Package: dream (via r-universe)

October 12, 2026

Title Dynamic Relational Event Analysis and Modeling

Version 2.1.5

Maintainer Kevin A. Carson <kacarson@arizona.edu>

Description A set of tools for relational and event analysis, including two- and one-mode network brokerage and structural measures, and helper functions optimized for relational event analysis with large datasets, including creating relational risk sets, computing network statistics, estimating relational event models, and simulating relational event sequences. For more information on relational event models, see Butts (2008) <doi:10.1111/j.1467-9531.2008.00203.x>, Lerner and Lomi (2020) <doi:10.1017/nws.2019.57>, Bianchi et al. (2024) <doi:10.1146/annurev-statistics-040722-060248>, and Butts et al. (2023) <doi:10.1017/nws.2023.9>. In terms of the structural measures in this package, see Leal (2025) <doi:10.1177/00491241251322517>, Burchard and Cornwell (2018) <doi:10.1016/j.socnet.2018.04.001>, and Fujimoto et al. (2018) <doi:10.1017/nws.2018.11>. This package was developed with support from the National Science Foundation's (NSF) Human Networks and Data Science Program (HNDS) under award number 2241536 (PI: Diego F. Leal). Any opinions, findings, and conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.3

Depends R (>= 3.5.0)

Imports collapse, data.table, foreach, parallel, doParallel, Rcpp, lifecycle, methods

LinkingTo Rcpp, RcppArmadillo

URL https://github.com/kevinCarson/dream

BugReports https://github.com/kevinCarson/dream/issues

NeedsCompilation yes

Author Kevin A. Carson [aut, cre] (ORCID:
<https://orcid.org/0009-0001-5762-2586>), Diego F. Leal [aut]
(ORCID: <https://orcid.org/0000-0001-9973-4626>)

Repository https://cran.r-universe.dev

Date/Publication 2026-07-18 22:40:07 UTC

RemoteUrl https://github.com/cran/dream

RemoteRef HEAD

RemoteSha 2820439c85f42fa372d9fd87d410cafaf968ff80

Contents

as.data.frame.dream_sequence . . . 3
coef.dream_rem . . . 5
create_res . . . 6
dream_information . . . 13
dream_sequence . . . 14
dreamstats_actor . . . 17
dreamstats_actorfe . . . 19
dreamstats_degree . . . 21
dreamstats_dyadcut . . . 25
dreamstats_dyadfe . . . 28
dreamstats_dyadic . . . 29
dreamstats_event . . . 32
dreamstats_fourcycles . . . 34
dreamstats_persistence . . . 37
dreamstats_prefattachment . . . 40
dreamstats_recency . . . 42
dreamstats_reciprocity . . . 46
dreamstats_repetition . . . 50
dreamstats_triads . . . 53
estimate_rem . . . 58
gof_rem . . . 64
logLik.dream_rem . . . 66
netstats_om_constraint . . . 67
netstats_om_effective . . . 69
netstats_om_nwalks . . . 71
netstats_om_pib . . . 72
netstats_tm_constraint . . . 74
netstats_tm_degreecent . . . 76
netstats_tm_density . . . 78
netstats_tm_effective . . . 79
netstats_tm_egodistance . . . 81
netstats_tm_homfourcycles . . . 83

netstats_tm_redundancy 84
plot.dream_gof 86
plot.dream_rem 87
predict.dream_rem 89
print.dream_gof 91
print.dream_rem 91
print.dream_sequence 92
print.summary.dream_gof 92
print.summary.dream_rem 93
print.summary.dream_sequence 93
residuals.dream_rem 94
simulate_rem_seq 95
simulated.res 100
southern.women 100
summary.dream_gof 101
summary.dream_rem 102
summary.dream_sequence 102
vcov.dream_rem 103
WikiEvent2018.first100k 104

Index **105**

`as.data.frame.dream_sequence`
Coerce a `dream_sequence` *Object into a* `data.frame` *Object*

Description

This function will create a `data.frame` object from a `dream_sequence` object, where the generated data.frame includes the processed event sequence and the computed statistics. If `all.events` is set to FALSE, then the created object will only contains the sampled events, whereas, if it is TRUE, the returned object will contain all events, and if any computed statistics are attached, the non-sampled event entries will be NAs.

Usage

```
## S3 method for class 'dream_sequence'
as.data.frame(x, row.names = NULL, optional = FALSE, all.events = FALSE, ...)
```

Arguments

`x`	An object of class `dream_sequence`.
`row.names`	The `row.names` argument from the `as.data.frame` function. Please see that function for more details on this argument.
`optional`	The `optional` argument from the `as.data.frame` function. Please see that function for more details on this argument.

`all.events` TRUE/FALSE. If sampling from the observed event sequence has occurred, TRUE returns all of the sampled and non-sampled observed events, whereas FALSE returns only those sampled observed events. FALSE by default.

`...` Additional arguments for other methods..

Value

Returns a `data.frame` object that contains the processed relational event statistics and the associated sufficient network statistics for the event sequence.

Examples

```
#a pseudo event sequence
events <- data.frame(time = 1:18, eventID = 1:18,
                     sender = c("A", "B", "C",
                                "A", "D", "E",
                                "F", "B", "A",
                                "F", "D", "B",
                               "G", "B", "D",
                               "H", "A", "D"),
                   target = c("B", "C", "D",
                               "E", "A", "F",
                               "D", "A", "C",
                                "G", "B", "C",
                                "H", "J", "A",
                                "F", "C", "B"))
#making a post-processing event sequence
processed <- create_res(type = "one-mode",
                        ordinal = TRUE,
                        riskset = "fixed",
                        case_control=TRUE,
                        time = events$time,
                        sender = events$sender,
                        receiver = events$target,
                        p_samplingobserved = 1.00,
                        n_controls = 1,
                        seed = 9999)

#computing the sender indegree statistics
processed <- dreamstats_degree(formation = "sender-indegree", data = processed)
 #computing the outgoing two paths statistics
processed <- dreamstats_triads(formation = "OTP", data = processed)
#computing the repetition/inertia statistics
processed <- dreamstats_repetition(data = processed)
#making the dream_sequence object a data.frame object
new.data <- as.data.frame(processed) #making the event set a data.frame object
new.data #the processed event sequence returned with the computed statistics
```

coef.dream_rem *Extract the ML parameter estimates from Relational Event Model Fits*

Description

This function extracts the Maximum Likelihood (ML) parameter estimates from estimated relational event model fits.

Usage

```
## S3 method for class 'dream_rem'
coef(object, ...)
```

Arguments

object	An object of class "dream_rem".
...	Additional arguments for other methods.

Examples

```
#Creating a psuedo one-mode relational event sequence with ordinal timing
relational.seq <- simulate_rem_seq(n_actors = 8,
                                   n_events = 50,
                                   inertia = TRUE,
                                   inertia_p = 0.10,
                                   sender_outdegree = TRUE,
                                   sender_outdegree_p = 0.05)

#Creating a post-processing event sequence for the above relational sequence
post.processing <-  create_res(type = "one-mode",
                               ordinal = TRUE,
                               riskset = "fixed",
                               time = relational.seq$eventID,
                               sender = as.character(relational.seq$sender),
                               receiver = as.character(relational.seq$target),
                               case_control=TRUE,
                               n_controls = 5)

#Computing the sender-outdegree statistic for the above post-processing
#one-mode relational event sequence
post.processing <- dreamstats_degree(formation = "sender-outdegree",
                                   data = post.processing,
                                   halflife = 2)

#Computing the inertia/repetition statistic for the above post-processing
#one-mode relational event sequence
post.processing <- dreamstats_repetition(data = post.processing,
                                         halflife = 2)
```

```
#Fitting an ordinal timing relational event model to the above one-mode relational
#event sequence
rem <- estimate_rem(~ sender.outdegree + repetition,
                        data=post.processing)
coef(rem) #summary of the relational event model
```

`create_res` *Process One- and Two-Mode Relational Event Sequences and Create Post-Processing Relational Event Sequences*

Description

[Stable]

This function creates one- and two-mode post-processing (sampled) relational event sequences with options for case-control sampling (Vu et al. 2015; Butts 2008), sampling from the observed event sequence (Lerner and Lomi 2020), dynamic time-varying supports sets, actor-varying supports sets, and the full Cartesian product for one-mode sequences and the full cross-product for two-mode event sequences (Butts 2008). The created post-processing relational event sequences are designed to be modeled by relational event models (Butts 2008). Case-control sampling samples an arbitrary m number of controls from the risk set for any event (Vu et al. 2015; Butts 2008). Lerner and Lomi (2020) proposed sampling from the observed event sequence where observed events are sampled with probability p. Importantly, this function allows users to generate post-processing relational event sequences for *ordinal* and *interval* relational event model likelihoods. Lastly, the post-processing relational event sequence is a `dream_sequence` object that is the required object for this package's functions to compute exogenous and endogenous network statistics, alongside the function to estimate Maximum Likelihood relational event models.

Usage

```
create_res(
  type = c("two-mode", "one-mode"),
  ordinal = TRUE,
  t = NULL,
  time,
  sender,
  receiver,
  riskset = c("fixed", "compositional", "cumulative"),
  p_samplingobserved = 1,
  case_control = FALSE,
  n_controls = NULL,
  active_times = NULL,
  seed = NULL
)
```

Arguments

type	"two-mode" indicates that this is a two-mode event sequence (i.e., observed actors can only be either event senders or event receivers). The option "one-mode" indicates that the observed event sequence is one-mode (i.e., observed actors can be event senders and receivers)
ordinal	TRUE/FALSE. TRUE indicates that observed timing of the events is ordinal (and the ordinal timing likelihood function will be used). FALSE denotes that the observed timing is observed, relative to the start of the event sequence (and the interval timing likelihood function will be used). The interval timing option adds the right-censored events to the post-processed relational event sequence (i.e., the set of (sampled) controls events for the time point t, that marks the end of the relational event sequence.) Please see the references for more information.
t	If ordinal is set to FALSE, the time that marks the end of the relational event sequence, relative to the start of the event sequence. If t is NULL and ordinal is set to FALSE, then the right-censored events are not added to the post-processing event sequence and the `estimate_rem` function will not add the right-censoring term to the log-likelihood. The log-likelihood thus reduces to that used in the MLE interval timing relational event model in the `remstimate` function. The default value is NULL.
time	The vector of event time values from the observed event sequence, where the jth entry is the relative time at which the jth event occurred. The event times should be relative to the onset (start) of the relational event sequence.
sender	The vector of event senders from the observed event sequence, where the jth entry is the event sender for the jth observed/realized event.
receiver	The vector of event receivers from the observed event sequence where the jth entry is the event receiver for the jth observed/realized event.
riskset	The argument should be one of the following strings: "fixed", "compositional", "cumulative". "fixed" will create a post-processing relational event sequence where the set of null events for each (sampled) realized/observed event is the full set of actors that were active at anytime in the event sequence (for one-mode sequences, this is the full Cartesian plot of actors, and for two-mode sequences, this is the full cross-product of the event sender and receiver sets). "cumulative" will create a post-processing relational event sequence where the set of null events for each (sampled) realized/observed event at time t is the full set of actors that have been active up to and including t. "compositional" will create a post-processing relational event sequence where the set of null events for each (sampled) realized/observed event at time t is the full set of actors that are considered relationally active at time t dependent upon the `active_times` argument.
p_samplingobserved	The numerical value for the probability of selection for sampling from the observed event sequence. Set to 1 by default indicating that all observed events from the event sequence will be included in the post-processing event sequence.
case_control	TRUE/FALSE. TRUE indicates that case-control sampling from the riskset will be implemented (i.e., a sample from the risk set will be returned). FALSE indicates that case-control sampling will not occur and that all non-realized events based upon the `riskset` argument will be returned. Set ot FALSE by default.

`n_controls` The numerical value for the number of null event controls for each (sampled) observed event. This argument should be specified when one of the following `riskset` types is provided: "constant_sample", "dynamic_sample", and "actor_varying_sample".

`active_times` This argument is either a `data.frame` object for "one-mode" relational event sequences or a `list` that contains 2 `data.frame` objects when `type` is set to "two-mode". For "one-mode" event sequence types, the `data.frame` object must have three named columns: "actor_id", "time_start", "time_end", and the number of rows (observations) is the number of actor active spells, where an actor spell is defined as the time in which the ith actor becomes relationally active (the ith entry for the "time_start" column) and then becomes relationally inactive (the ith entry for the "time_end" column). The "time_start" and "time_end" vectors should be relative to the start of the relational event sequence (commonly set to start at time 0). Importantly, a single actor (denoted by the "actor_id" column) can have multiple active spell rows, as actors may exit and re-enter the relational event sequence. In comparison, actors who do not leave the relational event sequence and always have the capacity to become relationally active should have one row (entry) where the "time_start" value is set to 0 and the "time_end" value should be set to the time point that marks the end of the relational event sequence (t for interval event sequences and the time of the last observed event for ordinal event sequences). Finally, for "two-mode" event sequences, the `active_times` argument should be a list of 2 `data.frame` objects where the list elements are named "senders" and "receivers", where "senders" is a `data.frame` object that follows the same design as the "one-mode" arguments and defines how event senders become relationally active and inactive. Similarly, the second element should be named "receivers" and is a `data.frame` object that follows the above stated design and defines how event receivers become relationally active and inactive.

`seed` The random number seed for user replication. This argument is set to NULL be default.

Details

This function processes observed events from the set A_t, where each event a_i is defined as:

$$a_i \in A_t = (s_i, r_i, \tau_i, G[A_t; \tau_i])$$

where:

- s_i is the sender of the event.
- r_i is the receiver of the event.
- τ_i represents the time of the event.
- $G[A_t; \tau_i] = \{a_1, a_2, \ldots, a_{t'} \mid t' < \tau_i\}$ is the network of past events, that is, all events that occurred prior to the current event, a_i.

For the post-processing event sequences where the `ordinal` argument is set to FALSE, the last set of processed events, marked with the time point t, represent the right-censoring events. The function generates post-processing relational event sequences across three axises: (1) the inclusion of

sampling from the observed/realized relational event sequence, A_t, (2) one-mode vs. two-mode event types, where the relevant actors can be either senders or receivers (in the case of two-mode) sequences, or where the relevant actors can be both (in the case of one-mode sequences), and (3) how the processed support set for each event should be constructed. The third axis is based upon the `riskset` argument, which is one of the following: "fixed", "compositional", "cumulative". "fixed" will create a post-processing relational event sequence where the set of null events for each (sampled) realized/observed event is the full set of actors that were active at anytime in the event sequence (for one-mode sequences, this is the full Cartesian plot of actors, and for two-mode sequences, this is the full cross-product of the event sender and receiver sets). "cumulative" will create a post-processing relational event sequence where the set of null events for each (sampled) realized/observed event at time t is the full set of actors that have been active up to and including t. "compositional" will create a post-processing relational event sequence where the set of null events for each (sampled) realized/observed event at time t is the full set of actors that are considered relationally active at time t dependent upon the `active_times` argument. In addition, case-control sampling from the riskset can be implemented by setting the `case_control` argument to TRUE and specifying a value for `n_controls`.

Fixed Risk Sets:

Following Butts (2008) and Butts and Marcum (2017), for one-mode event sequences (`type` = "one-mode"), the risk (support) set is defined as all possible events at time t, M_t, as the full Cartesian product of actors active in the relational event sequence and is the defined as the set N. Formally:

$$M_t = \{(s, r) \mid s \in N \times r \in N\}$$

where N is the set of all possible actors in the sequence. In this function, the full risk set is considered fixed (constant) across all time points.

For two-mode event sequences (`type` = "two-mode"), the risk (support) set is defined as all possible events at time t, M_t, as the cross product of two disjoint sets, namely, event senders (i.e., S) and event receivers (i.e., R). Formally:

$$M_t = \{(s, r) \mid s \in S \times r \in R\}$$

where S is the set of potential event senders and R is the set of potential event receivers. In this function, the full risk set is considered fixed across all time points.

Fixed Case-Control Risk Sets:

Following Butts (2008), Vu et al. (2015), and Lerner and Lomi (2020), case-control sampling samples an arbitrary number m of non-events from the above risk/support set definitions M_t. This process generates a new support set, $\tilde{M}_t$, for any relational event a_i contained in A_t. $\tilde{M}_t$, for one-mode relational event sequences, is formally defined as:

$$\tilde{M}_t \subseteq \{(s, r) \mid s \in N \times r \in N\}$$

Cumulative Risk Sets:

For cumulative risk sets, for one-mode event sequences (`type` = "one-mode"), the risk (support) set at time t, that is, M_t, is defined as the full Cartesian product of all past actors who have been involved in a relational event at or before time t. Formally:

$$M_t\{(s, r) \mid s \in N_t \times r \in N_t\}$$

where N_t is the set of potential event senders and targets at and time t. The definition follows the same as above for two-mode event sequences, where the sets are now defined as S_t and R_t.

Cumulative Case-Control Risk Sets:

For cumulative risk sets, for one-mode event sequences (`type = "one-mode"`), the risk (support) set at time t, that is, M_t, is defined as a sample of m dyads from the full Cartesian product of all past actors who have been involved in a relational event at or before time t. Formally:

$$\tilde{M}_t \subseteq \{(s, r) \mid s \in N_t \times r \in N_t\}$$

where N_t is the set of potential event senders and targets at and time t. The definition follows the same as above for two-mode event sequences, where the sets are now defined as S_t and R_t.

Compositional Risk Sets:

Compositional support sets allows for actors to enter, exit, and re-enter the relational event sequence as time progresses. For one-mode event sequences, the node set Y_t is defined as those actors who are considered relationally active at time t. For two-mode event sequences, the node sets S_t and R_t are defined, respectively, as the senders who can be active at time t and the receivers who can be active at time t. For one-mode sequences, the formal definition is:

$$V_t = \{(s, r) \mid s \in Y_t \times r \in Y_t\}$$

Compositional Case-Control Risk Sets:

Based upon the formal definition above, sampling from the full actor-varying risk set generates a new support set definition:

$$\tilde{V}_t \subseteq \{(s, r) \mid s \in Y_t \times r \in Y_t\}$$

where $\tilde{V}_t$ represents the m number of dyads sampled (with equal probability) from the set V_t.

Value

An object of class `dream_sequence` that contains a list of the following elements:

- `processed_sequence` - A `data.table` object that contains the post-processing relational event sequence with the following columns: "time", "eventID", "sender", "receiver", "sampled", and "observed". The "time" column is the vector of event times for the realized and control events. The "eventID" column represents the order that the event occurred in the relational event sequence. The "sender" and "receiver" columns are the specific dyad for that row. The "observed" vector takes a value of 1 if the dyadic pair is the observed dyad at the specific time. The "sampled" vector takes a value of 1 if the dyad was sampled at that event time and 0 if not (relevant for case-control sampling and sampling from the observed event sequence).
- `ordinal` - Based upon the user's input. (`ordinal` and `interval`)
- `t` - Based upon the user's input.
- `riskset` - Based upon the user's input.
- `p` - The probability of sampling from the observed event sequence. Based upon the user's input.
- `case_control` - Based upon the user's input.
- `m` - The number of null event controls for each (sampled) observed event. Based upon the user's input.

- `type` - Based upon the user's input. (`two-mode` and `one-mode`)
- `n` - The number of observed events.
- `sampled_events` - The number of sampled observed events.
- `null` - The number of sampled null (control) events.
- `statistics` - An empty list to store the future computed relational event network statistics.
- `interevent.times` - The vector of interevent times (the time difference between observed events).

Author(s)

Kevin A. Carson kacarson@arizona.edu and Diego F. Leal dflc@arizona.edu

References

Butts, Carter T. 2008. "A Relational Event Framework for Social Action." *Sociological Methodology* 38(1): 155-200.

Butts, Carter T. and Christopher Steven Marcum. 2017. "A Relational Event Approach to Modeling Behavioral Dynamics." In A. Pilny & M. S. Poole (Eds.), *Group processes: Data-driven computational approaches*. Springer International Publishing.

