'T4', 'T7', 'T8', 'T2')
time <- c('03.01.15', '04.01.15', '10.02.15', '28.02.15', '01.03.15',
          '07.03.15', '07.03.15', '12.03.15', '04.04.15', '28.04.15',
          '06.05.15', '11.05.15', '13.05.15', '17.05.15', '22.05.15',
          '09.08.15', '09.08.15', '14.08.15', '16.08.15', '29.08.15',
          '05.09.15', '25.09.15', '02.10.15', '03.10.15', '11.10.15',
          '18.10.15', '20.10.15', '28.10.15', '04.11.15', '09.11.15',
          '10.12.15', '11.12.15', '12.12.15')
type <- sample(c('con', 'pro'), 33, replace = TRUE)
important <- sample(c('important', 'not important'), 33,
                    replace = TRUE)

# combine them into a data.frame
dt <- data.frame(sender, target, time, type, important)

# create event sequence and order the data
dt <- eventSequence(datevar = dt$time, dateformat = '%d.%m.%y',
                    data = dt, type = 'continuous',
                    byTime = "daily", returnData = TRUE,
                    sortData = TRUE)

# create counting process data set (with null-events) - conditional logit setting
dts <- createRemDataset(dt, dt$sender, dt$target, dt$event.seq.cont,
                          eventAttribute = dt$type,
                          atEventTimesOnly = TRUE, untilEventOccurrs = TRUE,
  returnInputData = TRUE)
## divide up the results: counting process data = 1, original data = 2
dtrem <- dts[[1]]
dt <- dts[[2]]
## merge all necessary event attribute variables back in
dtrem$type <- dt$type[match(dtrem$eventID, dt$eventID)]
dtrem$important <- dt$important[match(dtrem$eventID, dt$eventID)]
# manually sort the data set
dtrem <- dtrem[order(dtrem$eventTime), ]

# calculate closing four-cycle statistic
dtrem$fourCycle <- fourCycleStat(data = dtrem,
                              time = dtrem$eventTime,
                              sender = dtrem$sender,
                              target = dtrem$target,
                              eventvar = dtrem$eventDummy,
                              halflife = 20)

# plot closing four-cycles over time:
library("ggplot2")
ggplot(dtrem, aes (eventTime, fourCycle,
                group = factor(eventDummy), color = factor(eventDummy)) ) +
  geom_point()+ geom_smooth()

# calculate positive closing four-cycles: general support
dtrem$fourCycle.pos <- fourCycleStat(data = dtrem,
                                 time = dtrem$eventTime,
                                 sender = dtrem$sender,
                                 target = dtrem$target,
                                 eventvar = dtrem$eventDummy,
                                 eventtypevar = dtrem$type,
                                 eventtypevalue = 'positive',
                                 halflife = 20)

# calculate negative closing four-cycles: general opposition
dtrem$fourCycle.neg <- fourCycleStat(data = dtrem,
                                     time = dtrem$eventTime,
                                     sender = dtrem$sender,
                                     target = dtrem$target,
                                     eventvar = dtrem$eventDummy,
                                     eventtypevar = dtrem$type,
                                     eventtypevalue = 'negative',
                                     halflife = 20)
```

`inertiaStat` *Calculate inertia statistics*

Description

Calculate the endogenous network statistic `inertia` for relational event models. `inertia` measures the tendency for events to consist of the same sender and target (i.e. repeated events).

Usage