Lerner, Jürgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: How to fit a relational event model to 360 million dyadic events." *Network Science* 8(1): 97–135.

Vu, Duy, Philippa Pattison, and Garry Robins. 2015. "Relational event models for social learning in MOOCs." *Social Networks* 43: 121-135.

Examples

```
# Generating a psuedo one-mode relational event sequence
set.seed(9999)
events <- data.frame(time = sort(rexp(18)),
                                eventID = 1:18,
                                sender = c("A", "B", "C",
                                           "A", "D", "E",
                                           "F", "B", "A",
                                           "F", "D", "B",
                                           "G", "B", "D",
                                           "H", "A", "D"),
                               target = c("B", "C", "D",
                                          "E", "A", "F",
                                          "D", "A", "C",
                                          "G", "B", "C",
                                          "H", "J", "A",
                                          "F", "C", "B"))

# Creating the full one-mode relational risk set with p = 1.00 (all true events)
# based upon the ordinal timing relational event framework
full.process <- create_res(ordinal = TRUE,
                      type = "one-mode",
```

```
                      riskset = "fixed",
                      time = events$time,
                      sender = events$sender,
                      receiver = events$target,
                      p_samplingobserved = 1.00,
                      seed = 9999)

# Creating a fixed one-mode relational risk set with p = 1.00 (all true events)
# and 5 controls based upon the ordinal timing relational event framework
sample.process <- create_res(ordinal = TRUE,
                             type = "one-mode",
                             riskset = "fixed",
                             case_control=TRUE,
                             time = events$time,
                             sender = events$sender,
                             receiver = events$target,
                             p_samplingobserved = 1.00,
                             n_controls = 10,
                             seed = 9999)

# Creating a dynamic one-mode relational risk set with p = 1.00 (all true events)
# based upon the interval timing relational event framework
dynamic.process <- create_res(ordinal = FALSE,
                      t = max(events$time) + rexp(1),
                      type = "one-mode",
                      riskset = "cumulative",
                      time = events$time,
                      sender = events$sender,
                      receiver = events$target,
                      p_samplingobserved = 1.00,
                      seed = 9999)

# Creating a dynamic one-mode relational risk set with p = 1.00 (all true events)
# and 5 controls based upon the interval timing relational event framework
dynamic.sample.process <- create_res(ordinal = FALSE,
                      t = max(events$time) + rexp(1),
                      type = "one-mode",
                      riskset = "cumulative",
                      case_control=TRUE,
                      time = events$time,
                      sender = events$sender,
                      receiver = events$target,
                      p_samplingobserved = 1.00,
                      n_controls = 5,
                      seed = 9999)
# Creating a actor-varying one-mode relational event sequence where actors
# enter, exit, and re-enter the event sequence dependent upon the user-specified
# active times. Each row contains the actor id, the time for which, in a specific relevant
#spell, become active (enter the sequence) and become inactive (exit the sequence)
#for actors who are always relevant, the active time ranges from 0 to the
#end of the sequence
t <- max(events$time) + rexp(1)
active.times <- data.frame(actor_id = c("A", "B", "B", "C", "D", "E", "F", "G",
```

```
                                    "H" ,"J"),
                          time_start = c(0,0,0.65,0,0,0.45,0.60,0,0,0),
                          time_end = c(t,0.45,t,t,t,1.00,t,t,t,t))

actor.varying.process <- create_res(ordinal = TRUE,
                                    type= "one-mode",
                                    riskset = "compositional",
                                    time = events$time,
                                    sender = events$sender,
                                    receiver = events$target,
                                    active_times = active.times)
```

`dream_information` *dream: A Package for Dynamic Relational Event Analysis and Modeling*

Description

The dream package provides users with helpful functions for relational and event analysis. In particular, dream provides users with helper functions for large relational event analysis, such as recently proposed sampling procedures for creating relational risk sets. Alongside the set of functions for relational event analysis, this package includes functions for the structural analysis of one- and two-mode networks, such as network constraint and effective size measures. This package was developed with support from the National Science Foundation's (NSF) Human Networks and Data Science Program (HNDS) under award number 2241536 (PI: Diego F. Leal). Any opinions, findings, and conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

dream functions

The dream package 'API' is structured into six categories, where the prefix identifies what category the specific function corresponds to (see below):

- `dreamstats_`
- `netstats_om_`
- `netstats_tm_`
- `estimate_rem`
- `simulate_rem_seq`
- `create_res`

The `dreamstats_` functions compute relational/network statistics for relational event sequences. For instance, `dreamstats_fourcycles` computes the four-cycles network statistic for a two-mode relational event sequence. The `create_res` function creates a risk-set for one- and two-mode relational event sequences based on a set of sampling procedures. The `netstats_om_` series of functions compute static network statics for one-mode networks (i.e., `netstats_om_pib` computes

Leal (2025) measure for potential for intercultural brokerage). The `netstats_om_` set of functions compute static network statics for two-mode networks (i.e., `netstats_om_effective` computes Burchard and Cornwell (2018) measure for two-mode ego effective size). The `estimate_rem` functions estimate relational event models for relational event sequences. The function estimates the interval and ordinal timing relational event model and, under certain conditions, can estimate a Cox-proportional hazard model for exact timing relational event models (see Bianchi et al. (2024) and Butts (2008) for more information on these models). Finally, the `simulate_rem_seq` functions simulate one-mode relational event sequences based upon results of a relational event model.

The dream package include two relational event sequences for users to start fitting relational event models: (1) the first 100,000 relational event sequences in the 2018 Wikipedia user article-edit event sequence (`WikiEvent2018.first100k`) and (2) a randomly generated one-mode ordinal timing relational event sequence (`simulated.res`). We encourage users to look at these datasets to look at the structure of relational event sequences that are processed via dream.

Author(s)

Kevin A. Carson kacarson@arizona.edu and Diego F. Leal dflc@arizona.edu

See Also

Useful links:

- `https://github.com/kevinCarson/dream`
- Report bugs at `https://github.com/kevinCarson/dream/issues`

`dream_sequence` *Helper Function to Create* `dream_sequence` *Objects*

Description

The function `dream_sequence()` is a user helper function that transforms user-created processed event sequences into `dream_sequence` objects to be used in the `dream` functions to compute sufficient network statistics and estimate relational event models. This function may also be helpful to user who need to computed network statistics for the estimation of relational outcome models (see Lerner and Hâncean (2023)).

If `ordinal` is FALSE, that is, if the relational event sequence is to use the interval timing likelihood, then the events for the last observation time point (the set of realized and null events at the last time point) should all be control events, as they represent the set of right-censoring non-realized events. The `t` should specify the time point that marks the end of the relational event sequence A_t. If `t` is not known, then the value should be left as `NULL`.

Usage

```
dream_sequence(
  ordinal = TRUE,
  t = NULL,
  time,
```

```
  sender,
  receiver,
  observed = NULL,
  sampled = NULL,
  type = "one-mode",
  statistics = NULL,
  ...
)
```

Arguments

ordinal	TRUE/FALSE. TRUE indicates that observed timing of the events is ordinal (and the ordinal timing likelihood function will be used). FALSE denotes that the observed timing is observed, relative to the start of the event sequence (and the interval timing likelihood function will be used). Please see the references for more information.
t	If ordinal is set to FALSE, the time that marks the end of the relational event sequence, relative to the start of the event sequence.
time	A numeric vector that contains the timing of the events in the relational event sequence.
sender	A character vector that contains the sender of the events in the relational event sequence.
receiver	A character vector that contains the receiver/target of the events in the relational event sequence.
observed	A numeric vector that is 1 if the observation is an observed event in the relational event sequence, or 0 if the observation is a control event in the relational event sequence (see create_res for more information).
sampled	A numeric vector that is 1 if the observation is a sampled event in the relational event sequence, or 0 if the observation is a non-sampled event in the relational event sequence (see create_res for more information).
type	"two-mode" indicates that this is a two-mode event sequence. "one-mode" indicates that the event sequence is one-mode.
statistics	A data.frame object that contains the previously computed statistics for the processed events. If processing has occurred (i.e., the sampled argument is specified; control events have been created and/or if sampling from the observed event sequence), the the statistics argument must have a number of observations equal to the sum of the sampled argument. If processing has not occurred, then the statistics argument must have a number of observations equal to the length of the time argument.
...	Additional arguments (currently unused).

Value

A dream_sequence object that contains the user-provided information.

Examples

```
#a pseudo event sequence
events <- data.frame(time = 1:18, eventID = 1:18,
                     sender = c("A", "B", "C",
                                "A", "D", "E",
                                "F", "B", "A",
                                "F", "D", "B",
                               "G", "B", "D",
                               "H", "A", "D"),
                   target = c("B", "C", "D",
                               "E", "A", "F",
                               "D", "A", "C",
                                "G", "B", "C",
                                "H", "J", "A",
                                "F", "C", "B"))
#making a post-processing event sequence
eventSet <- create_res(type = "one-mode",
                       ordinal = TRUE,
                       riskset = "fixed",
                       case_control=TRUE,
                       time = events$time,
                       sender = events$sender,
                       receiver = events$target,
                       p_samplingobserved = 1.00,
                       n_controls = 1,
                       seed = 9999)

#computing the sender indegree statistics
eventSet <- dreamstats_degree(formation = "sender-indegree", data = eventSet)
#making the dream_sequence object a data.frame object
new.data <- as.data.frame(eventSet) #making the event set a data.frame object
stats <- new.data["sender.indegree"] #the computed statistics

#reconverting the object to a dream_sequence object
psuedo.data <- dream_sequence(ordinal = TRUE,
                              time = new.data$time,
                              sender = new.data$sender,
                              receiver = new.data$receiver,
                              observed = new.data$observed,
                              sampled = new.data$sampled,
                              type= "one-mode",
                              statistics = stats)
psuedo.data #printing the object

#reconverting we original event sequence to a dream_sequence object
#(this is helpful for the estimation of relational outcome models!)
psuedo.data1 <-  dream_sequence(ordinal = TRUE,
                                time = events$time,
                                sender = events$sender,
                                receiver = events$target,
                                type= "one-mode")
```

```
psuedo.data1 #printing the object

#computing a statistic on the data
psuedo.data1 <- dreamstats_degree(formation = "sender-outdegree",
                                  data = psuedo.data1,
                                  counts = TRUE)

psuedo.data1 #printing the object with the computed statistics
psuedo.data1$statistics #printing the object with the computed statistics
```

`dreamstats_actor` *Add Actor-Level Statistics for Event Dyads in a Relational Event Sequence*

Description

[Stable]

This function allows users to add time-varying and time-invariant actor-level statistics to be used in the estimation of ordinal and interval timing relational event models.

Usage

```
dreamstats_actor(
  data,
  var_name,
  sender_ref = TRUE,
  actor_info,
  make_factor = FALSE,
  return_stats = FALSE
)
```

Arguments

`data`	A `dream_sequence` object that contains the processed relational event sequence.
`var_name`	A string that is the name of the variable from the `actor_info` argument that represents the actor-level statistic to be added.
`sender_ref`	TRUE/FALSE. TRUE indicates that the variable should be added with respect to the sender (i.e., the variable is associated with the sender). FALSE indicates that the variable is representative of the event receivers.
`actor_info`	A N x 4 `data.frame` object that contains four columns with the number of observations being the number of referenced actors (i.e., if sender = TRUE, then the senders in the relational event sequence). The object should contain four named columns: (1) `actor_id`, which is the id that is associated with the actors, (2) `var_name` (from the argument), where the var_name column is the vector of statistics, and for time-varying statistics, (3) `time_start`, a column that contains the time that represents the time at which the actor has that statistic, and (4) `time_end`, a column that contains the time for which the actor stops possessing that statistic value (see details for more information).

`make_factor` TRUE/FALSE. TRUE indicates that the vector of values will be made a factor, and FALSE if not.

`return_stats` TRUE/FALSE. TRUE indicates that the vector of computed statistics will be returned. FALSE indicates that the vector of computed statistics will be added to the `statistics` element of the `data` argument.

Details

This function adds user-provided time-varying and time-invariant actor-level statistics to a `dream_sequence` object for relational event models. The `actor_info` argument should be a N x 4 `data.frame` object with four named columns. The first column should be named `actor_id`, which represents the sender/receiver unique ids based upon the observed relational event sequence. The second column should be named `time_start` and represents the time in which the specific actor adopts the specific value. The third column should be named `time_end` and represents the time in which the specific actor adopts the specific value. For time-invariant actor-level statistics, the `time_start` value should be 0 and the `time_end` value should be the time that marks the end of the relational event sequence (e.g., the time of the last observed event). The last column should be named after the `var_name` argument and represents the actor-level statistic for the actor i during the times between `time_start` and `time_end`.

Value

The vector of actor-level statistics for the relational event sequence or the updated `data` argument.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

Examples

```
events <- data.frame(time = 1:18, eventID = 1:18,
                     sender = c("A", "B", "C",
                                "A", "D", "E",
                                "F", "B", "A",
                                "F", "D", "B",
                                "G", "B", "D",
                                "H", "A", "D"),
                     target = c("B", "C", "D",
                                "E", "A", "F",
                               "D", "A", "C",
                                "G", "B", "C",
                                "H", "J", "A",
                                "F", "C", "B"))

processed <- create_res(type = "one-mode",
                        ordinal = TRUE,
                        riskset = "fixed",
                        time = events$time,
                        sender = events$sender,
                        receiver = events$target,
                        p_samplingobserved = 1.00,
```

```
                        case_control=TRUE,
                      n_controls = 1,
                      seed = 9999)

#reconstructing the data.frame object to store the time-varying
#actor-level statistic for sender
ids <- unique(c(events$sender,events$target))
actor_stats <- data.frame(actor_id = ids,
                          time_start = 0,
                          time_end = 18,
                          rv = sample(0:1, length(ids), TRUE))

#adding the value to the post-processing relational event sequence where
#the new variable is named "rv"
processed <- dreamstats_actor(data = processed,
                              var_name = "rv",
                              sender_ref = TRUE,
                              actor_info = actor_stats)
processed
processed$statistics$rv.sender

#constructing the data.frame object to store the time-invariant
#actor-level statistic for receivers
ids <- unique(c(events$sender,events$target))
actor_stats <- data.frame(actor_id = rep(ids,2),
                          time_start = c(rep(0, 9), rep(10,9)),
                          time_end = c(rep(9,9),rep(18,9)),
                          trv = rnorm(length(ids)*2))

#adding the value to the post-processing relational event sequence where
#the new variable is named "trv" and the values is for the event receivers
processed <- dreamstats_actor(data = processed,
                              var_name = "trv",
                              sender_ref = FALSE,
                              actor_info = actor_stats)
processed
processed$statistics$trv.receiver

extract.data <- as.data.frame(processed)
extract.data
```

`dreamstats_actorfe` *Add Actor-Level Fixed Effects for Event Dyads in a Relational Event Sequence*

Description

[Stable]

This function allows users to add event sender and receiver fixed effects for relational event models to a `dream_sequence` object.

Usage

```
dreamstats_actorfe(data, sender = TRUE, return_stats = FALSE)
```

Arguments

data	A dream_sequence object that contains the processed relational event sequence.
sender	TRUE/FALSE. TRUE creates event sender fixed effects and FALSE created event receiver fixed effects.
return_stats	TRUE/FALSE. TRUE indicates that the vector of computed statistics will be returned. FALSE indicates that the vector of computed statistics will be added to the statistics element of the data argument.

Details

This function adds sender or receiver actor-level fixed effects to a dream_sequence object. Internally, the function creates a new variable (senderFE for sender fixed effects and receiverFE for receiver fixed effects) that is a factor of the event sender/receiver ids.

Value

The vector of actor-level fixed effects for the relational event sequence or the updated data argument.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

Examples

```
events <- data.frame(time = 1:18, eventID = 1:18,
                     sender = c("A", "B", "C",
                                "A", "D", "E",
                                "F", "B", "A",
                                "F", "D", "B",
                                "G", "B", "D",
                                "H", "A", "D"),
                     target = c("B", "C", "D",
                                "E", "A", "F",
                               "D", "A", "C",
                                "G", "B", "C",
                                "H", "J", "A",
                                "F", "C", "B"))

processed <- create_res(type = "one-mode",
                        ordinal = TRUE,
                        riskset = "fixed",
                        time = events$time,
                        sender = events$sender,
                        receiver = events$target,
                        p_samplingobserved = 1.00,
```

```
                      case_control=TRUE,
                      n_controls = 20,
                      seed = 9999)

#adding the sender fixed effects to the statistics list
processed <- dreamstats_actorfe(data=processed,sender=TRUE)

#adding the receiver fixed effects to the statistics list
processed <- dreamstats_actorfe(data=processed,sender=FALSE)

processed #the effects are added

#estimating the fixed effects only ordinal timing relational event model
model <- estimate_rem(~senderFE + receiverFE, data = processed)
```

`dreamstats_degree` *Compute Degree Network Statistics for Event Senders and Receivers in a Post-Processing Relational Event Sequence*

Description

[Stable]

The function computes the various degree network sufficient statistic for event senders in a relational event sequence (see Lerner and Lomi 2020; Butts 2008). This measure allows for the degree values to be only computed for the sampled events, while creating the weights based on the full event sequence (see Lerner and Lomi 2020; Vu et al. 2015). The function also allows users to use two different weighting functions, return the counts of past events, reduce computational runtime, and specify a dyadic cutoff for relational relevancy.

Usage

```
dreamstats_degree(
  formation = c("sender-indegree", "receiver-indegree", "sender-outdegree",
    "receiver-outdegree"),
  data,
  halflife = 2,
  counts = FALSE,
  dyadic_weight = 0,
  exp_weight_form = FALSE,
  return_stats = FALSE
)
```

Arguments

`formation`	The degree statistic to be computed. "sender-indegree" computes the indegree statistic for the event senders. "receiver-indegree" computes the indegree statistic for the event receivers. "sender-outdegree" computes the outdegree statistic

for the event senders. "receiver-outdegree" computes the outdegree statistic for the event receivers.

`data` An object of class `dream_sequence` that contains the processed relational event sequence.

`halflife` A numerical value that is the halflife value to be used in the exponential weighting function (see details section). Preset to 2 (should be updated by the user based on substantive context).

`counts` TRUE/FALSE. TRUE indicates that the counts of past events should be computed (see the details section). FALSE indicates that the temporal exponential weighting function should be used to downweigh past events (see the details section). Set to FALSE by default.

`dyadic_weight` A numerical value for the dyadic cutoff weight that represents the numerical cutoff value for temporal relevancy based on the exponential weighting function. For example, a numerical value of 0.01, indicates that an exponential weight less than 0.01 will become 0 and that events with such value (or smaller values) will not be included in the sum of the past event weights (see the details section). Set to 0 by default.

`exp_weight_form` TRUE/FALSE. TRUE indicates that the Lerner et al. (2013) exponential weighting function will be used (see the details section). FALSE indicates that the Lerner and Lomi (2020) exponential weighting function will be used (see the details section). Set to FALSE by default

`return_stats` TRUE/FALSE. TRUE indicates that the vector of computed statistics will be returned. FALSE indicates that the vector of computed statistics will be added to the `statistics` element of the `data` argument.

Details

The function calculates degree values for relational event sequences based on the exponential weighting function used in either Lerner and Lomi (2020) or Lerner et al. (2013).

Following Lerner and Lomi (2020), the exponential weighting function in relational event models is:

$$w(s,r,t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}}$$

Following Lerner et al. (2013), the exponential weighting function in relational event models is:

$$w(s,r,t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}} \cdot \frac{ln(2)}{T_{1/2}}$$

In both of the above equations, *s* is the current event sender, *r* is the current event receiver (target), *t* is the current event time, *t'* is the past event times that meet the weight subset, and $T_{1/2}$ is the halflife parameter.