```
inertiaStat(data, time, sender, target, halflife,
    weight = NULL,
    eventtypevar = NULL,
    eventtypevalue = "valuematch",
    eventfiltervar = NULL,
    eventfiltervalue = NULL,
    eventvar = NULL,
    variablename = "inertia",
    returnData = FALSE,
    showprogressbar = FALSE,
    inParallel = FALSE, cluster = NULL)
```

Arguments

`data`	A data frame containing all the variables.
`time`	Numeric variable that represents the event sequence. The variable has to be sorted in ascending order.
`sender`	A string (or factor or numeric) variable that represents the sender of the event.
`target`	A string (or factor or numeric) variable that represents the target of the event.
`halflife`	A numeric value that is used in the decay function. The vector of past events is weighted by an exponential decay function using the specified halflife. The halflife parameter determins after how long a period the event weight should be halved. E.g. if `halflife = 5`, the weight of an event that occured 5 units in the past is halved. Smaller halflife values give more importance to more recent events, while larger halflife values should be used if time does not affect the sequence of events that much.
`weight`	An optional numeric variable that represents the weight of each event. If `weight = NULL` each event is given an event weight of `1`.
`eventtypevar`	An optional variable that represents the type of the event. Use `eventtypevalue` to specify how the `eventtypevar` should be used to filter past events.
`eventtypevalue`	An optional value (or set of values) used to specify how paste events should be filtered depending on their type. `eventtypevalue = "valuematch"` indicates that only past events that have the same type as the current event should be used to calculate the inertia statistic. `eventtypevalue = "valuemix"` indicates that past and present events of specific types should be used for the inertia statistic. All the possible combinations of the eventtypevar-values will be used. E.g. if `eventtypevar` contains two unique values "a" and "b", 4 inertia statistics will be calculated. The first variable calculates the inertia effect where the present event is of type "a" and all the past events are of type "b". The next variable calculates inertia for present events of type "b" and past events of type "a". Additionally, a variable is calculated, where present events as well as past events are of type "a" and a fourth variable calculates inertia for events with type "b" (i.e. valuematch on value "b"). `eventtypevalue = c(.., ..)` is similar to the `"nodmix"`-option, all different combinations of the values specified in `eventtypevalue` are used to create inertia statistics.
`eventfiltervar`	An optional numeric/character/or factor variable for each event. If `eventfiltervar` is specified, `eventfiltervalue` has to be provided as well.
`eventfiltervalue`	An optional character string that represents the value for which past events should be filtered. To filter the current events, use `eventtypevar`.
`eventvar`	An optional dummy variable with 0 values for null-events and 1 values for true events. If the `data` is in the form of counting process data, use the `eventvar`-option to specify which variable contains the 0/1-dummy for event occurrence. If this variable is not specified, all events in the past will be considered for the calulation of the inertia statistic, regardless if they occurred or not (= are null-events).

`variablename`	An optional value (or values) with the name the inertia statistic variable should be given. To be used if `returnData = TRUE` or multiple inertia statistics are calculated.
`returnData`	`TRUE/FALSE`. Set to `FALSE` by default. The new variable(s) are bound directly to the `data.frame` provided and the data frame is returned in full.
`showprogressbar`	`TRUE/FALSE`. Can only be set to TRUE if the function is not run in parallel.
`inParallel`	`TRUE/FALSE`. An optional boolean to specify if the loop should be run in parallel.
`cluster`	An optional numeric or character value that defines the cluster. By specifying a single number, the cluster option uses the provided number of nodes to parallellize. By specifying a cluster using the `makeCluster`-command in the `doParallel`-package, the loop can be run on multiple nodes/cores. E.g., `cluster = makeCluster(12, type="FORK")`.

Details

The `inertiaStat()`-function calculates an endogenous statistic that measures whether events have a tendency to be repeated with the same sender and target over the entire event sequence.

The effect is calculated as follows.

$$G_t = G_t(E) = (A, B, w_t),$$

G_t represents the network of past events and includes all events E. These events consist each of a sender $a \in A$ and a target $b \in B$ and a weight function w_t:

$$w_t(i,j) = \sum_{e:a=i,b=j} |w_e| \cdot e^{-(t-t_e)\cdot \frac{ln(2)}{T_{1/2}}} \cdot \frac{ln(2)}{T_{1/2}},$$

where w_e is the event weight (usually a constant set to 1 for each event), t is the current event time, t_e is the past event time and $T_{1/2}$ is a halflife parameter.

For the inertia effect, the past events G_t are filtered to include only events where the senders and targets are identical to the current sender and target.

$$inertia(G_t, a, b) = w_t(a, b)$$

An exponential decay function is used to model the effect of time on the endogenous statistics. Each past event that contains the same sender and target and fulfills additional filtering options specivied via event type or event attributes is weighted with an exponential decay. The further apart the past event is from the present event, the less weight is given to this event. The halflife parameter in the `inertiaStat()`-function determins at which rate the weights of past events should be reduced.

The `eventfiltervar`- and `eventtypevar`-options help filter the past events more specifically. How they are filtered depends on the `eventfiltervalue`- and `eventtypevalue`-option.

Author(s)

Laurence Brandenberger <`laurence.brandenberger@eawag.ch`>

See Also

rem-package

Examples

```
# create some data with 'sender', 'target' and a 'time'-variable
# (Note: Data used here are random events from the Correlates of War Project)
sender <- c('TUN', 'NIR', 'NIR', 'TUR', 'TUR', 'USA', 'URU',
            'IRQ', 'MOR', 'BEL', 'EEC', 'USA', 'IRN', 'IRN',
            'USA', 'AFG', 'ETH', 'USA', 'SAU', 'IRN', 'IRN',
            'ROM', 'USA', 'USA', 'PAN', 'USA', 'USA', 'YEM',
            'SYR', 'AFG', 'NAT', 'NAT', 'USA')
target <- c('BNG', 'ZAM', 'JAM', 'SAU', 'MOM', 'CHN', 'IRQ',
            'AFG', 'AFG', 'EEC', 'BEL', 'ITA', 'RUS', 'UNK',
            'IRN', 'RUS', 'AFG', 'ISR', 'ARB', 'USA', 'USA',
            'USA', 'AFG', 'IRN', 'IRN', 'IRN', 'AFG', 'PAL',
            'ARB', 'USA', 'EEC', 'BEL', 'PAK')
time <- c('800107', '800107', '800107', '800109', '800109',
          '800109', '800111', '800111', '800111', '800113',
          '800113', '800113', '800114', '800114', '800114',
          '800116', '800116', '800116', '800119', '800119',
          '800119', '800122', '800122', '800122', '800124',
          '800125', '800125', '800127', '800127', '800127',
          '800204', '800204', '800204')
type <- sample(c('cooperation', 'conflict'), 33,
               replace = TRUE)

# combine them into a data.frame
dt <- data.frame(sender, target, time, type)

# create event sequence and order the data
dt <- eventSequence(datevar = dt$time, dateformat = "%y%m%d",
                    data = dt, type = "continuous",
                    byTime = "daily", returnData = TRUE,
                    sortData = TRUE)

# create counting process data set (with null-events) - conditional logit setting
dts <- createRemDataset(dt, dt$sender, dt$target,
dt$event.seq.cont, eventAttribute = dt$type,
atEventTimesOnly = TRUE, untilEventOccurrs = TRUE,
returnInputData = TRUE)
## divide up the results: counting process data = 1, original data = 2
dtrem <- dts[[1]]
dt <- dts[[2]]
## merge all necessary event attribute variables back in
dtrem$type <- dt$type[match(dtrem$eventID, dt$eventID)]
# manually sort the data set
dtrem <- dtrem[order(dtrem$eventTime), ]

# manually sort the data set
dtrem <- dtrem[order(dtrem$eventTime), ]
```

```
# calculate inertia statistics
dtrem$inertia <- inertiaStat(data = dtrem, time = dtrem$eventTime,
                            sender = dtrem$sender, target = dtrem$target,
                            eventvar = dtrem$eventDummy,
                            halflife = 2, returnData = FALSE,
                            showprogressbar = FALSE)

# plot inertia over time
library("ggplot2")
ggplot(dtrem, aes ( eventTime, inertia,
group = factor(eventDummy), color = factor(eventDummy)) ) +
geom_point() + geom_smooth()