Sender-Indegree Statistic:

The formula for sender indegree for event e_i is:

$$senderindegree_{e_i} = w(s', s, t)$$

That is, all past events in which the event receiver is the current sender. Following Butts (2008), if the counts of the past events are requested, the formula for sender indegree for event e_i is:

$$senderindegree_{e_i} = d(r' = s, t')$$

Where, $d()$ is the number of past events where the event receiver, *r'*, is the current event sender *s* .

Sender-Outdegree Statistic:

The formula for sender outdegree for event e_i is:

$$senderoutdegree_{e_i} = w(s, r', t)$$

That is, all past events in which the past sender is the current sender and the event target can be any past user. Following Butts (2008), if the counts of the past events are requested, the formula for sender outdegree for event e_i is:

$$senderoutdegree_{e_i} = d(s = s', t')$$

Where, $d()$ is the number of past events where the sender *s'* is the current event sender, *s*

Receiver-Outdegree Statistic:

The formula for receiver outdegree for event e_i is:

$$receiveroutdegree_{e_i} = w(r', r, t)$$

Following Butts (2008), if the counts of the past events are requested, the formula for receiver outdegree for event e_i is:

$$receiveroutdegreee_i = d(s' = r, t')$$

Where, $d()$ is the number of past events where the event sender, *s'*, is the current event receiver, *r'*.

Receiver-Indegree Statistic:

The formula for receiver indegree for event e_i is:

$$recieverindegree_{e_i} = w(s', r, t)$$

That is, all past events in which the event receiver is the current receiver. Following Butts (2008), if the counts of the past events are requested, the formula for receiver indegree for event e_i is:

$$recieverindegree_{e_i} = d(r' = r, t')$$

where, $d()$ is the number of past events where the past event receiver, *r'*, is the current event receiver (target).

Lastly, researchers interested in modeling temporal relevancy (see Quintane, Mood, Dunn, and Falzone 2022; Lerner and Lomi 2020) can specify the dyadic weight cutoff, that is, the minimum value for which the weight is considered relationally relevant. Users who do not know the specific dyadic cutoff value to use, can use the `dreamstats_dyadcut` function.

Value

The vector of degree statistics or the updated `data` argument.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Butts, Carter T. 2008. "A Relational Event Framework for Social Action." *Sociological Methodology* 38(1): 155-200.

Quintane, Eric, Martin Wood, John Dunn, and Lucia Falzon. 2022. "Temporal Brokering: A Measure of Brokerage as a Behavioral Process." *Organizational Research Methods* 25(3): 459-489.

Lerner, Jürgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: How to fit a relational event model to 360 million dyadic events." *Network Science* 8(1): 97-135.

Lerner, Jürgen, Margit Bussman, Tom A.B. Snijders, and Ulrik Brandes. 2013. " Modeling Frequency and Type of Interaction in Event Networks." *The Corvinus Journal of Sociology and Social Policy* 4(1): 3-32.

Vu, Duy, Philippa Pattison, and Garry Robins. 2015. "Relational event models for social learning in MOOCs." *Social Networks* 43: 121-135.

Examples

```
events <- data.frame(time = 1:18, eventID = 1:18,
                                sender = c("A", "B", "C",
                                           "A", "D", "E",
                                           "F", "B", "A",
                                           "F", "D", "B",
                                           "G", "B", "D",
                                          "H", "A", "D"),
                                target = c("B", "C", "D",
                                           "E", "A", "F",
                                           "D", "A", "C",
                                           "G", "B", "C",
                                           "H", "J", "A",
                                           "F", "C", "B"))

eventSet <- create_res(type = "one-mode",
                       ordinal = TRUE,
                       riskset = "fixed",
                       time = events$time,
                       sender = events$sender,
                       receiver = events$target,
                       p_samplingobserved = 1.00,
                       case_control=TRUE,
                       n_controls = 1,
                       seed = 9999)

#Computing the sender indegree statistic for the relational event sequence
eventSet <- dreamstats_degree(
   formation = "sender-indegree",
   data = eventSet,
   halflife = 2, #halflife parameter
```

```
   dyadic_weight = 0,
   exp_weight_form = FALSE)

eventSet #printing the post-processed relational event sequence
eventSet$statistics$sender.indegree #printing the vector of computed values

#Computing the sender indegree statistic for the relational event sequence
#and returning only the vector of computed sender indegree values
degree.stat <- dreamstats_degree(
   formation = "sender-indegree",
   data = eventSet,
   halflife = 2, #halflife parameter
   dyadic_weight = 0,
   exp_weight_form = FALSE,
   return_stats = TRUE)

cor(degree.stat, eventSet$statistics$sender.indegree)

#Computing the sender outdegree statistic for the relational event sequence
eventSet <- dreamstats_degree(
   formation = "sender-outdegree",
   data = eventSet,
   halflife = 2, #halflife parameter
   dyadic_weight = 0,
   exp_weight_form = FALSE)


#Computing the receiver outdegree statistic for the relational event sequence
eventSet <- dreamstats_degree(
   formation = "receiver-outdegree",
   data = eventSet,
   halflife = 2, #halflife parameter
   dyadic_weight = 0,
   exp_weight_form = FALSE)


#Computing the receiver indegree statistic for the relational event sequence
eventSet <- dreamstats_degree(
   formation = "receiver-indegree",
   data = eventSet,
   halflife = 2, #halflife parameter
   dyadic_weight = 0,
   exp_weight_form = FALSE)

#printing the post-processed relational event sequence that contains all computed degree statistics
degree.info <- as.data.frame(eventSet)
degree.info #printing the information to the user
```

`dreamstats_dyadcut` *A Helper Function to Assist Researchers in Finding Dyadic Weight Cutoff Values*

Description

[Stable]

A user-helper function to assist researchers in finding the dyadic cutoff value to compute sufficient statistics for relational event models based upon temporal dependency.

Usage

```
dreamstats_dyadcut(halflife = 2, relationalWidth, exp_weight_form = FALSE)
```

Arguments

`halflife`	A numerical value that is the halflife value to be used in the exponential weighting function (see details section). Preset to 2 (should be updated by the user based on substantive context).
`relationalWidth`	The numerical value that corresponds to the time range for which the user specifies for temporal relevancy.
`exp_weight_form`	TRUE/FALSE. TRUE indicates that the Lerner et al. (2013) exponential weighting function will be used (see the details section). FALSE indicates that the Lerner and Lomi (2020) exponential weighting function will be used (see the details section). Set to FALSE by default

Details

This function is specifically designed as a user-helper function to assist researchers in finding the dyadic cutoff value for creating sufficient statistics based upon temporal dependency. In other words, this function estimates a dyadic cutoff value for relational relevance, that is, the minimum dyadic weight for past events to be potentially relevant (i.e., to possibly have an impact) on the current event. All non-relevant events (i.e., events too distant in the past from the current event to be considered relevant, that is, those below the cutoff value) will have a weight of 0. This cutoff value is based upon two user-specified values: the events' halflife (i..e, `halflife`) and the relationally relevant event or time span (i.e., `relationalWidth`). Ideally, both the values for `halflife` and `relationalWidth` would be based on the researcher's command of the relevant substantive literature. Importantly, `halflife` and `relationalWidth` must be in the same units of measurement (e.g., days). If not, the function will not return the correct answer.

For example, let's say that the user defines the `halflife` to be 15 days (i.e., two weeks) and the relationally relevant event or time span (i.e., `relationalWidth`) to be 30 days (i.e., events that occurred more than 1 month in the past are not considered relationally relevant for the current event). The user would then specify `halflife` = 15 and `relationalWidth` = 30.

Following Lerner and Lomi (2020), the exponential weighting function in relational event models is:

$$w(s, r, t) = e^{-(t-t') \cdot \frac{ln(2)}{T_{1/2}}}$$

Following Lerner et al. (2013), the exponential weighting function in relational event models is:

$$w(s,r,t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}} \cdot \frac{ln(2)}{T_{1/2}}$$

In both of the above equations, *s* is the current event sender, *r* is the current event receiver (target), *t* is the current event time, *t'* is the past event times that meet the weight subset, and $T_{1/2}$ is the halflife parameter. The task of this function is to find the weight, $w(s,r,t)$, that corresponds to the time difference provided by the user.

Value

The dyadic weight cutoff based on user specified values.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Lerner, Jürgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: How to fit a relational event model to 360 million dyadic events." *Network Science* 8(1): 97-135.

Lerner, Jürgen, Margit Bussman, Tom A.B. Snijders, and Ulrik Brandes. 2013. " Modeling Frequency and Type of Interaction in Event Networks." *The Corvinus Journal of Sociology and Social Policy* 4(1): 3-32.

Examples

```
#To replicate the example in the details section:
# with the Lerner et al. 2013 weighting function
dreamstats_dyadcut(halflife = 15,
                relationalWidth = 30,
                exp_weight_form = TRUE)

# without the Lerner et al. 2013 weighting function
dreamstats_dyadcut(halflife = 15,
                relationalWidth = 30,
                exp_weight_form = FALSE)

# A result to test the function (should come out to 0.50)
dreamstats_dyadcut(halflife = 30,
                relationalWidth = 30,
                exp_weight_form = FALSE)


# Replicating Lerner and Lomi (2020):
#"We set T1/2 to 30 days so that an event counts as (close to) one in the very next instant of time,
#it counts as 1/2 one month later, it counts as 1/4 two months after the event, and so on. To reduce
#the memory consumption needed to store the network of past events, we set a dyadic weight to
#zero if its value drops below 0.01. If a single event occurred in some dyad this would happen after
```

```
#6.64×T1/2, that is after more than half a year." (Lerner and Lomi 2020: 104).

# Based upon Lerner and Lomi (2020: 104), the result should be around 0.01. Since the
# time values in Lerner and Lomi (2020) are in milliseconds, we have to change
# all measurements into milliseconds
dreamstats_dyadcut(halflife = (30*24*60*60*1000), #30 days in milliseconds
                relationalWidth = (6.64*30*24*60*60*1000), #Based upon the paper
                #using the Lerner and Lomi (2020) weighting function
                exp_weight_form = FALSE)
```

`dreamstats_dyadfe` *Add Dyadic-Level Fixed Effects for Event Dyads in a Relational Event Sequence*

Description

[Stable]

This function allows users to add dyad-level fixed effects for relational event models to a `dream_sequence` object.

Usage

```
dreamstats_dyadfe(data, directed = TRUE, return_stats = FALSE)
```

Arguments

`data`	A `dream_sequence` object that contains the processed relational event sequence.
`directed`	TRUE/FALSE. TRUE indicates that the dyadic-level fixed effects will be generated based upon the ordering of the sending and receiving actors (i.e., AB != BA). FALSE indicates that the dyadic-level fixed effects will be generated based upon the combo of the sending and receiving actors (i.e., AB == BA). Set to TRUE by default. Of course, these will lead to (1) a different number of fixed effects and (2) a different interpretation of the results.
`return_stats`	TRUE/FALSE. TRUE indicates that the vector of computed statistics will be returned. FALSE indicates that the vector of computed statistics will be added to the `statistics` element of the `data` argument.

Details

This function adds dyad-level fixed effects to a `dream_sequence` object. Internally, the function creates a new variable named "dyad.fe" which is a factor of the combined event sender/receiver ids.

Value

The vector of actor-level fixed effects for the relational event sequence or the updated `data` argument.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

Examples

```
events <- data.frame(time = 1:18, eventID = 1:18,
                     sender = c("A", "B", "C",
                                "A", "D", "E",
                                "F", "B", "A",
                                "F", "D", "B",
                                "G", "B", "D",
                                "H", "A", "D"),
                     target = c("B", "C", "D",
                                "E", "A", "F",
                               "D", "A", "C",
                                "G", "B", "C",
                                "H", "J", "A",
                                "F", "C", "B"))

processed <- create_res(type = "one-mode",
                        ordinal = TRUE,
                        riskset = "fixed",
                        time = events$time,
                        sender = events$sender,
                        receiver = events$target,
                        seed = 9999)

#adding the dyadic fixed effects to the statistics list
processed <- dreamstats_dyadfe(data=processed)

#estimating the dyadic fixed effects only ordinal timing relational event model
model <- estimate_rem(~dyad.fe, data = processed)
```

---

dreamstats_dyadic *Add Dyadic-Level Statistics for Event Dyads in a Relational Event Sequence*

Description

[Stable]

This function allows users to add time-varying and time-invariant dyadic-level statistics that impact the dyadic event rates in relational event models.

Usage

```
dreamstats_dyadic(
  data,
  var_name,
```

```
  transformation = c("same", "abs.diff", "inv.abs.diff"),
  return_stats = FALSE
)
```

Arguments

`data`	A `dream_sequence` object that contains the processed relational event sequence.
`var_name`	A string that is the name of the variable from the `statistics` element in the `dream_sequence` argument for which the statistic should be created.
`transformation`	The type of transformation for how the sender and receiver values will be compared (see details). The following arguments are provided: "same", "abs.diff", and "inv.abs.diff".
`return_stats`	TRUE/FALSE. TRUE indicates that the vector of computed statistics will be returned. FALSE indicates that the vector of computed statistics will be added to the `statistics` element of the `data` argument.

Details

This function adds user-provided time-varying and time-invariant dyadic-level statistics to a `dream_sequence` object for relational event models. For this function to work, in the object provided to the `data` argument, the `statistics` element must contain the following previously computed statistics: `var_name.sender` and `var_name.receiver`, where `var_name` is the user-provided argument. For instance, if the `var_name` is `male`, then the two variables need to be included in the `statistics` list: `male.sender` and `male.receiver`.

The function allows for three types of transformations to compare the values for the event senders and event receivers. When the `transformation` argument is `same`, then the values are 1 if the event sender and receivers are the same, and 0 if not. When the `transformation` argument is `abs.diff`, then the values are $|sender - receiver|$. Finally, when the `transformation` argument is `inv.abs.diff`, then the values are $1/|sender - receiver|$ (if the difference is 0, the value is set to 1).

Value

The vector of dyadic-level statistics for the relational event sequence or the updated `data` argument.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

Examples

```
events <- data.frame(time = 1:18, eventID = 1:18,
                     sender = c("A", "B", "C",
                                "A", "D", "E",
                                "F", "B", "A",
                                "F", "D", "B",
                                "G", "B", "D",
                                "H", "A", "D"),
                     target = c("B", "C", "D",
```

```
                                "E", "A", "F",
                               "D", "A", "C",
                                "G", "B", "C",
                                "H", "J", "A",
                                "F", "C", "B"))

processed <- create_res(type = "one-mode",
                        ordinal = TRUE,
                        riskset = "fixed",
                        time = events$time,
                        sender = events$sender,
                        receiver = events$target,
                        p_samplingobserved = 1.00,
                           case_control=TRUE,
                        n_controls = 1,
                        seed = 9999)

#reconstructing the data.frame object to store the time-varying
#actor-level statistic for receivers
ids <- unique(c(events$sender,events$target))
actor_stats <- data.frame(actor_id = ids,
                          time_start = 0,
                          time_end = 18,
                          male = sample(0:1, length(ids), TRUE))

#adding the value to the post-processing relational event sequence where
#the new variable is named "male"
processed <- dreamstats_actor(data = processed,
                              var_name = "male",
                              sender_ref = TRUE,
                              actor_info = actor_stats)
processed
processed$statistics$male.sender
#adding the value for the event receivers
processed <- dreamstats_actor(data = processed,
                              var_name = "male",
                              sender_ref = FALSE,
                              actor_info = actor_stats)
processed
processed$statistics$male.receiver


#adding the dyadic same value to the post-processing relational event sequence where
#the new variable is 1 if the event sender and receiver have the same value
# and 0 if not
processed <- dreamstats_dyadic(data = processed,
                              var_name = "male",
                              transformation = "same")
processed
processed$statistics$male.same

extract.data <- as.data.frame(processed)
extract.data #checking the values are the same!
```

`dreamstats_event` *Add Event-Level Statistics for a Relational Event Sequence*

Description

[Stable]

This function allows users to add event-level statistics that impact the event rates in interval timing relational event model, such as statistics that impact the waiting times between events.

Usage

```
dreamstats_event(
  data,
  var_name,
  event_info,
  make_factor = FALSE,
  return_stats = FALSE
)
```

Arguments

`data`	A `dream_sequence` object that contains the processed relational event sequence.
`var_name`	A string that is the name of the variable from the `event_info` argument that represents the actor-level statistic to be added.
`event_info`	A N x 2 `data.frame` object that contains two columns with the number of observations being the number of observed time points actors. The object should contain two named columns: (1) `time_id`, which is the time associated with each event and (2) `var_name` (from the argument), where the var_name column is the vector of statistics (each time point has one associated value; see details for more information).
`make_factor`	TRUE/FALSE. TRUE indicates that the vector of values will be made a factor, and FALSE if not.
`return_stats`	TRUE/FALSE. TRUE indicates that the vector of computed statistics will be returned. FALSE indicates that the vector of computed statistics will be added to the `statistics` element of the `data` argument.

Details

This function adds user-provided time-varying and time-invariant actor-level statistics to a `dream_sequence` object for relational event models. The `event_info` argument should be a N x 2 `data.frame` object with two named columns. The first column should be named `time_id`, which represents the observed time points based upon the relational event sequence. The second column should be named after the `var_name` argument and represents the event-level statistic for that i time point (i.e., the ith `time_id`).

Value

The vector of event-level statistics for the relational event sequence or the updated `data` argument.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

Examples

```
events <- data.frame(time = 1:18, eventID = 1:18,
                     sender = c("A", "B", "C",
                                "A", "D", "E",
                                "F", "B", "A",
                                "F", "D", "B",
                                "G", "B", "D",
                                "H", "A", "D"),
                     target = c("B", "C", "D",
                                "E", "A", "F",
                               "D", "A", "C",
                                "G", "B", "C",
                                "H", "J", "A",
                                "F", "C", "B"))

processed <- create_res(type = "one-mode",
                        ordinal = TRUE,
                        time = events$time,
                        riskset = "fixed",
                        sender = events$sender,
                        receiver = events$target,
                        p_samplingobserved = 1.00,
                        case_control=TRUE,
                        n_controls = 5,
                        seed = 9999)

#reconstructing the data.frame object to store the time-varying
#event-level statistic
event_stats <- data.frame(time_id = events$time,
                          oliver = rnorm(nrow(events)))

#reconstructing the data.frame object to store the time-varying
#event-level statistic
processed <- dreamstats_event(data = processed,
                              var_name = "oliver",
                              event_info = event_stats)
processed
processed$statistics$oliver.event

extract.data <- as.data.frame(processed)
extract.data
```

`dreamstats_fourcycles` *Compute the Four-Cycles Network Statistic for Event Dyads in a Relational Event Sequence*

Description

[Stable]

The function computes the four-cycles network sufficient statistic for a two-mode relational sequence with the exponential weighting function (Lerner and Lomi 2020). In essence, the four-cycles measure captures the tendency for clustering to occur in the network of past events, whereby an event is more likely to occur between a sender node *a* and receiver node *b* given that *a* has interacted with other receivers in past events who have received events from other senders that interacted with *b* (e.g., Duxbury and Haynie 2021, Lerner and Lomi 2020). The function also allows users to use two different weighting functions, return the counts of past events, reduce computational runtime, and specify a dyadic cutoff for relational relevancy.

Usage

```
dreamstats_fourcycles(
  data,
  halflife = 2,
  counts = FALSE,
  dyadic_weight = 0,
  exp_weight_form = FALSE,
  return_stats = FALSE
)
```

Arguments

`data`	An object of class `dream_sequence` that contains the processed relational event sequence.
`halflife`	A numerical value that is the halflife value to be used in the exponential weighting function (see details section). Preset to 2 (should be updated by the user based on substantive context).
`counts`	TRUE/FALSE. TRUE indicates that the counts of past events should be computed (see the details section). FALSE indicates that the temporal exponential weighting function should be used to downweigh past events (see the details section). Set to FALSE by default.
`dyadic_weight`	A numerical value for the dyadic cutoff weight that represents the numerical cutoff value for temporal relevancy based on the exponential weighting function. For example, a numerical value of 0.01, indicates that an exponential weight less than 0.01 will become 0 and that events with such value (or smaller values) will not be included in the sum of the past event weights (see the details section). Set to 0 by default.

`exp_weight_form`
: TRUE/FALSE. TRUE indicates that the Lerner et al. (2013) exponential weighting function will be used (see the details section). FALSE indicates that the Lerner and Lomi (2020) exponential weighting function will be used (see the details section). Set to FALSE by default

`return_stats`
: TRUE/FALSE. TRUE indicates that the vector of computed statistics will be returned. FALSE indicates that the vector of computed statistics will be added to the `statistics` element of the `data` argument.

Details

The function calculates the four-cycles network statistic for two-mode relational event models based on the exponential weighting function used in either Lerner and Lomi (2020) or Lerner et al. (2013).

Following Lerner and Lomi (2020), the exponential weighting function in relational event models is:

$$w(s,r,t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}}$$

Following Lerner et al. (2013), the exponential weighting function in relational event models is:

$$w(s,r,t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}} \cdot \frac{ln(2)}{T_{1/2}}$$

In both of the above equations, *s* is the current event sender, *r* is the current event receiver (target), *t* is the current event time, *t'* is the past event times that meet the weight subset (in this case, all events that have the same sender and receiver), and $T_{1/2}$ is the halflife parameter.

The formula for four-cycles for event e_i is:

$$fourcycles_{e_i} = \sqrt[3]{\sum_{s' and r'} w(s',r,t)\cdot w(s,r',t)\cdot w(s',r',t)}$$

That is, the four-cycle measure captures all the past event structures in which the current event pair, sender *s* and target *r* close a four-cycle. In particular, it finds all events in which: a past sender *s'* had a relational event with target *r*, a past target *r'* had a relational event with current sender *s*, and finally, a relational event occurred between sender *s'* and target *r'*.

Four-cycles are computationally expensive, especially for large relational event sequences (see Lerner and Lomi 2020 for a discussion on this), therefore this function allows the user to input previously computed target indegree and sender outdegree scores to reduce the runtime. Relational events where either the event target or event sender were not involved in any prior relational events (i.e., a target indegree or sender outdegree score of 0) will close no-four cycles. This function exploits this feature.

Moreover, researchers interested in modeling temporal relevancy (see Quintane, Mood, Dunn, and Falzone 2022; Lerner and Lomi 2020) can specify the dyadic weight cutoff, that is, the minimum value for which the weight is considered relationally relevant. Users who do not know the specific dyadic cutoff value to use, can use the `dreamstats_dyadcut` function.

Following Lerner and Lomi (2020), if the counts of the past events are requested, the formula for four-cycles formation for event e_i is:

$$fourcycles_{e_i} = \sum_{i=1}^{|S'|}\sum_{j=1}^{|R'|} \min\left[d(s'_i,r,t),\ d(s,r'_j,t),\ d(s'_i,r'_j,t)\right]$$

where, $d()$ is the number of past events that meet the specific set operations, $d(s_i', r, t)$ is the number of past events where the current event receiver received a tie from another sender s_i', $d(s, r_j', t)$ is the number of past events where the current event sender sent a tie to another receiver r_j', and $d(s_i', r_j', t)$ is the number of past events where the sender s_i' sent a tie to the receiver r_j'. Moreover, the counting equation can leverage relational relevancy, by specifying the halflife parameter, exponential weighting function, and the dyadic cut off weight values (see the above sections for help with this). If the user is not interested in modeling relational relevancy, then those value should be left at their default values.

Value

The vector of four-cycle statistics for the two-mode relational event sequence or the updated `data` argument.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Duxbury, Scott and Dana Haynie. 2021. "Shining a Light on the Shadows: Endogenous Trade Structure and the Growth of an Online Illegal Market." *American Journal of Sociology* 127(3): 787-827.

Quintane, Eric, Martin Wood, John Dunn, and Lucia Falzon. 2022. "Temporal Brokering: A Measure of Brokerage as a Behavioral Process." *Organizational Research Methods* 25(3): 459-489.

Lerner, Jürgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: How to fit a relational event model to 360 million dyadic events." *Network Science* 8(1): 97-135.

Lerner, Jürgen, Margit Bussman, Tom A.B. Snijders, and Ulrik Brandes. 2013. "Modeling Frequency and Type of Interaction in Event Networks." *The Corvinus Journal of Sociology and Social Policy* 4(1): 3-32.

Examples

```
data("WikiEvent2018.first100k", package = "dream")
WikiEvent2018 <- WikiEvent2018.first100k[1:1000,] #the first one thousand events
WikiEvent2018$time <- as.numeric(WikiEvent2018$time) #making the variable numeric
### Creating the EventSet By Employing Case-Control Sampling With M = 5 and
### Sampling from the Observed Event Sequence with P = 0.01
post.processing <- create_res(type = "two-mode",
 ordinal = TRUE,
 riskset = "fixed",
 time = WikiEvent2018$time, # The Time Variable
 sender = WikiEvent2018$user, # The Sender Variable
 receiver = WikiEvent2018$article, # The Receiver Variable
 p_samplingobserved = 0.01, # The Probability of Selection
    case_control=TRUE,
 n_controls = 8, # The Number of Controls to Sample from the Full Risk Set
 seed = 9999)
```

```
#Computing the four-cycles statistics for the relational event sequence with
#the exponential weights of past events returned
post.processing <- dreamstats_fourcycles(data = post.processing,
                                         halflife = 2.592e+09)

#printing the post-processed relational event sequence
post.processing


#Computing the four-cycles statistic for the relational event sequence
#and returning only the vector of computed values
cycle4.stat <-  dreamstats_fourcycles(data = post.processing,
                                      halflife = 2.592e+09,
                                      return_stats = TRUE)

cor(cycle4.stat, post.processing$statistics$four.cycles)

#Computing the four-cycles statistics for the relational event sequence with
#the counts of past events returned
post.processing <- dreamstats_fourcycles(data = post.processing,
                                         halflife = 2.592e+09,
                                         counts = TRUE)

cbind(post.processing$statistics$four.cycles, cycle4.stat)
```

`dreamstats_persistence` *Compute Butts' (2008) Persistence Network Statistic for Event Dyads in a Relational Event Sequence*

Description

[Stable]

This function computes the persistence network sufficient statistic for a relational event sequence (see Butts 2008). Persistence measures the proportion of past ties sent from the event sender that went to the current event receiver. Furthermore, this measure allows for persistence scores to be only computed for the sampled events, while creating the weights based on the full event sequence. Moreover, the function allows users to specify relational relevancy for the resulting statistic.

Usage

```
dreamstats_persistence(
  data,
  ref_sender = TRUE,
  nopastEvents = NA,
  dependency = FALSE,
  relationalTimeSpan = 0,
```

```
  return_stats = FALSE
)
```

Arguments

`data`	A `dream_sequence` object that contains the processed relational event sequence.
`ref_sender`	TRUE/FALSE. TRUE indicates that the persistence statistic will be computed in reference to the sender's past relational history (see details section). FALSE indicates that the persistence statistic will be computed in reference to the target's past relational history (see details section). Set to TRUE by default.
`nopastEvents`	The numerical value that specifies what value should be given to events in which the sender has sent not past ties (i's neighborhood when sender = TRUE) or has not received any past ties (j's neighborhood when sender = FALSE). Set to NA by default.
`dependency`	TRUE/FALSE. TRUE indicates that temporal relevancy will be modeled (see the details section). FALSE indicates that temporal relevancy will not be modeled, that is, all past events are relevant (see the details section). Set to FALSE by default.
`relationalTimeSpan`	If dependency = TRUE, a numerical value that corresponds to the temporal span for relational relevancy, which must be the same measurement unit as the observed_time and processed_time objects. When dependency = TRUE, the relevant events are events that have occurred between current event time, *t*, and *t-relationalTimeSpan*. For example, if the time measurement is the number of days since the first event and the value for relationalTimeSpan is set to 10, then only those events which occurred in the past 10 days are included in the computation of the statistic.
`return_stats`	TRUE/FALSE. TRUE indicates that the vector of computed statistics will be returned. FALSE indicates that the vector of computed statistics will be added to the `statistics` element of the `data` argument.

Details

The function calculates the persistence network sufficient statistic for a relational event sequence based on Butts (2008).

The formula for persistence for event e_i with reference to the sender's past relational history is:

$$Persistence_{e_i} = \frac{d(s(e_i), r(e_i), A_t)}{d(s(e_i), A_t)}$$

where $d(s(e_i), r(e_i), A_t)$ is the number of past events where the current event sender sent a tie to the current event receiver, and $d(s(e_i), A_t)$ is the number of past events where the current sender sent a tie.

The formula for persistence for event e_i with reference to the target's past relational history is:

$$Persistence_{e_i} = \frac{d(s(e_i), r(e_i), A_t)}{d(r(e_i), A_t)}$$

where $d(s(e_i), r(e_i), A_t)$ is the number of past events where the current event sender sent a tie to the current event receiver, and $d(r(e_i), A_t)$ is the number of past events where the current receiver recieved a tie.

Moreover, researchers interested in modeling temporal relevancy (see Quintane, Mood, Dunn, and Falzone 2022) can specify the relational time span, that is, length of time for which events are considered relationally relevant. This should be specified via the option *relationalTimeSpan* with *dependency* set to TRUE.

Value

The vector of persistence network statistics for the relational event sequence or the updated `data` argument.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Butts, Carter T. 2008. "A relational event framework for social action." *Sociological Methodology* 38(1): 155-200.

Quintane, Eric, Martin Wood, John Dunn, and Lucia Falzon. 2022. "Temporal Brokering: A Measure of Brokerage as a Behavioral Process." *Organizational Research Methods* 25(3): 459-489.

Examples

```
# A Psuedo One-Mode Event Dataset
events <- data.frame(time = 1:18,
                                 eventID = 1:18,
                                 sender = c("A", "B", "C",
                                            "A", "D", "E",
                                            "F", "B", "A",
                                            "F", "D", "B",
                                            "G", "B", "D",
                                           "H", "A", "D"),
                                target = c("B", "C", "D",
                                           "E", "A", "F",
                                           "D", "A", "C",
                                           "G", "B", "C",
                                           "H", "J", "A",
                                           "F", "C", "B"))

post.processing <- create_res(type = "one-mode",
                      ordinal = TRUE,
                      riskset = "fixed",
                      time = events$time,
                      sender = events$sender,
                      receiver = events$target,
                      p_samplingobserved = 1.00,
                         case_control=TRUE,
                      n_controls = 1,
```

```
                      seed = 9999)

#Computing the persistence statistic for the relational event sequence
post.processing <- dreamstats_persistence(data = post.processing)

#printing the post-processed relational event sequence
post.processing


#Computing the persistence statistic for the relational event sequence
#and returning only the vector of computed values
persistence.stat <- dreamstats_persistence(data = post.processing,
                                           return_stats = TRUE)

cor(persistence.stat, post.processing$statistics$persistence)
```

`dreamstats_prefattachment` *Compute Butts' (2008) Preferential Attachment Network Statistic for Event Dyads in a Relational Event Sequence*

Description

[Stable]

The function computes the preferential attachment network sufficient statistic for a relational event sequence (see Butts 2008). Preferential attachment measures the tendency towards a positive feedback loop in which actors involved in more past events are more likely to be involved in future events (see Butts 2008 for an empirical example and discussion).This measure allows for preferential attachment scores to be only computed for the sampled events, while creating the statistics based on the full event sequence. Moreover, the function allows users to specify relational relevancy for the resulting statistics.

Usage

```
dreamstats_prefattachment(
  data,
  dependency = FALSE,
  relationalTimeSpan = 0,
  return_stats = FALSE
)
```

Arguments

`data`	A `dream_sequence` object that contains the processed relational event sequence.
`dependency`	TRUE/FALSE. TRUE indicates that temporal relevancy will be modeled (see the details section). FALSE indicates that temporal relevancy will not be modeled, that is, all past events are relevant (see the details section). Set to FALSE by default.

`relationalTimeSpan`
: If dependency = TRUE, a numerical value that corresponds to the temporal span for relational relevancy, which must be the same measurement unit as the observed_time and processed_time objects. When dependency = TRUE, the relevant events are events that have occurred between current event time, *t*, and *t - relationalTimeSpan*. For example, if the time measurement is the number of days since the first event and the value for relationalTimeSpan is set to 10, then only those events which occurred in the past 10 days are included in the computation of the statistic.

`return_stats`
: TRUE/FALSE. TRUE indicates that the vector of computed statistics will be returned. FALSE indicates that the vector of computed statistics will be added to the `statistics` element of the `data` argument.

Details

The function calculates preferential attachment for a relational event sequence based on Butts (2008).

Following Butts (2008), the formula for preferential attachment for event e_i is:

$$PA_{e_i} = \frac{d^+(r(e_i), A_t) + d^-(r(e_i), A_t)}{\sum_{i=1}^{|S|}(d^+(i, A_t) + d^-(i, A_t))}$$

where $d^+(r(e_i), A_t)$ is the past outdegree of the receiver for e_i, $d^-(r(e_i), A_t)$ is the past indegree of the receiver for e_i, $\sum_{i=1}^{|S|}(d^+(i, A_t) + d^-(i, A_t))$ is the sum of the past outdegree and indegree for all past event senders in the relational history.

Moreover, researchers interested in modeling temporal relevancy (see Quintane, Mood, Dunn, and Falzone 2022) can specify the relational time span, that is, length of time for which events are considered relationally relevant. This should be specified via the option *relationalTimeSpan* with *dependency* set to TRUE.

Value

The vector of event preferential attachment statistics for the relational event sequence or the updated `data` argument.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Butts, Carter T. 2008. "A relational event framework for social action." *Sociological Methodology* 38(1): 155-200.

Quintane, Eric, Martin Wood, John Dunn, and Lucia Falzon. 2022. "Temporal Brokering: A Measure of Brokerage as a Behavioral Process." *Organizational Research Methods* 25(3): 459-489.

Examples

```
# A Psuedo One-Mode Event Dataset
events <- data.frame(time = 1:18,
                                eventID = 1:18,
                                sender = c("A", "B", "C",
                                           "A", "D", "E",
                                           "F", "B", "A",
                                           "F", "D", "B",
                                           "G", "B", "D",
                                          "H", "A", "D"),
                               target = c("B", "C", "D",
                                          "E", "A", "F",
                                          "D", "A", "C",
                                          "G", "B", "C",
                                          "H", "J", "A",
                                          "F", "C", "B"))

post.processing <- create_res(type = "one-mode",
                      ordinal = TRUE,
                      riskset = "fixed",
                      time = events$time,
                      sender = events$sender,
                      receiver = events$target,
                      p_samplingobserved = 1.00,
                         case_control=TRUE,
                      n_controls = 1,
                      seed = 9999)

#Computing the preferential attachment statistic for the relational event sequence
post.processing <- dreamstats_prefattachment(data = post.processing)

#printing the post-processed relational event sequence
post.processing


#Computing the preferential attachment statistic for the relational event sequence
#and returning only the vector of computed values
prefattach.stat <- dreamstats_prefattachment(data = post.processing,
                                             return_stats = TRUE)

cor(prefattach.stat, post.processing$statistics$pref.attachment)
```

`dreamstats_recency` *Compute Butts' (2008) Recency Network Statistic for Event Dyads in a Relational Event Sequence*

Description

[Stable]

This function computes the recency network sufficient statistic for a relational event sequence (see Butts 2008; Vu et al. 2015; Meijerink-Bosman et al. 2022). The recency statistic captures the tendency for more recent events (i.e., an exchange between two medical doctors) are more likely to re-occur in comparison to events that happened in the more distant past (see Butts 2008 for a discussion). This measure allows for recency scores to be only computed for the sampled events, while computing the statistics based on the full event sequence.

Usage

```
dreamstats_recency(
  data,
  type = c("raw.diff", "inv.diff.plus1", "rank.ordered.count"),
  i_neighborhood = TRUE,
  dependency = FALSE,
  relationalTimeSpan = NULL,
  nopastEvents = NA,
  return_stats = FALSE
)
```

Arguments

`data`	A `dream_sequence` object that contains the processed relational event sequence.
`type`	A string value that specifies which recency formula will be used to compute the statistics. The options are "raw.diff", "inv.diff.plus1", "rank.ordered.count" (see details section).
`i_neighborhood`	TRUE/FALSE. TRUE indicates that the recency statistic will be computed in reference to the sender's past relational history (see details section). FALSE indicates that the recency statistic will be computed in reference to the target's past relational history (see details section). Set to TRUE by default.
`dependency`	TRUE/FALSE. TRUE indicates that temporal relevancy will be modeled (see details section). FALSE indicates that temporal relevancy will not be modeled, that is, all past events are relevant (see details section). Set to FALSE by default.
`relationalTimeSpan`	If dependency = TRUE, a numerical value that corresponds to the temporal span for relational relevancy, which must be the same measurement unit as the observed_time and processed_time objects. When dependency = TRUE, the relevant events are events that have occurred between current event time, *t*, and *t - relationalTimeSpan*. For example, if the time measurement is the number of days since the first event and the value for relationalTimeSpan is set to 10, then only those events which occurred in the past 10 days are included in the computation of the statistic.
`nopastEvents`	The numerical value that specifies what value should be given to events in which the sender was not active as a sender in the past (i's neighborhood when i_neighborhood = TRUE) or was not the recipient of a past event (j's neighborhood when i_neighborhood = FALSE). Set to NA by default.
`return_stats`	TRUE/FALSE. TRUE indicates that the vector of computed statistics will be returned. FALSE indicates that the vector of computed statistics will be added to the `statistics` element of the `data` argument.

Details

This function calculates the recency network sufficient statistic for a relational event based on Butts (2008), Vu et al. (2015), or Meijerink-Bosman et al. (2022). Depending on the type and neighborhood requested, different formulas will be used.

In the below equations, when *i_neighborhood* is TRUE:

$$t^* = max(t \in \{(s', r', t') \in E : s' = s \wedge r' = r \wedge t' < t\})$$

When *i_neighborhood* is FALSE, the following formula is used:

$$t^* = max(t \in \{(s', r', t') \in E : s' = r \wedge r' = s \wedge t' < t\})$$

The formula for recency for event e_i with type set to "raw.diff" and *i_neighborhood* is TRUE (Vu et al. 2015):

$$recency_{e_i} = t_{e_i} - t^*$$

where t^*, is the most recent time in which the past event has the same receiver and sender as the current event. If there are no past events within the current dyad, then the value defaults to the *nopastEvents* argument.

The formula for recency for event e_i with type set to "raw.diff" and *i_neighborhood* is FALSE (Vu et al. 2015):

$$recency_{e_i} = t_{e_i} - t^*$$

where t^*, is the most recent time in which the past event's sender is the current event receiver and the past event receiver is the current event sender. If there are no past events within the current dyad, then the value defaults to the *nopastEvents* argument.

The formula for recency for event e_i with type set to "inv.diff.plus1" and *i_neighborhood* is TRUE (Meijerink-Bosman et al. 2022):

$$recency_{e_i} = \frac{1}{t_{e_i} - t^* + 1}$$

where t^*, is the most recent time in which the past event has the same receiver and sender as the current event. If there are no past events within the current dyad, then the value defaults to the *nopastEvents* argument.

The formula for recency for event e_i with type set to "inv.diff.plus1" and *i_neighborhood* is FALSE (Meijerink-Bosman et al. 2022):

$$recency_{e_i} = \frac{1}{t_{e_i} - t^* + 1}$$

where t^*, is the most recent time in which the past event's sender is the current event receiver and the past event receiver is the current event sender. If there are no past events within the current dyad, then the value defaults to the *nopastEvents* argument.