# inertia with typematch (e.g. for 'cooperation' events only count
# past 'cooperation' events)
dtrem$inertia.tm <- inertiaStat(data = dtrem, time = dtrem$eventTime,
                               sender = dtrem$sender, target = dtrem$target,
                               eventvar = dtrem$eventDummy,
                               halflife = 2,
                               eventtypevar = dtrem$type,
                               eventtypevalue = "valuematch",
                               returnData = FALSE,
                               showprogressbar = FALSE)

# inertia with valuemix: for each combination of types
# in the eventtypevar, create a variable
dtrem <- inertiaStat(data = dtrem, time = dtrem$eventTime,
              sender = dtrem$sender, target = dtrem$target,
              eventvar = dtrem$eventDummy,
              halflife = 2,
              eventtypevar = dtrem$type,
              eventtypevalue = "valuemix",
              returnData = TRUE,
              showprogressbar = FALSE)
```

reciprocityStat *Calculate reciprocity statistics*

Description

Calculate the endogenous network statistic `reciprocity` for relational event models. `reciprocity` measures the tendency for senders to reciprocate prior events where they were targeted by other senders. One-mode network statistic only.

Usage

```
reciprocityStat(data, time, sender, target, halflife,
    weight = NULL,
    eventtypevar = NULL,
    eventtypevalue = "valuematch",
    eventfiltervar = NULL,
    eventfiltervalue = NULL,
    eventvar = NULL,
    variablename = "recip",
    returnData = FALSE,
    showprogressbar = FALSE,
    inParallel = FALSE, cluster = NULL)
```

Arguments

`data`	A data frame containing all the variables.
`time`	Numeric variable that represents the event sequence. The variable has to be sorted in ascending order.
`sender`	A string (or factor or numeric) variable that represents the sender of the event.
`target`	A string (or factor or numeric) variable that represents the target of the event.
`halflife`	A numeric value that is used in the decay function. The vector of past events is weighted by an exponential decay function using the specified halflife. The halflife parameter determines after how long a period the event weight should be halved. E.g. if `halflife = 5`, the weight of an event that occurred 5 units in the past is halved. Smaller halflife values give more importance to more recent events, while larger halflife values should be used if time does not affect the time between events that much.
`weight`	An optional numeric variable that represents the weight of each event. If `weight = NULL` each event is given an event weight of `1`.
`eventtypevar`	An optional variable that represents the type of the event. Use `eventtypevalue` to specify how the `eventtypevar` should be used to filter past events.
`eventtypevalue`	An optional value (or set of values) used to specify how paste events should be filtered depending on their type. `eventtypevalue = "valuematch"` indicates that only past events that have the same type as the current event should be used to calculate the reciprocity statistic. `eventtypevalue = "valuemix"` indicates that past and present events of specific types should be used for the reciprocity statistic. All the possible combinations of the eventtypevar-values will be used. E.g. if `eventtypevar` contains three unique values "a" and "b", 4 reciprocity statistics will be calculated. The first variable calculates the reciprocity effect where the present event is of type "a" and all the past events are of type "b". The next variable calculates reciprocity for present events of type "b" and past events of type "a". Additionally, a variable is calculated, where present events as well as past events are of type "a" and a fourth variable calculates reciprocity for events with type "b" (i.e. valuematch on value "b"). `eventtypevalue = c(.., ..)`, similar to the "`nodmix`"-option, all different combinations of the values specified in `eventtypevalue` are used to create reciprocity statistics.
`eventfiltervar`	An optional numeric/character/or factor variable for each event. If `eventfiltervar` is specified, `eventfiltervalue` has to be provided as well.
`eventfiltervalue`	An optional character string that represents the value for which past events should be filtered. To filter the current events, use `eventtypevar`.

`eventvar`	An optional dummy variable with 0 values for null-events and 1 values for true events. If the `data` is in the form of counting process data, use the `eventvar`-option to specify which variable contains the 0/1-dummy for event occurrence. If this variable is not specified, all events in the past will be considered for the calulation of the reciprocity statistic, regardless if they occurred or not (= are null-events).
`variablename`	An optional value (or values) with the name the reciprocity statistic variable should be given. To be used if `returnData = TRUE` or multiple reciprocity statistics are calculated.
`returnData`	`TRUE/FALSE`. Set to `FALSE` by default. The new variable(s) are bound directly to the `data.frame` provided and the data frame is returned in full.
`showprogressbar`	`TRUE/FALSE`. Can only be set to TRUE if the function is not run in parallel.
`inParallel`	`TRUE/FALSE`. An optional boolean to specify if the loop should be run in parallel.
`cluster`	An optional numeric or character value that defines the cluster. By specifying a single number, the cluster option uses the provided number of nodes to parallellize. By specifying a cluster using the `makeCluster`-command in the `doParallel`-package, the loop can be run on multiple nodes/cores. E.g., `cluster = makeCluster(12, type="FORK")`.

Details

The `reciprocityStat()`-function calculates an endogenous statistic that measures whether senders have a tendency to reciprocate events.

The effect is calculated as follows:

$$G_t = G_t(E) = (A, B, w_t),$$

G_t represents the network of past events and includes all events E. These events consist each of a sender $a \in A$ and a target $b \in B$ and a weight function w_t:

$$w_t(i,j) = \sum_{e:a=i,b=j} |w_e| \cdot e^{-(t-t_e)\cdot \frac{ln(2)}{T_{1/2}}} \cdot \frac{ln(2)}{T_{1/2}},$$

where w_e is the event weight (usually a constant set to 1 for each event), t is the current event time, t_e is the past event time and $T_{1/2}$ is a halflife parameter.

For the reciprocity effect, the past events G_t are filtered to include only events where the senders are the present targets and the targets are the present senders:

$$reciprocity(G_t, a, b) = w_t(b, a)$$

An exponential decay function is used to model the effect of time on the endogenous statistics. Each past event that involves the sender as target and the target as sender, and fulfills additional filtering options specified via event type or event attributes, is weighted with an exponential decay. The further apart the past event is from the present event, the less weight is given to this event. The halflife parameter in the `reciprocityStat()`-function determines at which rate the weights of past events should be reduced.

The `eventtypevar`- and `eventattributevar`-options help filter the past events more specifically. How they are filtered depends on the `eventtypevalue`- and `eventattributevalue`-option.

Author(s)

Laurence Brandenberger <`laurence.brandenberger@eawag.ch`>

See Also

rem-package

Examples