The formula for recency for event e_i with type set to "rank.ordered.count" and *i_neighborhood* is TRUE (Butts 2008):

$$recency_{e_i} = \rho(s(e_i), r(e_i), A_t)^{-1}$$

where $\rho(s(e_i), r(e_i), A_t)$, is the current event receiver's rank amongst the current sender's recent relational events. That is, as Butts (2008: 174) argues, "$\rho(s(e_i), r(e_i), A_t)$ is j's recency rank among

i's in-neighborhood. Thus, if j is the last person to have called i, then $\rho(s(e_i), r(e_i), A_t)^{-1} = 1$. This falls to 1/2 if j is the second most recent person to call i, 1/3 if j is the third most recent person, etc." Moreover, if j is not in i's neighborhood, the value defaults to infinity. If there are no past events with the current sender, then the value defaults to the *nopastEvents* argument.

The formula for recency for event e_i with type set to "rank.ordered.count" and *i_neighborhood* is FALSE (Butts 2008):

$$recency_{e_i} = \rho(r(e_i), s(e_i), A_t)^{-1}$$

where $\rho(r(e_i), s(e_i), A_t)$, is the current event sender's rank amongst the current receiver's recent relational events. That is, this measure is the same as above where the dyadic pair is flipped for the past relational events. Moreover, if j is not in i's neighborhood, the value defaults to infinity. If there are no past events with the current sender, then the value defaults to the *nopastEvents* argument.

Finally, researchers interested in modeling temporal relevancy (see Quintane, Mood, Dunn, and Falzone 2022) can specify the relational time span, that is, length of time for which events are considered relationally relevant. This should be specified via the option *relationalTimeSpan* with *dependency* set to TRUE.

Value

The vector of recency network statistics for the relational event sequence or the updated `data` argument.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Butts, Carter T. 2008. "A relational event framework for social action." *Sociological Methodology* 38(1): 155-200.

Meijerink-Bosman, Marlyne, Roger Leenders, and Joris Mulder. 2022. "Dynamic relational event modeling: Testing, exploring, and applying." *PLOS One* 17(8): e0272309.

Quintane, Eric, Martin Wood, John Dunn, and Lucia Falzon. 2022. "Temporal Brokering: A Measure of Brokerage as a Behavioral Process." *Organizational Research Methods* 25(3): 459-489.

Vu, Duy, Philippa Pattison, and Garry Robbins. 2015. "Relational event models for social learning in MOOCs." *Social Networks* 43: 121-135.

Examples

```
# A Dummy One-Mode Event Dataset
events <- data.frame(time = 1:18,
                                eventID = 1:18,
                                sender = c("A", "B", "C",
                                           "A", "D", "E",
                                           "F", "B", "A",
                                           "F", "D", "B",
                                           "G", "B", "D",
                                           "H", "A", "D"),
```

```
                                target = c("B", "C", "D",
                                           "E", "A", "F",
                                           "D", "A", "C",
                                           "G", "B", "C",
                                           "H", "J", "A",
                                           "F", "C", "B"))

# Creating the Post-Processing Event Dataset with Null Events
post.processing <- create_res(type = "one-mode",
                              riskset = "fixed",
                      ordinal = TRUE,
                      time = events$time,
                      sender = events$sender,
                      receiver = events$target,
                      p_samplingobserved = 1.00,
                      case_control=TRUE,
                      n_controls = 1,
                      seed = 9999)

#Computing the recency statistics (with raw time difference) for the relational event sequence
post.processing <- dreamstats_recency(data = post.processing,
                                      type = "raw.diff")

#printing the post-processed relational event sequence
post.processing

#Computing the recency statistics (with raw time difference) for the relational event sequence
#and returning only the vector of computed values
recency.stat <- dreamstats_recency(data = post.processing,
                                   type = "raw.diff",
                                   return_stats = TRUE)

cor(recency.stat, post.processing$statistics$recency)

#Computing the recency statistics (with inverse of time difference) for the
#relational event sequence
post.processing <- dreamstats_recency(data = post.processing,
                                      type = "inv.diff.plus1")

#Computing the rank-based recency statistics for the relational event sequence
post.processing <- dreamstats_recency(data = post.processing,
                                      type = "rank.ordered.count")
```

---

dreamstats_reciprocity

Compute the Reciprocity Network Statistic for Event Dyads in a Relational Event Sequence

---

Description

[Stable]

This function calculates the reciprocity network sufficient statistic for a relational event sequence (see Lerner and Lomi 2020; Butts 2008). The reciprocity statistic captures the tendency for a sender a to 'send a tie' to (e.g., initiate a communication event with) receiver b given that b sent a tie to a in the past (i.e., an exchange between two medical doctors). This function allows for reciprocity scores to be only computed for the sampled events, while creating the weights based on the full event sequence (see Lerner and Lomi 2020; Vu et al. 2015). The function also allows users to use two different weighting functions, return the counts of past events, reduce computational runtime, and specify a dyadic cutoff for relational relevancy.

Usage

```
dreamstats_reciprocity(
  data,
  halflife = 2,
  counts = FALSE,
  dyadic_weight = 0,
  exp_weight_form = FALSE,
  return_stats = FALSE
)
```

Arguments

`data`	An object of class `dream_sequence` that contains the processed relational event sequence.
`halflife`	A numerical value that is the halflife value to be used in the exponential weighting function (see details section). Preset to 2 (should be updated by the user based on substantive context).
`counts`	TRUE/FALSE. TRUE indicates that the counts of past events should be computed (see the details section). FALSE indicates that the temporal exponential weighting function should be used to downweigh past events (see the details section). Set to FALSE by default.
`dyadic_weight`	A numerical value for the dyadic cutoff weight that represents the numerical cutoff value for temporal relevancy based on the exponential weighting function. For example, a numerical value of 0.01, indicates that an exponential weight less than 0.01 will become 0 and that events with such value (or smaller values) will not be included in the sum of the past event weights (see the details section). Set to 0 by default.
`exp_weight_form`	TRUE/FALSE. TRUE indicates that the Lerner et al. (2013) exponential weighting function will be used (see the details section). FALSE indicates that the Lerner and Lomi (2020) exponential weighting function will be used (see the details section). Set to FALSE by default
`return_stats`	TRUE/FALSE. TRUE indicates that the vector of computed statistics will be returned. FALSE indicates that the vector of computed statistics will be added to the `statistics` element of the `data` argument.

Details

This function calculates reciprocity scores for relational event models based on the exponential weighting function used in either Lerner and Lomi (2020) or Lerner et al. (2013).

Following Lerner and Lomi (2020), the exponential weighting function in relational event models is:

$$w(s,r,t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}}$$

Following Lerner et al. (2013), the exponential weighting function in relational event models is:

$$w(s,r,t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}} \cdot \frac{ln(2)}{T_{1/2}}$$

In both of the above equations, *s* is the current event sender, *r* is the current event receiver (target), *t* is the current event time, *t'* is the past event times that meet the weight subset, and $T_{1/2}$ is the halflife parameter.

The formula for reciprocity for event e_i is:

$$reciprocity_{e_i} = w(r,s,t)$$

That is, all past events in which the past sender is the current receiver and the past receiver is the current sender.

Moreover, researchers interested in modeling temporal relevancy (see Quintane, Mood, Dunn, and Falzone 2022; Lerner and Lomi 2020) can specify the dyadic weight cutoff, that is, the minimum value for which the weight is considered relationally relevant. Users who do not know the specific dyadic cutoff value to use, can use the `dreamstats_dyadcut` function.

Following Butts (2008), if the counts of the past events are requested, the formula for reciprocity for event e_i is:

$$reciprocity_{e_i} = d(r = s', s = r', t')$$

Where, $d()$ is the number of past events where the event sender, *s'*, is the current event receiver, *r*, and the event receiver (target), *r'*, is the current event sender, *s*. Moreover, the counting equation can be used in tandem with relational relevancy, by specifying the halflife parameter, exponential weighting function, and the dyadic cut off weight values. If the user is not interested in modeling relational relevancy, then those value should be left at their baseline values.

Value

The vector of reciprocity statistics for the relational event sequence or the updated `data` argument.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Butts, Carter T. 2008. "A Relational Event Framework for Social Action." *Sociological Methodology* 38(1): 155-200.

Quintane, Eric, Martin Wood, John Dunn, and Lucia Falzon. 2022. "Temporal Brokering: A Measure of Brokerage as a Behavioral Process." *Organizational Research Methods* 25(3): 459-489.

Lerner, Jürgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: How to fit a relational event model to 360 million dyadic events." *Network Science* 8(1): 97-135.

Lerner, Jürgen, Margit Bussman, Tom A.B. Snijders, and Ulrik Brandes. 2013. " Modeling Frequency and Type of Interaction in Event Networks." *The Corvinus Journal of Sociology and Social Policy* 4(1): 3-32.

Vu, Duy, Philippa Pattison, and Garry Robins. 2015. "Relational event models for social learning in MOOCs." *Social Networks* 43: 121-135.

Examples

```
# A Psuedo One-Mode Event Dataset
events <- data.frame(time = 1:18,
                               eventID = 1:18,
                               sender = c("A", "B", "C",
                                          "A", "D", "E",
                                          "F", "B", "A",
                                          "F", "D", "B",
                                          "G", "B", "D",
                                         "H", "A", "D"),
                              target = c("B", "C", "D",
                                          "E", "A", "F",
                                          "D", "A", "C",
                                          "G", "B", "C",
                                          "H", "J", "A",
                                          "F", "C", "B"))

post.processing <- create_res(type = "one-mode",
                      ordinal = TRUE,
                      riskset = "fixed",
                      time = events$time,
                      sender = events$sender,
                      receiver = events$target,
                      p_samplingobserved = 1.00,
                         case_control=TRUE,
                      n_controls = 1,
                      seed = 9999)

#Computing the reciprocity statistic for the relational event sequence
post.processing <- dreamstats_reciprocity(data = post.processing,
                                          halflife = 2)

#printing the post-processed relational event sequence
post.processing
```

```
#Computing the reciprocity statistic for the relational event sequence
#and returning only the vector of computed values
reciprocity.stat <-  dreamstats_reciprocity(data = post.processing,
                                            halflife = 2,
                                            return_stats = TRUE)

cor(reciprocity.stat, post.processing$statistics$reciprocity)
```

`dreamstats_repetition` *Compute Butts' (2008) Repetition Network Statistic for Event Dyads in a Relational Event Sequence*

Description

[Stable]

This function computes the repetition network sufficient statistic for a relational event sequence (see Lerner and Lomi 2020; Butts 2008). Repetition measures the increased tendency for events between S and R to occur given that S and R have interacted in the past. Furthermore, this function allows for repetition scores to be only computed for the sampled events, while creating the weights based on the full event sequence (see Lerner and Lomi 2020; Vu et al. 2015). The function also allows users to use two different weighting functions, return the counts of past events, reduce computational runtime, and specify a dyadic cutoff for relational relevancy.

Usage

```
dreamstats_repetition(
  data,
  halflife = 2,
  counts = FALSE,
  dyadic_weight = 0,
  exp_weight_form = FALSE,
  return_stats = FALSE
)
```

Arguments

`data`	A `dream_sequence` object that contains the processed relational event sequence.
`halflife`	A numerical value that is the halflife value to be used in the exponential weighting function (see details section). Preset to 2 (should be updated by the user based on substantive context).
`counts`	TRUE/FALSE. TRUE indicates that the counts of past events should be computed (see the details section). FALSE indicates that the temporal exponential weighting function should be used to downweigh past events (see the details section). Set to FALSE by default.

`dyadic_weight` A numerical value for the dyadic cutoff weight that represents the numerical cutoff value for temporal relevancy based on the exponential weighting function. For example, a numerical value of 0.01, indicates that an exponential weight less than 0.01 will become 0 and that events with such value (or smaller values) will not be included in the sum of the past event weights (see the details section). Set to 0 by default.

`exp_weight_form` TRUE/FALSE. TRUE indicates that the Lerner et al. (2013) exponential weighting function will be used (see the details section). FALSE indicates that the Lerner and Lomi (2020) exponential weighting function will be used (see the details section). Set to FALSE by default

`return_stats` TRUE/FALSE. TRUE indicates that the vector of computed statistics will be returned. FALSE indicates that the vector of computed statistics will be added to the `statistics` element of the `data` argument.

Details

This function calculates the repetition scores for relational event models based on the exponential weighting function used in either Lerner and Lomi (2020) or Lerner et al. (2013).

Following Lerner and Lomi (2020), the exponential weighting function in relational event models is:

$$w(s,r,t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}}$$

Following Lerner et al. (2013), the exponential weighting function in relational event models is:

$$w(s,r,t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}} \cdot \frac{ln(2)}{T_{1/2}}$$

In both of the above equations, *s* is the current event sender, *r* is the current event receiver (target), *t* is the current event time, *t'* is the past event times that meet the weight subset (in this case, all events that have the same sender and receiver), and $T_{1/2}$ is the halflife parameter.

The formula for repetition for event e_i is:

$$repetition_{e_i} = w(s,r,t)$$

Moreover, researchers interested in modeling temporal relevancy (see Quintane, Mood, Dunn, and Falzone 2022; Lerner and Lomi 2020) can specify the dyadic weight cutoff, that is, the minimum value for which the weight is considered relationally relevant. Users who do not know the specific dyadic cutoff value to use, can use the `dreamstats_dyadcut` function.

Following Butts (2008), if the counts of the past events are requested, the formula for repetition for event e_i is:

$$repetition_{e_i} = d(s = s', r = r', t')$$

Where, $d()$ is the number of past events where the event sender, *s'*, is the current event sender, *s*, the event receiver (target), *r'*, is the current event receiver, *r*. Moreover, the counting equation can be used in tandem with relational relevancy, by specifying the halflife parameter, exponential weighting function, and the dyadic cut off weight values. If the user is not interested in modeling relational relevancy, then those value should be left at their baseline values.

Value

The vector of repetition statistics for the relational event sequence or the updated `data` argument.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Butts, Carter T. 2008. "A Relational Event Framework for Social Action." *Sociological Methodology* 38(1): 155-200.

Quintane, Eric, Martin Wood, John Dunn, and Lucia Falzon. 2022. "Temporal Brokering: A Measure of Brokerage as a Behavioral Process." *Organizational Research Methods* 25(3): 459-489.

Lerner, Jürgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: How to fit a relational event model to 360 million dyadic events." *Network Science* 8(1): 97-135.

Lerner, Jürgen, Margit Bussman, Tom A.B. Snijders, and Ulrik Brandes. 2013. " Modeling Frequency and Type of Interaction in Event Networks." *The Corvinus Journal of Sociology and Social Policy* 4(1): 3-32.

Vu, Duy, Philippa Pattison, and Garry Robins. 2015. "Relational event models for social learning in MOOCs." *Social Networks* 43: 121-135.

Examples

```
data("WikiEvent2018.first100k", package = "dream")
WikiEvent2018 <- WikiEvent2018.first100k[1:1000,] #the first one thousand events
WikiEvent2018$time <- as.numeric(WikiEvent2018$time) #making the variable numeric
### Creating the EventSet By Employing Case-Control Sampling With M = 5 and
### Sampling from the Observed Event Sequence with P = 0.01
post.processing <- create_res(type = "two-mode",
 ordinal = TRUE,
 riskset = "fixed",
 time = WikiEvent2018$time, # The Time Variable
 sender = WikiEvent2018$user, # The Sender Variable
 receiver = WikiEvent2018$article, # The Receiver Variable
 p_samplingobserved = 0.01, # The Probability of Selection
    case_control=TRUE,
 n_controls = 8, # The Number of Controls to Sample from the Full Risk Set
 seed = 9999)


#Computing the repetition statistics for the relational event sequence with
#the exponential weights of past events returned
post.processing <- dreamstats_repetition(data = post.processing,
                                         halflife = 2.592e+09)

#printing the post-processed relational event sequence
post.processing
```

```
#Computing the repetition statistic for the relational event sequence
#and returning only the vector of computed values
repetition.stat <-  dreamstats_repetition(data = post.processing,
                                          halflife = 2.592e+09,
                                          return_stats = TRUE)

cor(repetition.stat, post.processing$statistics$repetition)

#Computing the repetition statistics for the relational event sequence with
#the counts of past events returned
post.processing <- dreamstats_repetition(data = post.processing,
                                         halflife = 2.592e+09,
                                         counts = TRUE)

cbind(post.processing$statistics$repetition, repetition.stat)
```

`dreamstats_triads` *Compute Butts' (2008) Triadic Formation Statistics for Relational Event Sequences*

Description

[Stable]

The function computes the set of one-mode triadic formation statistics discussed in Butts (2008) for a one-mode relational event sequence (see also Lerner and Lomi 2020). The function can compute the following triadic formations: 1) incoming shared partners (ISP), 2) outgoing shared partners (OSP), 3) incoming two-paths (ITP), and 4) outgoing two-paths (OTP). Importantly, this function allows for the triadic formation statistics to be computed only for the sampled events, while creating the weights based on the full event sequence (see Lerner and Lomi 2020; Vu et al. 2015). The function also allows users to use two different weighting functions, return the counts of past events, reduce computational runtime, and specify a dyadic cutoff for relational relevancy.

Usage

```
dreamstats_triads(
  formation = c("ISP", "OSP", "ITP", "OTP"),
  data,
  halflife = 2,
  counts = FALSE,
  dyadic_weight = 0,
  exp_weight_form = FALSE,
  return_stats = FALSE
)
```

Arguments

formation	The specific triadic formation the statistic will be based on (see details section). "ISP" = incoming shared partners. "OSP" = outgoing shared partners. "OTP" = outgoing two-paths. "ITP" = incoming two-paths.
data	A dream_sequence object that contains the processed relational event sequence.
halflife	A numerical value that is the halflife value to be used in the exponential weighting function (see details section). Preset to 2 (should be updated by the user based on substantive context).
counts	TRUE/FALSE. TRUE indicates that the counts of past events should be computed (see the details section). FALSE indicates that the temporal exponential weighting function should be used to downweigh past events (see the details section). Set to FALSE by default.
dyadic_weight	A numerical value for the dyadic cutoff weight that represents the numerical cutoff value for temporal relevancy based on the exponential weighting function. For example, a numerical value of 0.01, indicates that an exponential weight less than 0.01 will become 0 and that events with such value (or smaller values) will not be included in the sum of the past event weights (see the details section). Set to 0 by default.
exp_weight_form	TRUE/FALSE. TRUE indicates that the Lerner et al. (2013) exponential weighting function will be used (see the details section). FALSE indicates that the Lerner and Lomi (2020) exponential weighting function will be used (see the details section). Set to FALSE by default
return_stats	TRUE/FALSE. TRUE indicates that the vector of computed statistics will be returned. FALSE indicates that the vector of computed statistics will be added to the statistics element of the data argument.

Details

The function calculates the triadic formation statistics discussed in Butts (2008) for relational event sequences based on the exponential weighting function used in either Lerner and Lomi (2020) or Lerner et al. (2013).

Following Lerner and Lomi (2020), the exponential weighting function in relational event models is:

$$w(s,r,t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}}$$

Following Lerner et al. (2013), the exponential weighting function in relational event models is:

$$w(s,r,t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}} \cdot \frac{ln(2)}{T_{1/2}}$$

In both of the above equations, *s* is the current event sender, *r* is the current event receiver (target), *t* is the current event time, *t'* is the past event times that meet the weight subset, and $T_{1/2}$ is the halflife parameter.

Outgoing Shared Partners:

The general formula for outgoing shared partners for event e_i is:

$$OSP_{e_i} = \sqrt{\sum_h w(s,h,t) \cdot w(r,h,t)}$$

That is, as discussed in Butts (2008), outgoing shared partners finds all past events where the current sender and target sent a relational tie (i.e., were a sender in a relational event) to the same *h* node.