```
# create some data with 'sender', 'target' and a 'time'-variable
# (Note: Data used here are random events from the Correlates of War Project)
sender <- c('TUN', 'NIR', 'NIR', 'TUR', 'TUR', 'USA', 'URU',
            'IRQ', 'MOR', 'BEL', 'EEC', 'USA', 'IRN', 'IRN',
            'USA', 'AFG', 'ETH', 'USA', 'SAU', 'IRN', 'IRN',
            'ROM', 'USA', 'USA', 'PAN', 'USA', 'USA', 'YEM',
            'SYR', 'AFG', 'NAT', 'NAT', 'USA')
target <- c('BNG', 'ZAM', 'JAM', 'SAU', 'MOM', 'CHN', 'IRQ',
            'AFG', 'AFG', 'EEC', 'BEL', 'ITA', 'RUS', 'UNK',
            'IRN', 'RUS', 'AFG', 'ISR', 'ARB', 'USA', 'USA',
            'USA', 'AFG', 'IRN', 'IRN', 'IRN', 'AFG', 'PAL',
            'ARB', 'USA', 'EEC', 'BEL', 'PAK')
time <- c('800107', '800107', '800107', '800109', '800109',
          '800109', '800111', '800111', '800111', '800113',
          '800113', '800113', '800114', '800114', '800114',
          '800116', '800116', '800116', '800119', '800119',
          '800119', '800122', '800122', '800122', '800124',
          '800125', '800125', '800127', '800127', '800127',
          '800204', '800204', '800204')
type <- sample(c('cooperation', 'conflict'), 33,
               replace = TRUE)
important <- sample(c('important', 'not important'), 33,
                    replace = TRUE)

# combine them into a data.frame
dt <- data.frame(sender, target, time, type, important)

# create event sequence and order the data
dt <- eventSequence(datevar = dt$time, dateformat = "%y%m%d",
                    data = dt, type = "continuous",
                    byTime = "daily", returnData = TRUE,
                    sortData = TRUE)

# create counting process data set (with null-events) - conditional logit setting
dts <- createRemDataset(dt, dt$sender, dt$target, dt$event.seq.cont,
                          eventAttribute = dt$type,
                          atEventTimesOnly = TRUE, untilEventOccurrs = TRUE,
  returnInputData = TRUE)
## divide up the results: counting process data = 1, original data = 2
dtrem <- dts[[1]]
dt <- dts[[2]]
## merge all necessary event attribute variables back in
dtrem$type <- dt$type[match(dtrem$eventID, dt$eventID)]
dtrem$important <- dt$important[match(dtrem$eventID, dt$eventID)]
# manually sort the data set
dtrem <- dtrem[order(dtrem$eventTime), ]

# calculate reciprocity statistic
dtrem$recip <- reciprocityStat(data = dtrem,
                            time = dtrem$eventTime,
                            sender = dtrem$sender,
                            target = dtrem$target,
                            eventvar = dtrem$eventDummy,
                            halflife = 2)

# plot sender-outdegree over time
library("ggplot2")
ggplot(dtrem, aes(eventTime, recip,
               group = factor(eventDummy), color = factor(eventDummy)) ) +
  geom_point()+ geom_smooth()

# calculate reciprocity statistic with typematch
# if a cooperated with b in the past, does
# b cooperate with a now?
dtrem$recip.typematch <- reciprocityStat(data = dtrem,
                               time = dtrem$eventTime,
                               sender = dtrem$sender,
                               target = dtrem$target,
                               eventvar = dtrem$eventDummy,
                               eventtypevar = dtrem$type,
                               eventtypevalue = 'valuematch',
                               halflife = 2)

# calculate reciprocity with valuemix on type
dtrem <- reciprocityStat(data = dtrem,
                         time = dtrem$eventTime,
                         sender = dtrem$sender,
                         target = dtrem$target,
                         eventvar = dtrem$eventDummy,
                         eventtypevar = dtrem$type,
                         eventtypevalue = 'valuemix',
                         halflife = 2,
                         returnData = TRUE)

# calculate reciprocity and count important events only
dtrem$recip.filtered <- reciprocityStat(data = dtrem,
                               time = dtrem$eventTime,
                               sender = dtrem$sender,
                               target = dtrem$target,
                               eventvar = dtrem$eventDummy,
                               eventfiltervar = dtrem$important,
                               eventfiltervalue = 'important',
                               halflife = 2)
```

`similarityStat` *Calculate similarity statistics*

Description

Calculate the endogenous network statistic `similarity` for relational event models. `similarityStat` measures the tendency for senders to adapt their behavior to that of their peers.

Usage