Following Butts (2008), if the counts of the past events are requested, the formula for outgoing shared partners for event e_i is:

$$OSPe_i = \sum_{i=1}^{|H|} \min\left[d(s,h,t), d(s,h,t)\right]$$

Where, $d()$ is the number of past events that meet the specific set operations. $d(s,h,t)$ is the number of past events where the current event sender sent a tie to a third actor, *h*, and $d(r,h,t)$ is the number of past events where the current event receiver sent a tie to a third actor, *h*. The sum loops through all unique actors that have formed past outgoing shared partners structures with the current event sender and receiver. Moreover, the counting equation can be used in tandem with relational relevancy, by specifying the halflife parameter, exponential weighting function, and the dyadic cut off weight values. If the user is not interested in modeling relational relevancy, then those value should be left at their defaults.

Outgoing Two-Paths:

The general formula for outgoing two-paths for event e_i is:

$$OTP_{e_i} = \sqrt{\sum_h w(s,h,t) \cdot w(h,r,t)}$$

That is, as discussed in Butts (2008), outgoing two-paths finds all past events where the current sender sends a relational tie to node *h* and the current target receives a relational tie from the same *h* node.

Following Butts (2008), if the counts of the past events are requested, the formula for outgoing two paths for event e_i is:

$$OTP_{e_i} = \sum_{i=1}^{|H|} \min\left[d(s,h,t), d(h,r,t)\right]$$

Where, $d()$ is the number of past events that meet the specific set operations. $d(s,h,t)$ is the number of past events where the current event sender sent a tie to a third actor, *h*, and $d(h,r,t)$ is the number of past events where the third actor *h* sent a tie to the current event receiver. The sum loops through all unique actors that have formed past outgoing two-path structures with the current event sender and receiver.

Incoming Two-Paths:

The general formula for incoming two-paths for event e_i is:

$$ITP_{e_i} = \sqrt{\sum_h w(r,h,t) \cdot w(h,s,t)}$$

That is, as discussed in Butts (2008), incoming two-paths finds all past events where the current sender was the receiver in a relational event where the sender was a node h and the current target was the sender in a past relational event where the target was the same node h.

Following Butts (2008), if the counts of the past events are requested, the formula for incoming two paths for event e_i is:

$$ITP_{e_i} = \sum_{i=1}^{|H|} \min\left[d(r,h,t), d(h,s,t\right]$$

Where, $d()$ is the number of past events that meet the specific set operations. $d(r,h,t)$ is the number of past events where the current event receiver sent a tie to a third actor, *h*, and $d(h,s,t$ is the number of past events where the third actor *h* sent a tie to the current event sender. The sum loops through all unique actors that have formed past incoming two-path structures with the current event sender and receiver.

Incoming Shared Partners:

The general formula for incoming shared partners for event e_i is:

$$ISP_{e_i} = \sqrt{\sum_{h} w(h,s,t) \cdot w(h,r,t)}$$

That is, as discussed in Butts (2008), incoming shared partners finds all past events where the current sender and target were themselves the target in a relational event from the same *h* node.

Following Butts (2008), if the counts of the past events are requested, the formula for incoming shared partners for event e_i is:

$$ISP_{e_i} = \sum_{i=1}^{|H|} \min\left[d(h,s,t), d(h,r,t)\right]$$

Where, $d()$ is the number of past events that meet the specific set operations, $d(h,s,t)$ is the number of past events where the current event sender received a tie from a third actor, *h*, and $d(h,r,t)$ is the number of past events where the current event receiver received a tie from a third actor, *h*. The sum loops through all unique actors that have formed past incoming shared partners structures with the current event sender and receiver.

Lastly, researchers interested in modeling temporal relevancy (see Quintane, Mood, Dunn, and Falzone 2022; Lerner and Lomi 2020) can specify the dyadic weight cutoff, that is, the minimum value for which the weight is considered relationally relevant. Users who do not know the specific dyadic cutoff value to use, can use the `dreamstats_dyadcut` function.

Value

The vector of triadic formation statistics for the relational event sequence or the updated `data` argument.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Butts, Carter T. 2008. "A Relational Event Framework for Social Action." *Sociological Methodology* 38(1): 155-200.

Quintane, Eric, Martin Wood, John Dunn, and Lucia Falzon. 2022. "Temporal Brokering: A Measure of Brokerage as a Behavioral Process." *Organizational Research Methods* 25(3): 459-489.

Lerner, Jürgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: How to fit a relational event model to 360 million dyadic events." *Network Science* 8(1): 97-135.

Lerner, Jürgen, Margit Bussman, Tom A.B. Snijders, and Ulrik Brandes. 2013. " Modeling Frequency and Type of Interaction in Event Networks." *The Corvinus Journal of Sociology and Social Policy* 4(1): 3-32.

Vu, Duy, Philippa Pattison, and Garry Robins. 2015. "Relational event models for social learning in MOOCs." *Social Networks* 43: 121-135.

Examples

```
events <- data.frame(time = 1:18,
                                eventID = 1:18,
                                sender = c("A", "B", "C",
                                           "A", "D", "E",
                                           "F", "B", "A",
                                           "F", "D", "B",
                                           "G", "B", "D",
                                          "H", "A", "D"),
                               target = c("B", "C", "D",
                                          "E", "A", "F",
                                          "D", "A", "C",
                                          "G", "B", "C",
                                          "H", "J", "A",
                                          "F", "C", "B"))

eventSet <- create_res(type = "one-mode",
                       ordinal = TRUE,
                       riskset = "fixed",
                       time = events$time,
                       sender = events$sender,
                       receiver = events$target,
                       p_samplingobserved = 1.00,
                       case_control=TRUE,
                       n_controls = 1,
                       seed = 9999)

#compute the triadic statistic for the outgoing shared partners formation
eventSet <- dreamstats_triads(formation = "OSP",
                              data = eventSet,
                              halflife = 2, #halflife parameter
                              dyadic_weight = 0)
#printing the post-processed relational event sequence
eventSet
#printing the vector of computed values
```

```
eventSet$statistics$outgoing.shared.partners


#Computing theoutgoing shared partners statistic for the relational event sequence
#and returning only the vector of computed values
osp.stat <- dreamstats_triads(formation = "OSP",
                              data = eventSet,
                              halflife = 2, #halflife parameter
                              dyadic_weight = 0,
                              return_stats = TRUE)

cor(osp.stat, eventSet$statistics$outgoing.shared.partners)

#compute the triadic statistic for the incoming shared partners formation
eventSet <- dreamstats_triads(
   formation = "ISP",
   data = eventSet,
   halflife = 2, #halflife parameter
   dyadic_weight = 0)

#compute the triadic statistic for the outgoing two-paths formation
eventSet <- dreamstats_triads(
   formation = "OTP",
   data = eventSet,
   halflife = 2, #halflife parameter
   dyadic_weight = 0)

#compute the triadic statistic for the incoming two-paths formation
eventSet <- dreamstats_triads(
   formation = "ITP",
   data = eventSet,
   halflife = 2, #halflife parameter
   dyadic_weight = 0)

#extracting the relational event information
triad.rems <- as.data.frame(eventSet)
triad.rems
```

`estimate_rem` *Fit a Maximum Likelihood Relational Event Model (REM) to A Processed Relational Event Sequence*

Description

[Stable]

This function estimates the ordinal and interval timing relational event model by maximizing the likelihood function given by Butts (2008) via maximum likelihood estimation. A nice outcome is that the ordinal timing relational event model is equivalent to the conditional logistic regression (see Greene 2003; for R functions, see `clogit`). In addition, based on this outcome and the structure

of the data, this function can estimate the Cox proportional hazards model (see Box-Steffensmeier and Jones 2004; for R functions, see coxph) given that the likelihood functions are equivalent. An important assumption this model makes is that only one event occurs at each time point. If this is unfeasible for the user's specific dataset, we encourage the user to see the clogit function for the Breslow approximation technique (Box-Steffensmeier and Jones 2004). Future versions of the package will include options for tied event data (e.g., multiple events at one time point).

Usage

```
estimate_rem(
  formula,
  data,
  newton.rhapson = TRUE,
  optim.method = "BFGS",
  optim.control = list(),
  tolerance = 1e-09,
  maxit = 100,
  starting.beta = NULL,
  multiple.events = FALSE,
  ...
)
```

Arguments

formula	A formula object where the covariates are on the right hand side of ~. The names of the covariates must follow the names in the statistics element of the data argument, since the right hand side covariates are taken from this list. The dependent variable does not need to be defined, since it is internally defined based upon the data argument. This is the same argument found in lm and glm.
data	An object of class dream_sequence that contains the processed relational event sequence and statistics that are included in the formula argument.
newton.rhapson	TRUE/FALSE. TRUE indicates an internal Newton-Rhapson iteration procedure with line searching is used to find the set of maximum likelihood estimates. FALSE indicates that the log likelihood function will be optimized via the optim function. The function defaults to TRUE.
optim.method	If newton.rhapson is FALSE, what optim method should be used in conjunction with the optim function. Defaults to "BFGS". See the optim function for the set of options.
optim.control	If newton.rhapson is FALSE, a list of control to be used in the optim function. See the optim function for the set of controls.
tolerance	If newton.rhapson is TRUE, the stopping criterion for the absolute difference in the log likelihoods for each Newton-Rhapson iteration. The optimization procedure stops when the absolute change in the log likelihoods is less than tolerance (see Greene 2003).
maxit	If newton.rhapson is TRUE, the maximum number of iterations for the Newton-Rhapson optimization procedure (see Greene 2003).

`starting.beta` A numeric vector that represents the starting parameter estimates for the Newton-Rhapson optimization procedure. This may be a beneficial argument if the optimization procedure fails, since the Newton-Rhapson optimization procedure is sensitive to starting values. Preset to NULL.

`multiple.events` TRUE/FALSE. Currently, this function assumes that only one event occurs per event cluster (i.e., time point). Future versions of the package will include estimation options for multiple events per time point, commonly referred to as tied events, via the Breslow approximation technique (see Box-Steffensmeier and Jones 2004). At this moment, this argument is preset to FALSE and should not be modified by the user.

`...` Additional arguments.

Details

This function estimates the ordinal and interval timing relational event model by maximizing the likelihood function provided in the seminal REM paper by Butts (2008) via maximum likelihood estimation. The ordinal timing likelihood function is:

$$L(A_t|\beta) = \prod_{i=1}^{|A_t|} \frac{\lambda_{a_i}}{\sum_{a' \in M_t} \lambda_{a'}}$$

where, following Butts (2008) and Duxbury (2020), A_t is the relational event sequence, λ_{a_i} is the hazard rate for event *i*, which is formulated to be equal to $exp(\beta^T z(x, Y))$, that is, the linear combination of user-specific covariates, $z(x, Y)$, and associated REM parameters, β. The user provides these covariates via the `formula` argument. M_t is the support set for event $a_i \in A_t$. The likelihood function for the interval timing relational event model is:

$$L(A_t|\beta) = [\prod_{i=1}^{|A_t|} \lambda_i \prod_{j \in M_{\tau(i)}} \exp(-\lambda_j \{\tau(i) - \tau(i-1)\})] \times [\prod_{j \in M_t} \exp(-\lambda_j \{t - \tau(M)\})]$$

where $\tau(i)$ is the time of the observed (realized) event *i* and t is the time that marks the end of the relational event sequence. Following Duxbury (2020), $z(x, Y)$ is a mapping function that represents the endogenous network statistics computed on the network of past events,x, and exogenous covariates, Y. In comparison to the ordinal timing relational event formulation, the hazard rate for event *i*, λ_{a_i}, includes the baseline hazard rate (the intercept), $exp(\beta_0 + \beta^T z(x, Y))$. If t is not known by the user, then the interval timing likelihood is:

$$L(A_t|\beta) = \prod_{i=1}^{|A_t|} \lambda_i \prod_{j \in M_{\tau(i)}} \exp(-\lambda_j \{\tau(i) - \tau(i-1)\})$$

In this case, the likelihood function is the same as employed in the `remstimate` for interval timing relational event models. The values for t are taken from the `data` object.

This function provides two numerical optimization techniques to find the maximum likelihood estimates for the associated parameters. First, this function allows the user to use the `optim` function to find the associated parameters based on the above likelihood function. Secondly, and by default, this function employs a Newton-Rhapson iteration algorithm with line-searching to find the unknown

parameters (see Greene 2003 for a discussion of this algorithm). If desired, the user can provide the initial searching values for both algorithms with the `starting.beta` argument.

It's important to note that the modeling concerns of the conditional logistic regression apply to the ordinal timing relational event model, such as no within-sequence fixed effects, that is, a variable that does not vary within event cluster (i.e., a variable that is the same for both the null and observed events). The function internally checks for this and provides the user with a warning if any requested effects has no total within-event variance. Moreover, any observed events that have no associated control events are removed from the analysis as they provide no information to the log likelihood (see Greene 2003). The function removes these events from the sequence prior to estimation.

Value

An object of class "dream_rem" as a list containing the following components:

- `optimization.method` - The optimzation method used to find the parameters.
- `converged` - TRUE/FALSE. TRUE indicates that the REM converged.
- `loglikelihood.null` - The log likelihood of the null model (i.e., the model where the parameters are assumed to be 0).
- `loglikelihood.full` - The log likelihood of the estimated model.
- `chi.stat` - The chi-statistic of the likelihood ratio test.
- `loglikelihood.test` - The p-value of the likelihood ratio test.
- `df.null` - The degrees of freedom of the null model.
- `df.full` - The degrees of freedom of the full model.
- `parameters` - The MLE parameter estimates.
- `hessian` - The estimated hessian matrix.
- `gradient` - The estimated gradient vector.
- `se.parameter` - The standard errors of the MLE parameter estimates.
- `covariance.mat` - The estimated variance-covariance matrix.
- `z.values` - The z-scores for the MLE parameter estimates.
- `p.values` - The p-values for the MLE parameter estimates.
- `AIC` - The AIC of the estimated REM.
- `BIC` - The BIC of the estimated REM.
- `n.events` - The number of observed events in the relational event sequence.
- `null.events` - The number of control events in the relational event sequence.
- `newton.iterations` - The number of Newton-Rhapson iterations.
- `search.algo` - A data.frame object that contains the Newton-Rhapson searching algorithm results.

Author(s)

Kevin A. Carson kacarson@arizona.edu and Diego F. Leal dflc@arizona.edu

References

Box-Steffensmeier, Janet and Bradford S. Jones. 2004. *Event History Modeling: A Guide for Social Scientists*. Cambridge University Press.

Butts, Carter T. 2008. "A Relational Event Framework for Social Action." *Sociological Methodology* 38(1): 155-200.

Duxbury, Scott. 2020. *Longitudinal Network Models*. Sage University Press. Quantitative Applications in the Social Sciences: 192.

Greene, William H. 2003. *Econometric Analysis*. Fifth Edition. Prentice Hall Press.

See Also

predict.dream_rem, vcov.dream_rem, logLik.dream_rem, AIC, gof_rem, residuals.dream_rem,plot.dream_rem,coef

Examples

```
#Creating a psuedo one-mode relational event sequence with ordinal timing
relational.seq <- simulate_rem_seq(n_actors = 8,
                                   n_events = 50,
                                   inertia = TRUE,
                                   inertia_p = 0.10,
                                   sender_outdegree = TRUE,
                                   sender_outdegree_p = 0.05)

#Creating a post-processing event sequence for the above relational sequence
post.processing <-  create_res(type = "one-mode",
                               ordinal = TRUE,
                               riskset = "fixed",
                               time = relational.seq$eventID,
                               sender = as.character(relational.seq$sender),
                               receiver = as.character(relational.seq$target))

#Computing the sender-outdegree statistic for the above post-processing
#one-mode relational event sequence
post.processing <- dreamstats_degree(formation = "sender-outdegree",
                                   data = post.processing,
                                   halflife = 2)

#Computing the inertia/repetition statistic for the above post-processing
#one-mode relational event sequence
post.processing <- dreamstats_repetition(data = post.processing,
                                         halflife = 2)

#Fitting an ordinal timing relational event model to the above one-mode relational
#event sequence
rem <- estimate_rem(~ sender.outdegree + repetition,
                          data=post.processing)
summary(rem) #summary of the relational event model

vcov(rem) #printing the variance-covariance matrix
logLik(rem) #printing the model log-likelihood
```

```
AIC(rem) #printing the model AIC
rates <- predict(rem) #extracting the predicted event rates


#Fitting a (ordinal) relational event model to the above one-mode relational
#event sequence via the optim function
rem1 <- estimate_rem(~ sender.outdegree + repetition,
                  data=post.processing,
                  newton.rhapson=FALSE)
summary(rem1) #summary of the relational event model


# a psuedo relational event sequence
events <- data.frame(time = 1:18, eventID = 1:18,
                                sender = c("A", "B", "C",
                                           "A", "D", "E",
                                           "F", "B", "A",
                                           "F", "D", "B",
                                           "G", "B", "D",
                                          "H", "A", "D"),
                               target = c("B", "C", "D",
                                          "E", "A", "F",
                                          "D", "A", "C",
                                          "G", "B", "C",
                                          "H", "J", "A",
                                          "F", "C", "B"))

# Creating a dynamic one-mode relational risk set with p = 1.00 (all true events)
# and 5 controls based upon the interval timing relational event framework
eventSet <- create_res(ordinal = FALSE,
                       t = max(events$time) + rexp(1),
                       riskset = "cumulative",
                       type = "one-mode",
                       time = events$time,
                       sender = events$sender,
                       receiver = events$target,
                       p_samplingobserved = 1.00,
                       case_control=TRUE,
                       n_controls = 5,
                       seed = 9999)

#Computing the sender indegree statistic for the relational event sequence
eventSet <- dreamstats_degree(formation = "sender-indegree",
                              data = eventSet,
                              halflife = 2)

#Computing the outgoing two path statistic for the relational event sequence
eventSet <- dreamstats_triads(formation = "OTP",
                              data = eventSet,
                              halflife = 2)

#Fitting an interval timing relational event model to the above one-mode relational
#event sequence
```

```
rem.interval <- estimate_rem(~ sender.indegree + outgoing.two.paths,
                             data=eventSet)
summary(rem.interval) #summary of the relational event model

rem.interval.optim <- estimate_rem(~ sender.indegree + outgoing.two.paths,
                                   data=eventSet,
                                   newton.rhapson=FALSE)
summary(rem.interval.optim) #summary of the relational event model
```

`gof_rem` *Estimate the proportion of dyads predicted by* `dream_rem` *relational event model fits*

Description

This function returns the proportion of dyads, senders, and targets predicted by a `dream_rem` relational event model object. To compute the proportion for each category, the function finds, for each realized event time, the dyad (based upon the user's provided risk set) that is most likely to occur (i.e., the dyad with the largest hazard rate) (Butts 2008). The predicted dyad is then compared to the realized event dyad.

Usage

```
gof_rem(object, realized = TRUE, rseed = NULL, ...)
```

Arguments

`object`	An object of class "dream_rem".
`realized`	How should cases be handled when the largest hazard for a specific event is the same for the realized event and one or more control events? TRUE indicates that the method will default to the realized event. FALSE indicates that the function will take a sample from the set of realized and control events that have the largest event hazard.
`rseed`	The random seed for reproducibility of results. When more than 1 dyad has the maximum hazard rate, the predicted dyad is selected at random with equal probabilty for each dyad. For example, if three dyads have the same hazard value, then the probability of selecting each dyad will be 1/3.
`...`	Additional arguments for other methods.

Details

To compute the proportion for each category, the function finds, for each realized event time, the dyad (based upon the user's provided risk set) that is most likely to occur (i.e., the dyad with the largest hazard rate) (Butts 2008). The predicted dyad is then compared to the realized event dyad.

Value

A dream_gof S3 object that contains the following results:

- props - A data.frame that contains the proportion of correctly predicted dyads, senders, and targets.
- edgelist - A data.frame object that contains the predicted and realized relational event sequence.
- estimated_rem - The dream_rem S3 object provided to the function.
- match_realized - The realized argument provided to the function.

References

Butts, Carter T. 2008. "A Relational Event Framework for Social Action." *Sociological Methodology* 38(1): 155-200. #'

Examples

```
#Creating a psuedo one-mode relational event sequence with ordinal timing
relational.seq <- simulate_rem_seq(n_actors = 8,
                                   n_events = 50,
                                   inertia = TRUE,
                                   inertia_p = 0.10,
                                   sender_outdegree = TRUE,
                                   sender_outdegree_p = 0.05)

#Creating a post-processing event sequence for the above relational sequence
post.processing <-  create_res(type = "one-mode",
                               ordinal = TRUE,
                                riskset = "fixed",
                               time = relational.seq$eventID,
                               sender = as.character(relational.seq$sender),
                               receiver = as.character(relational.seq$target),
                               case_control=TRUE,
                               n_controls = 5)

#Computing the sender-outdegree statistic for the above post-processing
#one-mode relational event sequence
post.processing <- dreamstats_degree(formation = "sender-outdegree",
                                   data = post.processing,
                                   halflife = 2)

#Computing the inertia/repetition statistic for the above post-processing
#one-mode relational event sequence
post.processing <- dreamstats_repetition(data = post.processing,
                                         halflife = 2)

#Fitting an ordinal timing relational event model to the above one-mode relational
#event sequence
rem <- estimate_rem(~ sender.outdegree + repetition,
                          data=post.processing)
summary(rem) #summary of the relational event model
```

```
#the model goodness of fit
gof <- gof_rem(rem,rseed=3718)
summary(gof) #printing the summary of the goodness of fit assessment
plot(gof) #plotting the goodness of fit assessment
gof$edgelist #check the predicted edgelist for the dyadic events
```

`logLik.dream_rem` *Extract the model log-likelihood from Relational Event Model Fits*

Description

This function extracts the model loglikelhood from estimated relational event model fits.

Usage

```
## S3 method for class 'dream_rem'
logLik(object, ..., REML = FALSE)
```

Arguments

`object`	An object of class "dream_rem".
`...`	Additional arguments for other methods.
`REML`	From the generic `logLik` function. Set to FALSE and does not need to changed by the user.

Examples

```
#Creating a psuedo one-mode relational event sequence with ordinal timing
relational.seq <- simulate_rem_seq(n_actors = 8,
                                   n_events = 50,
                                   inertia = TRUE,
                                   inertia_p = 0.10,
                                   sender_outdegree = TRUE,
                                   sender_outdegree_p = 0.05)

#Creating a post-processing event sequence for the above relational sequence
post.processing <-  create_res(type = "one-mode",
                               ordinal = TRUE,
                               riskset = "fixed",
                               time = relational.seq$eventID,
                               sender = as.character(relational.seq$sender),
                               receiver = as.character(relational.seq$target),
                               case_control=TRUE,
                               n_controls = 5)

#Computing the sender-outdegree statistic for the above post-processing
#one-mode relational event sequence
post.processing <- dreamstats_degree(formation = "sender-outdegree",
```

```
                                    data = post.processing,
                                    halflife = 2)

#Computing the inertia/repetition statistic for the above post-processing
#one-mode relational event sequence
post.processing <- dreamstats_repetition(data = post.processing,
                                         halflife = 2)

#Fitting an ordinal timing relational event model to the above one-mode relational
#event sequence
rem <- estimate_rem(~ sender.outdegree + repetition,
                          data=post.processing)
logLik(rem)
```

`netstats_om_constraint` *Compute Burt's (1992) Constraint for Ego Networks from a Sociomatrix*

Description

[Stable]

This function computes Burt's (1992) one-mode ego constraint based upon a sociomatrix.

Usage

```
netstats_om_constraint(
  net,
  inParallel = FALSE,
  nCores = NULL,
  isolates = NA,
  pendants = 1
)
```

Arguments

`net`	A one-mode sociomatrix with network ties.
`inParallel`	TRUE/FALSE. TRUE indicates that parallel processing will be used to compute the statistic with the *foreach* package. FALSE indicates that parallel processing will not be used. Set to FALSE by default.
`nCores`	If inParallel = TRUE, the number of computing cores for parallel processing. If this value is not specified, then the function internally provides it by dividing the number of available cores in half.
`isolates`	What value should isolates be given? Set to NA by default.
`pendants`	What value should be given to pendant vertices? Set to 1 by default. Pendant vertices are those nodes who have one outgoing tie.

Details

The formula for Burt's (1992) one-mode ego constraint is:

$$c_{ij} = \left(p_{ij} + \sum_{q} p_{iq} p_{qj} \right)^2 \quad ; \; q \neq i \neq j$$

where:

- p_{iq} is formulated as: $p_{iq} = \frac{z_{iq} + z_{qi}}{\sum_j (z_{ij} + z_{ji})} \quad ; \; i \neq j$

Finally, the aggregate constraint of an ego *i* is:

$$C_i = \sum_{j} c_{ij}$$

While this function internally locates isolates (i.e., nodes who have no ties) and pendants (i.e., nodes who only have one tie), the user should specify what values for constraint are returned for them via the *isolates* and *pendants* options. In particular, pendant vertices are those nodes who have one outgoing tie.

Lastly, this function allows users to compute the values in parallel via the *foreach*, *doParallel*, and *parallel* R packages.

Value

The vector of ego network constraint values.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Burt, Ronald. 1992. *Structural Holes: The Social Structure of Competition*. Harvard University Press.

Examples

```
# For this example, we recreate the ego network provided in Burt (1992: 56):
BurtEgoNet <- matrix(c(
  0,1,0,0,1,1,1,
 1,0,0,1,0,0,1,
 0,0,0,0,0,0,1,
 0,1,0,0,0,0,1,
 1,0,0,0,0,0,1,
 1,0,0,0,0,0,1,
 1,1,1,1,1,1,0),
 nrow = 7, ncol = 7)
colnames(BurtEgoNet) <- rownames(BurtEgoNet) <- c("A", "B", "C", "D", "E",
                                                  "F", "ego")
#the constraint value for the ego replicates that provided in Burt (1992: 56)
```

```
netstats_om_constraint(BurtEgoNet)
```

`netstats_om_effective` *Compute Burt's (1992) Effective Size for Ego Networks from a Sociomatrix*

Description

[Stable]

This function computes Burt's (1992) one-mode ego effective size based upon a sociomatrix (see details).

Usage

```
netstats_om_effective(
  net,
  inParallel = FALSE,
  nCores = NULL,
  isolates = NA,
  pendants = 1
)
```

Arguments

`net`	The one-mode sociomatrix with network ties.
`inParallel`	TRUE/FALSE. TRUE indicates that parallel processing will be used to compute the statistic with the *foreach* package. FALSE indicates that parallel processing will not be used. Set to FALSE by default.
`nCores`	If inParallel = TRUE, the number of computing cores for parallel processing. If this value is not specified, then the function internally provides it by dividing the number of available cores in half.
`isolates`	The numerical value that represents what value will isolates be given. Set to NA by default.
`pendants`	The numerical value that represents what value will pendant vertices be given. Set to 1 by default. Pendant vertices are those nodes who have one outgoing tie.

Details

The formula for Burt's (1992; see also Borgatti 1997) one-mode ego effective size is:

$$E_i = \sum_j 1 - \sum_q p_{iq} m_{jq}; q \neq i \neq j$$

where E_i is the ego effective size for an ego i. p_{iq} is formulated as:

$$\frac{(z_{iq} + z_{qi})}{\sum_j (z_{ij} + z_{ji})}; i \neq j$$

and m_{jq} is:

$$m_{jq} = \frac{(z_{jq} + z_{qj})}{max(z_{jk} + z_{kj})}$$

While this function internally locates isolates (i.e., nodes who have no ties) and pendants (i.e., nodes who only have one tie), the user should specify what values for constraint are returned for them via the *isolates* and *pendants* options. In particular, pendant vertices are those nodes who have one outgoing tie.

Value

The vector of ego network effective size values.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Burt, Ronald. 1992. *Structural Holes: The Social Structure of Competition*. Harvard University Press.

Borgatti, Stephen. 1997. "Structural Holes: Unpacking Burt's Redundancy Measures." *Connections* 20(1): 35-38.

Examples

```
# For this example, we recreate the ego network provided in Borgatti (1997):
BorgattiEgoNet <- matrix(
 c(0,1,0,0,0,0,0,0,1,
   1,0,0,0,0,0,0,0,1,
   0,0,0,1,0,0,0,0,1,
   0,0,1,0,0,0,0,0,1,
   0,0,0,0,0,1,0,0,1,
  0,0,0,0,1,0,0,0,1,
  0,0,0,0,0,0,0,1,1,
   0,0,0,0,0,0,1,0,1,
   1,1,1,1,1,1,1,1,0),
 nrow = 9, ncol = 9, byrow = TRUE)
colnames(BorgattiEgoNet) <- rownames(BorgattiEgoNet) <- c("A", "B", "C",
                                                          "D", "E", "F",
                                                         "G", "H", "ego")
#the effective size value for the ego replicates that provided in Borgatti (1997)
netstats_om_effective(BorgattiEgoNet)

# For this example, we recreate the ego network provided in Burt (1992: 56):
BurtEgoNet <- matrix(c(
```

```
  0,1,0,0,1,1,1,
 1,0,0,1,0,0,1,
 0,0,0,0,0,0,1,
 0,1,0,0,0,0,1,
 1,0,0,0,0,0,1,
 1,0,0,0,0,0,1,
 1,1,1,1,1,1,0),
 nrow = 7, ncol = 7)
colnames(BurtEgoNet) <- rownames(BurtEgoNet) <- c("A", "B", "C", "D", "E",
                                                  "F", "ego")
#the effective size value for the ego replicates that provided in Burt (1992: 56)
netstats_om_effective(BurtEgoNet)
```

`netstats_om_nwalks` *Compute the Number of Walks of Length K in a One-Mode Network*

Description

[Stable]

This function calculates the number of walks of length *k* between any two vertices in an unweighted one-mode network.

Usage

```
netstats_om_nwalks(net, k)
```

Arguments

`net`	An unweighted one-mode network adjacency matrix.
`k`	A numerical value that corresponds to the length of the paths to be computed.

Details

A nice result from graph theory is that the number of walks of length *k* between vertices *i* and *j* can be found by:

$$A_{ij}^{k}$$

This function assumes that there are no self-loops (i.e., the diagonal of the matrix is 0).

Value

An *n* x *n* matrix of counts of paths.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

Examples

```
# For this example, we generate a random one-mode graph with the sna package.
#creating the random network with 10 actors
set.seed(9999)
rnet <- matrix(sample(c(0,1), 10*10, replace = TRUE, prob = c(0.8,0.2)),
               nrow = 10, ncol = 10, byrow = TRUE)
diag(rnet) <- 0 #setting self ties to 0
#counting the walks of length 2
netstats_om_nwalks(rnet, k = 2)
#counting the walks of length 5
netstats_om_nwalks(rnet, k = 5)
```

`netstats_om_pib` *Compute Potential for Intercultural Brokerage (PIB) Based on Leal (2025)*

Description

[Stable]

Following Leal (2025), this function calculates node's Potential for Intercultural Brokerage (PIB) in a one-mode network, that is, brokerage based on nodes' distinct group memberships. For example, users can examine PIB based on actors' gender. The option count determines what is returned by the function. If count is TRUE, then the count of 'culturally' dissimilar pairs brokered by ego is included (i.e., ego's total count of brokered open triangles where the alters at the two endpoints of said open triangles are 'culturally' dissimilar from one another). If count is FALSE, the proportion of ego's brokered open triangles where the endpoints are 'culturally' dissimilar out of all of ego's brokered open triangles (regardless of the cultural identity of the alters) is returned. The formula for computing interpersonal brokerage is presented in the details section.

Usage

```
netstats_om_pib(
  net,
  g.mem,
  symmetric = TRUE,
  triad.type = NULL,
  count = TRUE,
  isolate = NA
)
```

Arguments

`net`	The one-mode adjacency matrix.
`g.mem`	The vector of membership values that the brokerage scores will be based on.
`symmetric`	TRUE/FALSE. TRUE indicates that network matrix will be treated as symmetric. FALSE indicates that the network matrix will be treated as asymmetric. Set to TRUE by default.

`triad.type`	The string value (or vector) that indicates what specific triadic (star) structures the potential for cultural brokerage will be computed for. Possible values are "ANY", "OTS", "ITS", "MTS" (see the details section). The function defaults to "ANY".
`count`	TRUE/FALSE. TRUE indicates that the number of culturally brokered open triangles will be returned. FALSE indicates that the proportion of culturally brokered open triangles to all open triangles will be returned (see the details section). Set to TRUE by default.
`isolate`	If count = FALSE, the numerical value that will be given to isolates. This value is set to NA by default, as 0/0 is undefined. The user can specify this value!

Details

Following Leal (2025), the formula for interpersonal brokerage is:

$$\text{PIB}_i = \sum_{j<k} \frac{S_{jik}}{S_{jk}} m_{jk}, \quad S_{jik} \neq 0 \text{ and } i \neq j \neq k$$

where:

- $S_{jik} = 1$ if there is an (un)directed two-path connecting actors *j* and *k* through actor *i*; 0 otherwise.
- $m_{jk} = 1$ if actors *j* and *k* are on different sides of a symbolic boundary; 0 otherwise.
- Following Gould (1989), S_{jik} represents the total number of two-paths between actors *j* and *k*.

If the network is non-symmetric (i.e., the user specified symmetric = FALSE), then the function can compute the cultural brokerage scores for different star structures. The possible values are: "ANY", which computes the scores for all structures, where a tie exists between *i* and *j*, *j* and *k*, and one does not exist between *i* and *k*. "OTS" computes the values for outgoing two-stars (i<-j->k or the 021D triad according to the M.A.N. notation; see Wasserman and Faust 1994), where j is the broker. "ITS" computes the values for incoming two-stars (i->j<-k or the 021U triad according to the M.A.N. notation; see Wasserman and Faust 1994), where j is the broker. "MTS" computes PIB for mixed triadic structures (i<-j<-k or i->j->k or the 021C triad according to the M.A.N. notation; see Wasserman and Faust 1994). If not specified, the function defaults to the "ANY" category. This function can also compute all of the formations at once.

Value

The vector of interpersonal cultural brokerage values for the one-mode network.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Gould, Roger. 1989. "Power and Social Structure in Community Elites." *Social Forces* 68(2): 531-552.

Leal, Diego F. 2025. "Locating Cultural Holes Brokers in Diffusion Dynamics Across Bright Symbolic Boundaries." *Sociological Methods & Research* doi:10.1177/00491241251322517

Wasserman, Stanley and Katherine Faust. 1994. *Social Network Analysis: Methods and Applications*. Cambridge: Cambridge University Press.

Examples

```
# For this example, we recreate Figure 3 in Leal (2025)
LealNet <- matrix( c(
 0,1,0,0,0,0,0,
 1,0,1,1,0,0,0,
 0,1,0,0,1,1,0,
 0,1,0,0,1,0,0,
 0,0,1,1,0,0,0,
 0,0,1,0,0,0,1,
 0,0,0,0,0,1,0),
 nrow = 7, ncol = 7, byrow = TRUE)

colnames(LealNet) <- rownames(LealNet) <- c("A", "B", "C","D",
                                            "E", "F", "G")
categorical_variable <- c(0,0,1,0,0,0,0)
#These values are exactly the same as reported by Leal (2025)
netstats_om_pib(LealNet,
   symmetric = TRUE,
   g.mem = categorical_variable)
```

`netstats_tm_constraint`

Compute Burchard and Cornwell's (2018) Two-Mode Constraint

Description

[Stable]

This function calculates the values for two-mode network constraint for weighted and unweighted two-mode networks based on Burchard and Cornwell (2018).

Usage

```
netstats_tm_constraint(
  net,
  isolates = NA,
```

```
  returnCIJmat = FALSE,
  weighted = FALSE
)
```

Arguments

net	A two-mode adjacency matrix or affiliation matrix.
isolates	What value should isolates be given? Preset to be NA.
returnCIJmat	TRUE/FALSE. TRUE indicates that the full constraint matrix, that is, the network constraint from an alter j on node i, will be returned to the user. FALSE indicates that the total constraint will be returned. Set to FALSE by default.
weighted	TRUE/FALSE. TRUE indicates the resulting statistic will be based on the weighted formula (see the details section). FALSE indicates the statistic will be based on the original non-weighted formula. Set to FALSE by default.

Details

Following Burchard and Cornwell (2018), the formula for two-mode constraint is:

$$c_{ij} = \left(\frac{|\zeta(j) \cap \zeta(i)|}{|\zeta^{(i*)}|} \right)^2$$

where:

- c_{ij} is the constraint of ego *i* with respect to actor *j*.
- $|\zeta(j) \cap \zeta(i)|$ is the number of opposite-class contacts that *i* and *j* both share.
- The denominator, $|\zeta^{(i*)}|$, represents the total number of opposite-class contacts of ego *i* excluding pendants. Pendants are level 2 groups that only have one member (i.e., incoming tie).

The total constraint for ego *i* is given by:

$$C_i = \sum_{j \in \sigma(i)} c_{ij}$$

The function returns the aggregate constraint for each actor; however, the user can specify the function to return the constraint matrix by setting *returnCIJmat* to TRUE.

The function can also compute constraint for weighted two-mode networks by setting *weighted* to TRUE. The formula for two-mode weighted constraint is:

$$c_{ij} = \left(\frac{|\zeta(j) \cap \zeta(i)|}{|\zeta^{(i*)}|} \right)^2 \times w_t$$

where w_t is the average of the tie weights that *i* and *j* send to their shared opposite-class contacts.

Value

The vector of two-mode constraint scores for level 1 actors in a two-mode network.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Burchard, Jake and Benjamin Cornwell. 2018. "Structural Holes and Bridging in Two-Mode Networks." *Social Networks* 55:11-20.

Examples

```
# For this example, we recreate Figure 2 in Burchard and Cornwell (2018: 13)
BCNet <- matrix(
 c(1,1,0,0,
   1,0,1,0,
   1,0,0,1,
   0,1,1,1),
 nrow = 4, ncol = 4, byrow = TRUE)
colnames(BCNet) <- c("1", "2", "3", "4")
rownames(BCNet) <- c("i", "j", "k", "m")
#library(sna) #To plot the two mode network, we use the sna R package
#gplot(BCNet, usearrows = FALSE,
#      gmode = "twomode", displaylabels = TRUE)
netstats_tm_constraint(BCNet)

#For this example, we recreate Figure 9 in Burchard and Cornwell (2018:18) for
#weighted two mode networks.
BCweighted <- matrix(c(1,2,1, 1,0,0,
                       0,2,1,0,0,1),
                     nrow = 4, ncol = 3,
                     byrow = TRUE)
rownames(BCweighted) <- c("i", "j", "k", "l")
netstats_tm_constraint(BCweighted, weighted = TRUE)
```

`netstats_tm_degreecent`

Compute Degree Centrality Values for Two-Mode Networks

Description

[Stable]

This function computes the degree centrality values for two-mode networks following Knoke and Yang (2020). The computed degree centrality is based on the specified level. That is, in an affiliation matrix, the density can be computed on the symmetric *g x g* co-membership matrix of level 1 actors (e.g., medical doctors) or on the symmetric *h x h* shared actors matrix for level 2 groups (e.g., hospitals) based on their shared members.

Usage

```
netstats_tm_degreecent(net, level1 = TRUE)
```

Arguments

net	A two-mode adjacency matrix
level1	TRUE/FALSE. TRUE indicates that the degree centrality will be computed for level 1 nodes. FALSE indicates that the degree centrality will be computed for level 2 nodes. Set to TRUE by default.