```
similarityStat(data, time, sender, target,
    senderOrTarget = 'sender',
    whichSimilarity = NULL,
    halflifeLastEvent = NULL,
    halflifeTimeBetweenEvents = NULL,
    eventtypevar = NULL,
    eventfiltervar = NULL,
    eventfiltervalue = NULL,
    eventvar = NULL,
    variablename = 'similarity',
    returnData = FALSE,
    dataPastEvents = NULL,
    showprogressbar = FALSE,
    inParallel = FALSE, cluster = NULL
)
```

Arguments

`data`	A data frame containing all the variables.
`time`	Numeric variable that represents the event sequence. The variable has to be sorted in ascending order.
`sender`	A string (or factor or numeric) variable that represents the sender of the event.
`target`	A string (or factor or numeric) variable that represents the target of the event.
`senderOrTarget`	`sender` or `target`. Indicates on which variable (sender or target) the similarity should be calculated on. Sender similarity measures how many targets the current sender has in common with other senders who used the same targets in the past. Target similarity measures how many senders have used the current target as well as another target that the current sender used in the past.
`whichSimilarity`	`"total"` or `"average"`. Indicates how the variable should be aggregated. `"total"` counts the number of similar events there are in the past event history. `"average"` divides the count of similar events by the number of senders or the number of targets, depending on which mode of similarity is chosen.

`halflifeLastEvent`	A numeric value that is used in the decay function. The vector of past events is weighted by an exponential decay function using the specified halflife. The halflife parameter determines after how long a period the event weight should be halved. For sender similarity: The halflife determines the weight of the count of targets that two actors have in common. The further back the second sender was active, the less weight is given the similarity between this sender and the current sender. For target similarity: The halflife determines the weight of the count of targets that have used both been used by other senders in the past. The longer ago the current sender engaged in an event with the other target, the less weight is given the count.
`halflifeTimeBetweenEvents`	A numeric value that is used in the decay function. Instead of counting each past event for the similarity statistic, each event is reduced depending on the time that passed between the current event and the past event. For sender similarity: Each target that two actors have in common is weighted by the time that passed between the two events. For target similarity: Each sender that two targets have in common is weighted by the time that passed between the two events.
`eventtypevar`	An optional dummy variable that represents the type of the event. If specified, only past events are considered for the count that reflect the same type as the current event (typematch).
`eventfiltervar`	An optional variable that filters past events by the `eventfiltervalue` specified.
`eventfiltervalue`	A string that represents an event attribute by which all past events have to be filtered by.
`eventvar`	An optional dummy variable with 0 values for null-events and 1 values for true events. If the `data` is in the form of counting process data, use the `eventvar`-option to specify which variable contains the 0/1-dummy for event occurrence. If this variable is not specified, all events in the past will be considered for the calulation of the similarity statistic, regardless if they occurred or not (= are null-events). Misspecification could result in grievous errors in the calculation of the network statistic.
`variablename`	An optional value (or values) with the name the similarity statistic variable should be given. To be used if `returnData = TRUE`.
`returnData`	`TRUE`/`FALSE`. Set to `FALSE` by default. The new variable(s) are bound directly to the `data.frame` provided and the data frame is returned in full.
`dataPastEvents`	An optional `data.frame` with the following variables: column 1 = time variable, column 2 = sender variable, column 3 = target on other variable (or all "1"), column 4 = event type variable (or all "1"), column 5 = event filter variable (or all "1"). Make sure that the data frame does not contain null events. Filter it out for true events only.
`showprogressbar`	`TRUE`/`FALSE`. To be implemented.
`inParallel`	`TRUE`/`FALSE`. An optional boolean to specify if the loop should be run in parallel.
`cluster`	An optional numeric or character value that defines the cluster. By specifying a single number, the cluster option uses the provided number of nodes

to parallellize. By specifying a cluster using the `makeCluster`-command in the `doParallel`-package, the loop can be run on multiple nodes/cores. E.g., `cluster = makeCluster(12, type="FORK")`.

Details

The `similiarityStat()`-function calculates an endogenous statistic that measures whether sender (or targets) have a tendency to cluster together. Tow distinct types of similarity measures can be calculated: sender similarity or target similarity.

Sender similarity: How many targets does the current sender have in common with senders who used the current target in the past? How likely is it that two senders are alike?

The function proceeds as follows:

1. First it filters out all the targets that the present sender a used in the past
2. Next it filters out all the senders that have also used the current target b
3. For each of the senders found in (2) it compiles a list of targets that this sender has used in the past
4. For each of the senders found in (2) it cross-checks the two lists generated in (1) and (3) and count how many targets the two senders have in common.

Target similarity: How many senders have used the same two concepts that the current sender has used (in the past and is currently using)? For each target that the current sender has used in the past, how many senders have also used these past targets as well as the current target? How likely is it that two targets are used together?

The function proceeds as follows:

1. First filter out all the targets that the current sender a has used in the past
2. Next it filters out all the senders that have also used the current target b
3. For each target found in (1) it compiles a list of senders that have also used this target in the past
4. For each target found in (1) it cross-checks the list of senders that have used b (found under (2)) and the list of senders that also used one other target that a used (found under (3))

Two decay functions may be used in the calculation of the similarity score for each event.

Author(s)

Laurence Brandenberger <`laurence.brandenberger@eawag.ch`>

See Also

rem-package

Examples

```
# create some data with 'sender', 'target' and a 'time'-variable
# (Note: Data used here are random events from the Correlates of War Project)
sender <- c('TUN', 'NIR', 'NIR', 'TUR', 'TUR', 'USA', 'URU',
            'IRQ', 'MOR', 'BEL', 'EEC', 'USA', 'IRN', 'IRN',
            'USA', 'AFG', 'ETH', 'USA', 'SAU', 'IRN', 'IRN',
            'ROM', 'USA', 'USA', 'PAN', 'USA', 'USA', 'YEM',
            'SYR', 'AFG', 'NAT', 'NAT', 'USA')
target <- c('BNG', 'ZAM', 'JAM', 'SAU', 'MOM', 'CHN', 'IRQ',
            'AFG', 'AFG', 'EEC', 'BEL', 'ITA', 'RUS', 'UNK',
            'IRN', 'RUS', 'AFG', 'ISR', 'ARB', 'USA', 'USA',
            'USA', 'AFG', 'IRN', 'IRN', 'IRN', 'AFG', 'PAL',
            'ARB', 'USA', 'EEC', 'BEL', 'PAK')
time <- c('800107', '800107', '800107', '800109', '800109',
          '800109', '800111', '800111', '800111', '800113',
          '800113', '800113', '800114', '800114', '800114',
          '800116', '800116', '800116', '800119', '800119',
          '800119', '800122', '800122', '800122', '800124',
          '800125', '800125', '800127', '800127', '800127',
          '800204', '800204', '800204')
type <- sample(c('cooperation', 'conflict'), 33,
               replace = TRUE)
important <- sample(c('important', 'not important'), 33,
                    replace = TRUE)

# combine them into a data.frame
dt <- data.frame(sender, target, time, type, important)

# create event sequence and order the data
dt <- eventSequence(datevar = dt$time, dateformat = "%y%m%d",
                    data = dt, type = "continuous",
                    byTime = "daily", returnData = TRUE,
                    sortData = TRUE)

# create counting process data set (with null-events) - conditional logit setting
dts <- createRemDataset(dt, dt$sender, dt$target, dt$event.seq.cont,
                          eventAttribute = dt$type,
                          atEventTimesOnly = TRUE, untilEventOccurrs = TRUE,
  returnInputData = TRUE)
## divide up the results: counting process data = 1, original data = 2
dtrem <- dts[[1]]
dt <- dts[[2]]
## merge all necessary event attribute variables back in
dtrem$type <- dt$type[match(dtrem$eventID, dt$eventID)]
dtrem$important <- dt$important[match(dtrem$eventID, dt$eventID)]
# manually sort the data set
dtrem <- dtrem[order(dtrem$eventTime), ]

# average sender similarity
dtrem$s.sim.av <- similarityStat(data = dtrem,
                              time = dtrem$eventTime,
                              sender = dtrem$sender,
                              target = dtrem$target,
                              eventvar = dtrem$eventDummy,
                              senderOrTarget = "sender",
                              whichSimilarity = "average")

# average target similarity
dtrem$t.sim.av <- similarityStat(data = dtrem,
                                 time = dtrem$eventTime,
                                 sender = dtrem$sender,
                                 target = dtrem$target,
                                 eventvar = dtrem$eventDummy,
                                 senderOrTarget = "target",
                                 whichSimilarity = "average")

# Calculate sender similarity with 1 halflife
# parameter: This parameter makes sure, that those other
# senders (with whom you compare your targets) have been
# active in the past. THe longer they've done nothing, the
# less weight is given to the number of similar targets.
dtrem$s.sim.hl2 <- similarityStat(data = dtrem,
                                 time = dtrem$eventTime,
                                 sender = dtrem$sender,
                                 target = dtrem$target,
                                 eventvar = dtrem$eventDummy,
                                 senderOrTarget = "sender",
                                 halflifeLastEvent = 2)

# Calculate sender similarity with 2 halflife parameters:
# The first parameter makes sure that the actors against
# whom you compare yourself have been active in the
# recent past. The second halflife parameter makes
# sure that the two events containing the same
# targets (once by the current actor, once by the other
# actor) are not that far apart. The longer apart, the
# less likely it is that the current sender will remember
# how the similar-past sender has acted.
dtrem$s.sim.hl2.hl1 <- similarityStat(data = dtrem,
                                  time = dtrem$eventTime,
                                  sender = dtrem$sender,
                                  target = dtrem$target,
                                  eventvar = dtrem$eventDummy,
                                  senderOrTarget = "sender",
                                  halflifeLastEvent = 2,
                                  halflifeTimeBetweenEvents = 1)
```

timeToEvent *Calculate the time-to-next-event or the time-since-date for a REM data set.*

Description

Calculate time-to-next-event or time-since-date for a REM data set.

Usage

```
timeToEvent(time, type = 'time-to-next-event', timeEventPossible = NULL)
```

Arguments

`time`	A integer or Date variable reflecting the time of the event. Note: make sure to specify event time not the event sequence in a counting process data set.
`type`	Either 'time-to-next-event' or 'time-since-date'. `type = 'time-to-next-event'` calculates the time between the current event and the event closes to the current in the past. `type = 'time-since-date'` uses the `time`-variable as well as the `timeEventPossible`-variable to calculate how much time has passed between the two variables, i.e., how long the event took to come true.
`timeEventPossible`	An optional integer or Date variable to be used if `type = 'time-since-date'` is specified.

Details

To come.

Author(s)

Laurence Brandenberger <`laurence.brandenberger@eawag.ch`>

See Also

rem-package

Examples

```
## get some random data
dt <- data.frame(
  sender = c('a', 'c', 'd', 'a', 'a', 'f', 'c'),
  target = c('b', 'd', 'd', 'b', 'b', 'a', 'd'),
  date = c(rep('10.01.90',2), '11.01.90', '04.01.90',
 '05.01.90', rep('10.01.90',2)),
  start = c(0, 0, 1, 1, 1, 3, 3),
  end = rep(6, 7),
  targetAvailableSince = c(rep(-10,6), -2),
  dateTargetAvailable = c(rep('31.12.89',6), '01.01.90')
)

## create event sequence
dt <- eventSequence(dt$date, dateformat = '%d.%m.%y', data = dt,
              type = "continuous", byTime = "daily",
              excludeDate = '07.01.90',
              returnData = TRUE, sortData = TRUE,
              returnDateSequenceData = FALSE)
## also return the sequenceData
dt.seq <- eventSequence(dt$date, dateformat = '%d.%m.%y', data = dt,
                    type = "continuous", byTime = "daily",
                    excludeDate = '07.01.90',
                    returnDateSequenceData = TRUE)

## create counting process data set
dts <- createRemDataset(
data = dt, sender = dt$sender, target = dt$target,
eventSequence = dt$event.seq.cont,
eventAttribute = NULL, time = NULL,
start = dt$start, startDate = NULL,
end = dt$end, endDate = NULL,
timeformat = NULL, atEventTimesOnly = TRUE,
untilEventOccurrs = TRUE,
includeAllPossibleEvents = FALSE,
possibleEvents = NULL, returnInputData = TRUE)
## divide up the results: counting process data = 1, original data = 2
dt.rem <- dts[[1]]
dt <- dts[[2]]

## merge all necessary event attribute variables back in
dt.rem$targetAvailableSince <- dt$targetAvailableSince[match(dt.rem$eventID,
dt$eventID)]
dt.rem$dateTargetAvailable <- dt$dateTargetAvailable[match(dt.rem$eventID,
dt$eventID)]

## add dates to the eventTime
dt.rem$eventDate <- dt.seq$date.sequence[match(dt.rem$eventTime,
dt.seq$event.sequence)]

## sort the dataframe according to eventTime
dt.rem <- dt.rem[order(dt.rem$eventTime), ]

## 1. numeric, time-to-next-event
dt.rem$timeToNextEvent <- timeToEvent(as.integer(dt.rem$eventTime))

## 2. numeric, time-since
dt.rem$timeSince <- timeToEvent(dt.rem$eventTime,
                                type = 'time-since-date',
                                dt.rem$targetAvailableSince)

## 3. Date, time-to-next-event
# since the event sequence excluded 06.01.90 => time to next event differs
# for the two specification with the integr (1) and the Date-variable (2).
# To be consistent, pick the eventTime instead of the Date-variable.
dt.rem$timeToNextEvent2 <- timeToEvent(as.Date(dt.rem$eventDate, '%d.%m.%y'))


## 4. Date, time-since
dt.rem$timeSince2 <- timeToEvent(
as.Date(dt.rem$eventDate, '%d.%m.%y'),
type = 'time-since-date',
as.Date(dt.rem$dateTargetAvailable, '%d.%m.%y'))
```

`triadStat` *Calculate triad statistics*

Description

Calculate the endogenous network statistic `triads` that measures the tendency for events to close open triads.

Usage

```
triadStat(data, time, sender, target, halflife,
    weight = NULL,
    eventtypevar = NULL,
    eventtypevalues = NULL,
    eventfiltervar = NULL,
    eventfilterAI = NULL,
    eventfilterBI = NULL,
    eventfilterAB = NULL,
    eventvar = NULL,
    variablename = 'triad',
    returnData = FALSE,
    showprogressbar = FALSE,
    inParallel = FALSE, cluster = NULL
)
```

Arguments

`data`	A data frame containing all the variables.
`time`	Numeric variable that represents the event sequence. The variable has to be sorted in ascending order.
`sender`	A string (or factor or numeric) variable that represents the sender of the event.
`target`	A string (or factor or numeric) variable that represents the target of the event.
`halflife`	A numeric value that is used in the decay function. The vector of past events is weighted by an exponential decay function using the specified halflife. The halflife parameter determins after how long a period the event weight should be halved. E.g. if `halflife = 5`, the weight of an event that occured 5 units in the past is halved. Smaller halflife values give more importance to more recent events, while larger halflife values should be used if time does not affect the sequence of events that much.
`weight`	An optional numeric variable that represents the weight of each event. If `weight = NULL` each event is given an event weight of `1`.
`eventtypevar`	An optional dummy variable that represents the type of the event. Use `eventtypevalues` to specify how the `eventtypevar` should be used to filter past events. Specifying the `eventtypevar` is needed to calculate effects of social balance theory, such as 'friend-of-friend' or 'enemy-of-enemy' statistics.

`eventtypevalues`
: Two string values that represent the type of the past events. The first string value represents the eventtype that exists for all past events that include the current sender (either as sender or target) and a third actor. The second value represents the eventtype for all past events that include the target (either as sender or target) as well as the third actor. An example: Let the `eventtypevar` indicate whether an event is of cooperative or hostile nature. To test whether the hypothesis 'the friend of my friend is my friend' holds, both `eventtypevalues` must be the same and point to the cooperative type (e.g. `eventtypevalues = c("cooperation", "cooperation")`) depending on how the `eventtypevar` is coded. To test whether the hypothesis 'the friend of my enemy is my enemy' holds, the first value in `eventtypevalues` represents the hostile event between current sender and a third actor and the second value represents the cooperative event between the third actor and the target. To test the hypothesis 'the enemy of my enemy is my friend', the first value represents the hostile events between current sender and a third actor and the second value represents the hostile event between the current target and the third actor. For the fourth hypothesis, to test social balance theory 'the enemy of my friend is my enemy', the first value represents a cooperative event between the current sender and a third actor and the second value represents a hostile event between the current target and the third actor.

`eventfiltervar`
: An optional string (or factor or numeric) variable that can be used to filter past and current events. Use `eventfilterAI`, `eventfilterBI` or `eventfilterAB` to specify which past events should be filtered and by what value.

`eventfilterAI`
: An optional value used to specify how paste events should be filtered depending on their attribute. Each distinct edge that form a triad can be filtered. `eventfilterAI` refers to the past event involving the current sender (a) and a third actor (i). `eventfilterBI`referrs to past events involving target (b) and the third actor (i). `eventfilterAB` refers to the current event involving sender (a) and target (b).

`eventfilterBI`
: see `eventfilterAI`.

`eventfilterAB`
: see `eventfilterAI`.

`eventvar`
: An optional dummy variable with 0 values for null-events and 1 values for true events. If the `data` is in the form of counting process data, use the `eventvar`-option to specify which variable contains the 0/1-dummy for event occurrence. If this variable is not specified, all events in the past will be considered for the calulation of the triad statistic, regardless if they occurred or not (= are null-events).

`variablename`
: An optional value (or values) with the name the triad statistic variable should be given. To be used if `returnData = TRUE`.

`returnData`
: `TRUE/FALSE`. Set to `FALSE` by default. The new variable is bound directly to the `data.frame` provided and the data frame is returned in full.

`showprogressbar`
: `TRUE/FALSE`. Can only be set to TRUE if the function is not run in parallel.

`inParallel`
: `TRUE/FALSE`. An optional boolean to specify if the loop should be run in parallel.

`cluster`
: An optional numeric or character value that defines the cluster. By specifying a single number, the cluster option uses the provided number of nodes to parallellize. By specifying a cluster using the `makeCluster`-command in

the `doParallel`-package, the loop can be run on multiple nodes/cores. E.g., `cluster = makeCluster(12, type="FORK")`.

Details

The `triadStat()`-function calculates an endogenous statistic that measures whether events have a tendency to form closing triads.

The effect is calculated as follows:

$$G_t = G_t(E) = (A, B, w_t),$$

G_t represents the network of past events and includes all events E. These events consist each of a sender $a \in A$ and a target $b \in B$ and a weight function w_t:

$$w_t(i,j) = \sum_{e:a=i,b=j} |w_e| \cdot e^{-(t-t_e)\cdot \frac{ln(2)}{T_{1/2}}} \cdot \frac{ln(2)}{T_{1/2}},$$

where w_e is the event weight (usually a constant set to 1 for each event), t is the current event time, t_e is the past event time and $T_{1/2}$ is a halflife parameter.

For the triad effect, the past events G_t are filtered to include only events where the current event closes an open triad in the past.

$$triad(G_t, a, b) = \sqrt{\sum_{i \in A} w_t(a,i) \cdot w_t(i,b)}$$

An exponential decay function is used to model the effect of time on the endogenous statistics. The further apart the past event is from the present event, the less weight is given to this event. The halflife parameter in the `triadStat()`-function determines at which rate the weights of past events should be reduced. Therefore, if the one (or more) of the two events in the triad have occurred further in the past, less weight is given to this triad because it becomes less likely that the sender and target actors reacted to each other in the way the triad assumes.

The `eventtypevar`- and `eventattributevar`-options help filter the past events more specifically. How they are filtered depends on the `eventtypevalue`- and `eventattributevalue`-option.

Author(s)

Laurence Brandenberger <`laurence.brandenberger@eawag.ch`>

See Also

rem-package

Examples

```
# create some data with 'sender', 'target' and a 'time'-variable
# (Note: Data used here are random events from the Correlates of War Project)
sender <- c('TUN', 'UNK', 'NIR', 'TUR', 'TUR', 'USA', 'URU',
            'IRQ', 'MOR', 'BEL', 'EEC', 'USA', 'IRN', 'IRN',
            'USA', 'AFG', 'ETH', 'USA', 'SAU', 'IRN', 'IRN',
            'ROM', 'USA', 'USA', 'PAN', 'USA', 'USA', 'YEM',
            'SYR', 'AFG', 'NAT', 'UNK', 'IRN')
target <- c('BNG', 'RUS', 'JAM', 'SAU', 'MOM', 'CHN', 'IRQ',
            'AFG', 'AFG', 'EEC', 'BEL', 'ITA', 'RUS', 'UNK',
            'IRN', 'RUS', 'AFG', 'ISR', 'ARB', 'USA', 'USA',
            'USA', 'AFG', 'IRN', 'IRN', 'IRN', 'AFG', 'PAL',
            'ARB', 'USA', 'EEC', 'IRN', 'CHN')
time <- c('800107', '800107', '800107', '800109', '800109',
          '800109', '800111', '800111', '800111', '800113',
          '800113', '800113', '800114', '800114', '800114',
          '800116', '800116', '800116', '800119', '800119',
          '800119', '800122', '800122', '800122', '800124',
          '800125', '800125', '800127', '800127', '800127',
          '800204', '800204', '800204')
type <- sample(c('cooperation', 'conflict'), 33,
               replace = TRUE)
important <- sample(c('important', 'not important'), 33,
                    replace = TRUE)

# combine them into a data.frame
dt <- data.frame(sender, target, time, type, important)

# create event sequence and order the data
dt <- eventSequence(datevar = dt$time, dateformat = "%y%m%d",
                    data = dt, type = "continuous",
                    byTime = "daily", returnData = TRUE,
                    sortData = TRUE)

# create counting process data set (with null-events) - conditional logit setting
dts <- createRemDataset(dt, dt$sender, dt$target, dt$event.seq.cont,
                          eventAttribute = dt$type,
                          atEventTimesOnly = TRUE, untilEventOccurrs = TRUE,
  returnInputData = TRUE)
dtrem <- dts[[1]]
dt <- dts[[2]]
# manually sort the data set
dtrem <- dtrem[order(dtrem$eventTime), ]
# merge type-variable back in
dtrem$type <- dt$type[match(dtrem$eventID, dt$eventID)]

# calculate triad statistic
dtrem$triad <- triadStat(data = dtrem, time = dtrem$eventTime,
                      sender = dtrem$sender, target = dtrem$target,
                      eventvar = dtrem$eventDummy,
                      halflife = 2)
```

```
# calculate friend-of-friend statistic
dtrem$triad.fof <- triadStat(data = dtrem, time = dtrem$eventTime,
                          sender = dtrem$sender, target = dtrem$target,
                          halflife = 2, eventtypevar = dtrem$type,
                          eventtypevalues = c("cooperation",
                                              "cooperation"),
                          eventvar = dtrem$eventDummy)

# calculate friend-of-enemy statistic
dtrem$triad.foe <- triadStat(data = dtrem, time = dtrem$eventTime,
                             sender = dtrem$sender, target = dtrem$target,
                             halflife = 2, eventtypevar = dtrem$type,
                             eventtypevalues = c("conflict",
                                                 "cooperation"),
                             eventvar = dtrem$eventDummy)

# calculate enemy-of-friend statistic
dtrem$triad.eof <- triadStat(data = dtrem, time = dtrem$eventTime,
                             sender = dtrem$sender, target = dtrem$target,
                             halflife = 2, eventtypevar = dtrem$type,
                             eventtypevalues = c("cooperation",
                                                 "conflict"),
                             eventvar = dtrem$eventDummy)

# calculate enemy-of-enemy statistic
dtrem$triad.eoe <- triadStat(data = dtrem, time = dtrem$eventTime,
                             sender = dtrem$sender, target = dtrem$target,
                             halflife = 2, eventtypevar = dtrem$type,
                             eventtypevalues = c("conflict",
                                                 "conflict"),
                             eventvar = dtrem$eventDummy)
```

Index