Details

Following Knoke and Yang (2020), the computation of degree for two-mode affiliation networks is level specific. A two-mode affiliation matrix *X* with dimensions *g x h*, where *g* is the number of level 1 nodes (e.g., medical doctors) and *h* is the number of level 2 nodes (i.e., hospitals). If the function is defined on the level 1 nodes, the degree centrality of an actor *i* is computed as:

$$X^G = XX^T$$

$$C_D^G(g_i) = \sum_{i=1}^{g} x_{ij}^g \quad (i \neq j)$$

In contrast, if it is defined on the level 2 nodes, the degree centrality of an actor *i* is computed as:

$$X^H = X^T X$$

$$C_D^H(h_i) = \sum_{i=1}^{h} x_{ij}^h \quad (i \neq j)$$

Value

The vector of two-mode level-specific degree centrality values.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Knoke, David and Song Yang. 2020. *Social Network Analysis*. Sage: Quantitative Applications in the Social Sciences (154)

Examples

```
#Replicating the biparitate graph presented in Knoke and Yang (2020: 109)
knoke_yang_PC <- matrix(c(1,1,0,0, 1,1,0,0,
                          1,1,1,0, 0,0,1,1,
                          0,0,1,1), byrow = TRUE,
                          nrow = 5, ncol = 4)
colnames(knoke_yang_PC) <- c("Rubio-R","McConnell-R", "Reid-D", "Sanders-D")
```

```
rownames(knoke_yang_PC) <- c("UPS", "MS", "HD", "SEU", "ANA")
netstats_tm_degreecent(knoke_yang_PC, level1 = TRUE) #this value matches the book
netstats_tm_degreecent(knoke_yang_PC, level1 = FALSE) #this value matches the book
```

`netstats_tm_density` *Compute Level-Specific Graph Density for Two-Mode Networks*

Description

[Stable]

This function computes the density of a two-mode network following Wasserman and Faust (1994) and Knoke and Yang (2020). The density is computed based on the specified level. That is, in an affiliation matrix, density can be computed on the symmetric *g x g* matrix of co-membership for the level 1 actors or on the symmetric *h x h* matrix of shared actors for level 2 groups.

Usage

```
netstats_tm_density(net, binary = FALSE, level1 = TRUE)
```

Arguments

`net`	A two-mode adjacency matrix.
`binary`	TRUE/FALSE. TRUE indicates that the transposed matrices will be binarized (see Wasserman and Faust 1995: 316). FALSE indicates that the transposed matrices will not be binarized. Set to FALSE by default.
`level1`	TRUE/FALSE. TRUE indicates that the graph density will be computed for level 1 nodes. FALSE indicates that the graph density will be computed for level 2 nodes. Set to FALSE by default.

Details

Following Wasserman and Faust (1994) and Knoke and Yang (2020), the computation of density for two-mode networks is level specific. A two-mode matrix *X* with dimensions *g x h*, where *g* is the number of level 1 nodes (e.g., medical doctors) and *h* is the number of level 2 nodes (i.e., hospitals). If the function is defined on the level 1 nodes, the density is computed as:

$$X^g = XX^T$$
$$D^g = \frac{\sum_{i=1}^{g}\sum_{j=1}^{g} x_{ij}^g}{g(g-1)}$$

In contrast, if it is defined on the level 2 nodes, the density is:

$$X^h = X^T X$$
$$D^h = \frac{\sum_{i=1}^{h}\sum_{j=1}^{h} x_{ij}^h}{h(h-1)}$$

Moreover, as discussed in Wasserman and Faust (1994: 316), the density can be based on the dichotomous relations instead of the shared membership values. This can be specified by *binary = TRUE*.

Value

The level-specific network density for the two-mode graph.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Wasserman, Stanley and Katherine Faust. 1994. *Social Network Analysis: Methods and Applications*. Cambridge University Press.

Knoke, David and Song Yang. 2020. *Social Network Analysis*. Sage: Quantitative Applications in the Social Sciences (154).

Examples

```
#Replicating the biparitate graph presented in Knoke and Yang (2020: 109)
knoke_yang_PC <- matrix(c(1,1,0,0, 1,1,0,0,
                          1,1,1,0, 0,0,1,1,
                          0,0,1,1), byrow = TRUE,
                          nrow = 5, ncol = 4)
colnames(knoke_yang_PC) <- c("Rubio-R","McConnell-R", "Reid-D", "Sanders-D")
rownames(knoke_yang_PC) <- c("UPS", "MS", "HD", "SEU", "ANA")
#compute two-mode density for level 1
#note: this value does not match that of Knoke and Yang (which we believe
#is a typo in that book), but does match that of Wasserman and
#Faust (1995: 317) for the ceo dataset.
netstats_tm_density(knoke_yang_PC, level1 = TRUE)
#compute two-mode density for level 2.
#note: this value matches that of the book
netstats_tm_density(knoke_yang_PC, level1 = FALSE)
```

---

netstats_tm_effective *Compute Burchard and Cornwell's (2018) Two-Mode Effective Size*

Description

[Stable]

This function calculates the values for two-mode effective size for weighted and unweighted two-mode networks based on Burchard and Cornwell (2018).

Usage

```
netstats_tm_effective(
  net,
  inParallel = FALSE,
  nCores = NULL,
  isolates = NA,
  weighted = FALSE
)
```

Arguments

net	A two-mode adjacency matrix or affiliation matrix
inParallel	TRUE/FALSE. TRUE indicates that parallel processing will be used to compute the statistic with the *foreach* package. FALSE indicates that parallel processing will not be used. Set to FALSE by default.
nCores	If inParallel = TRUE, the number of computing cores for parallel processing. If this value is not specified, then the function internally provides it by dividing the number of available cores in half.
isolates	What value should isolates be given? Preset to be NA.
weighted	TRUE/FALSE. TRUE indicates the resulting statistic will be based on the weighted formula (see the details section). FALSE indicates the statistic will be based on the original non-weighted formula. Set to FALSE by default.

Details

The formula for two-mode effective size is:

$$ES_i = |\sigma(i)| - \sum_{j \in \sigma(i)} r_{ij}$$

where:

- ES_i is the effective size of ego *i*.
- $|\sigma(i)|$ is the number of same-class contacts of ego *i*.
- $\sum_{j \in \sigma(i)} r_{ij}$ is the summation of the redundancy for each alter *j* in the two-mode ego network of *i*.

This function allows the user to compute the scores in parallel through the *foreach* and *doParallel* R packages. If the matrix is weighted, the user should specify *weighted = TRUE*. If the matrix is weighted, following Burchard and Cornwell (2018), the formula for two-mode weighted redundancy is:

$$r_{ij} = \frac{|\sigma(j) \cap \sigma(i)|}{|\sigma(i)| \times w_t}$$

where w_t is the average of the tie weights that *i* and *j* send to their shared opposite class contacts.

Value

The vector of two-mode effective size values for level 1 actors in a two-mode network.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Burchard, Jake and Benjamin Cornwell. 2018. "Structural Holes and Bridging in Two-Mode Networks." *Social Networks* 55:11-20.

Examples

```
# For this example, we recreate Figure 2 in Burchard and Cornwell (2018: 13)
BCNet <- matrix(
 c(1,1,0,0,
   1,0,1,0,
   1,0,0,1,
   0,1,1,1),
 nrow = 4, ncol = 4, byrow = TRUE)
colnames(BCNet) <- c("1", "2", "3", "4")
rownames(BCNet) <- c("i", "j", "k", "m")
#library(sna) #To plot the two mode network, we use the sna R package
#gplot(BCNet, usearrows = FALSE,
#      gmode = "twomode", displaylabels = TRUE)
netstats_tm_effective(BCNet)

#In this example, we recreate Figure 9 in Burchard and Cornwell (2018:18)
#for weighted two mode networks.
BCweighted <- matrix(c(1,2,1, 1,0,0,
                       0,2,1,0,0,1),
                       nrow = 4, ncol = 3,
                       byrow = TRUE)
rownames(BCweighted) <- c("i", "j", "k", "l")
netstats_tm_effective(BCweighted, weighted = TRUE)
```

---

netstats_tm_egodistance

Compute Fujimoto, Snijders, and Valente's (2018) Ego Homophily Distance for Two-Mode Networks

---

Description

[Stable]

This function computes the ego homophily distance in two-mode networks as proposed by Fujimoto, Snijders, and Valente (2018: 380). See Fujimoto, Snijders, and Valente (2018) for more details about this measure.

Usage

```
netstats_tm_egodistance(net, mem, standardize = FALSE)
```

Arguments

`net`	The two-mode adjacency matrix.
`mem`	The vector of membership values that the homophilous four cycles will be based on.
`standardize`	TRUE/FALSE. TRUE indicates that the sores will be standardized by the number of level 2 nodes the level 1 node is connected to. FALSE indicates that the scores will not be standardized. Set to FALSE by default.

Details

The formula for ego homophily distance in two-mode networks is:

$$Ego2Dist_i = \sum_a y_{ia} 1 - |v_i - p_i a|$$

where:

- $\sum_a$ sums across all level 2 nodes in the network
- y_{ia} is the 1 if node i is tied to node a and 0 else.
- v_i is the value of the respondent. Within the function this is predefined to be 1 if there are multiple categories.
- $p_i a$ is the proportion of same-category actors that are tied to node a not including the ego itself.
- $|v_i - p_i a|$ is equal to 1 if all the level 1 nodes that are tied to the level 2 node share the same categorical membership and 0 if all level 1 nodes are a different category.

If the ego is a level 2 isolate or a level 2 pendant, that is, only one level 1 node (e.g., patient) is connected to that specific level 2 node (e.g., medical doctor), then they are given a value of 0. In particular, the contribution to the ego distance for a pendant is 0. The ego distance value can be standardized by the number of groups which would provide the average ego distance as a proportion between 0 and 1.

Value

The vector of two-mode ego homophily distance.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Fujimoto, Kayo, Tom A.B. Snijders, and Thomas W. Valente. 2018. "Multivariate dynamics of one-mode and two-mode networks: Explaining similarity in sports participation among friends." *Network Science* 6(3): 370-395.

Examples

```
# For this example, we use the Davis Southern Women's Dataset.
data("southern.women")
#creating a random binary membership vector
set.seed(9999)
membership <- sample(0:1, nrow(southern.women), replace = TRUE)
#the ego 2 mode distance non-standardized
netstats_tm_egodistance(southern.women, mem = membership)
#the ego 2 mode distance standardized
netstats_tm_egodistance(southern.women, mem = membership, standardize = TRUE)
```

`netstats_tm_homfourcycles`

Compute Fujimoto, Snijders, and Valente's (2018) Homophilous Four-Cycles for Two-Mode Networks

---

Description

[Stable]

This function computes the number of homophilous four-cycles in a two-mode network as proposed by Fujimoto, Snijders, and Valente (2018: 380). See Fujimoto, Snijders, and Valente (2018) for more details about this measure.

Usage

```
netstats_tm_homfourcycles(net, mem)
```

Arguments

`net`	The two-mode adjacency matrix.
`mem`	The vector of membership values that the homophilous four-cycles will be based on.

Details

Following Fujimoto, Snijders, and Valente (2018: 380), the number of homophilous four-cycles for actor *i* is:

$$\sum_{j} \sum_{a \neq b} y_{ia} y_{ib} y_{ja} y_{jb} I v_i = v_j$$

where *y* is the two-mode adjacency matrix, *v* is the vector of membership scores (e.g., sports/club membership), *a* and *b* represent the level two groups, and $Iv_i = v_j$ is the indicator function that is 1 if the values are the same and 0 if not.

Value

The vector of counts of homophilous four-cycles for the two-mode network.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Fujimoto, Kayo, Tom A.B. Snijders, and Thomas W. Valente. 2018. "Multivariate dynamics of one-mode and two-mode networks: Explaining similarity in sports participation among friends." *Network Science* 6(3): 370-395.

Examples

```
# For this example, we use the Davis Southern Women's Dataset.
data("southern.women")
#creating a random binary membership vector
set.seed(9999)
membership <- sample(0:1, nrow(southern.women), replace = TRUE)
#the homophilous four-cycle values
netstats_tm_homfourcycles(southern.women, mem = membership)
```

`netstats_tm_redundancy`

Compute Burchard and Cornwell's (2018) Two-Mode Redundancy

Description

[Stable]

This function calculates the values for two mode redundancy for weighted and unweighted two-mode networks based on Burchard and Cornwell (2018).

Usage

```
netstats_tm_redundancy(
  net,
  inParallel = FALSE,
  nCores = NULL,
  isolates = NA,
  weighted = FALSE
)
```

Arguments

`net`	A two-mode adjacency matrix or affiliation matrix.
`inParallel`	TRUE/FALSE. TRUE indicates that parallel processing will be used to compute the statistic with the *foreach* package. FALSE indicates that parallel processing will not be used. Set to FALSE by default.

nCores	If inParallel = TRUE, the number of computing cores for parallel processing. If this value is not specified, then the function internally provides it by dividing the number of available cores in half.
isolates	What value should isolates be given? Preset to be NA.
weighted	TRUE/FALSE. TRUE indicates the resulting statistic will be based on the weighted formula (see the details section). FALSE indicates the statistic will be based on the original non-weighted formula. Set to FALSE by default.

Details

The formula for two-mode redundancy is:

$$r_{ij} = \frac{|\sigma(j) \cap \sigma(i)|}{|\sigma(i)|}$$

where:

- r_{ij} is the redundancy of ego *i* with respect to actor *j*.
- $|\sigma(j) \cap \sigma(i)|$ is the number of same-class contacts (e.g., medical doctors in a hospital) that *i* and *j* both share.
- $|\sigma(i)|$ is the number of same-class contacts of ego *i*.

The two-mode redundancy is ego-bound, that is, the redundancy is only based on the two-mode ego network of *i*. Put differently, r_{ij} only considers the perspective of the ego. This function allows the user to compute the scores in parallel through the *foreach* and *doParallel* R packages. If the matrix is weighted, the user should specify *weighted = TRUE*. Following Burchard and Cornwell (2018), the formula for two-mode weighted redundancy is:

$$r_{ij} = \frac{|\sigma(j) \cap \sigma(i)|}{|\sigma(i)| \times w_t}$$

where w_t is the average of the tie weights that *i* and *j* send to their shared opposite class contacts.

Value

An *n x n* matrix with level 1 redundancy scores for actors in a two-mode network.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Burchard, Jake and Benjamin Cornwell. 2018. "Structural Holes and bridging in two-mode networks." *Social Networks* 55:11-20.

Examples

```
# For this example, we recreate Figure 2 in Burchard and Cornwell (2018: 13)
BCNet <- matrix(
 c(1,1,0,0,
   1,0,1,0,
   1,0,0,1,
   0,1,1,1),
 nrow = 4, ncol = 4, byrow = TRUE)
colnames(BCNet) <- c("1", "2", "3", "4")
rownames(BCNet) <- c("i", "j", "k", "m")
#this values replicate those reported by Burchard and Cornwell (2018: 14)
netstats_tm_redundancy(BCNet)


#For this example, we recreate Figure 9 in Burchard and Cornwell (2018:18)
#for weighted two mode networks.
BCweighted <- matrix(c(1,2,1, 1,0,0,
                       0,2,1,0,0,1),
                       nrow = 4, ncol = 3,
                       byrow = TRUE)
rownames(BCweighted) <- c("i", "j", "k", "l")
netstats_tm_redundancy(BCweighted, weighted = TRUE)
```

---

plot.dream_gof *Plot method for* dream_gof *Relational Event Model Goodness-of-Fits*

Description

Creates barplots of a dream_gof to plot the proportion of event dyads, senders, and receivers that are correctly specified.

Usage

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

Arguments

x	An object of class dream_gof.
...	Additional arguments for the barplot method.

Details

Generate plots for dream_gof relational event model goodness-of-fits.

Examples

```
#Creating a psuedo one-mode relational event sequence with ordinal timing
relational.seq <- simulate_rem_seq(n_actors = 8,
                                   n_events = 50,
                                   inertia = TRUE,
                                   inertia_p = 0.10,
                                   sender_outdegree = TRUE,
                                   sender_outdegree_p = 0.05)

#Creating a post-processing event sequence for the above relational sequence
post.processing <-  create_res(type = "one-mode",
                               ordinal = TRUE,
                                riskset = "fixed",
                               time = relational.seq$eventID,
                               sender = as.character(relational.seq$sender),
                               receiver = as.character(relational.seq$target),
                               case_control=TRUE,
                               n_controls = 5)

#Computing the sender-outdegree statistic for the above post-processing
#one-mode relational event sequence
post.processing <- dreamstats_degree(formation = "sender-outdegree",
                                   data = post.processing,
                                   halflife = 2)

#Computing the inertia/repetition statistic for the above post-processing
#one-mode relational event sequence
post.processing <- dreamstats_repetition(data = post.processing,
                                         halflife = 2)

#Fitting an ordinal timing relational event model to the above one-mode relational
#event sequence
rem <- estimate_rem(~ sender.outdegree + repetition,
                          data=post.processing)
summary(rem) #summary of the relational event model

#the model goodness of fit
gof <- gof_rem(rem,rseed=3718)
plot(gof) #plotting the goodness of fit assessment
```

`plot.dream_rem` *Plot method for* `dream_rem` *Relational Event Model Fits*

Description

Plot the residuals of a `dream_rem` relational event model fit. Currently, the `plot` function returns plots of either: (1) standardized deviance residual values (y) by realized event times (x) or, for each covariate, (2) the Schoenfeld residual values (y) by realized event times (x). By default, the function plots the standardized deviance residual values.

Usage

```
## S3 method for class 'dream_rem'
plot(x, type = c("std.deviance", "schoenfeld"), ...)
```

Arguments

x	An object of class "dream_rem".
type	If "std.deviance", the returned plot is the absolute value standardized deviance residual values (y) by realized event times (x). If "schoenfeld", then the returned plots are for each covariates, (2) the Schoenfeld residual values (y) by realized event times (x).
...	Additional arguments for other methods.

Details

Generate plots for dream_rem relational event model fits to plot model diagnostics.

Examples

```
#Creating a psuedo one-mode relational event sequence with ordinal timing
relational.seq <- simulate_rem_seq(n_actors = 8,
                                   n_events = 50,
                                   inertia = TRUE,
                                   inertia_p = 0.10,
                                   sender_outdegree = TRUE,
                                   sender_outdegree_p = 0.05)

#Creating a post-processing event sequence for the above relational sequence
post.processing <-  create_res(type = "one-mode",
                               ordinal = TRUE,
                               riskset = "fixed",
                               time = relational.seq$eventID,
                               sender = as.character(relational.seq$sender),
                               receiver = as.character(relational.seq$target),
                               case_control=TRUE,
                               n_controls = 5)

#Computing the sender-outdegree statistic for the above post-processing
#one-mode relational event sequence
post.processing <- dreamstats_degree(formation = "sender-outdegree",
                                     data = post.processing,
                                     halflife = 2)

#Computing the inertia/repetition statistic for the above post-processing
#one-mode relational event sequence
post.processing <- dreamstats_repetition(data = post.processing,
                                         halflife = 2)

#Fitting an ordinal timing relational event model to the above one-mode relational
#event sequence
rem <- estimate_rem(~ sender.outdegree + repetition,
```

```
                          data=post.processing)
summary(rem) #summary of the relational event model

#plotting the absolute value standardized deviance residuals for the estimated model
plot(rem, type="std.deviance")
```

`predict.dream_rem` *Predict method for Relational Event Model Fits*

Description

Predicted event hazard rates based on `dream_rem` relational event model objects.

Usage

```
## S3 method for class 'dream_rem'
predict(object, newdata = NULL, se.fit = FALSE, ...)
```

Arguments

`object`	An object of class "dream_rem".
`newdata`	If requested, either an object of `data.frame` or `dream_sequence` that contains the transformed variables.
`se.fit`	TRUE/FALSE. If TRUE, the standard errors of the predicted event rates will be returned.
`...`	Additional arguments for other methods.

Details

Following Butts (2008: 166), the rate for an event a_i at time t is formulated as:

$$\lambda(a_i) = \exp[\lambda_0 + \theta^T z(s(a_i), r(a_i), x_i, A_t)]$$

where $z()$ is a mapping function that represents a set of sufficient statistics for event a_i. $z()$ represents the covariates included in the estimated relational event model (i.e., the `object` argument) and θ are the estimated parameters/coefficients. λ_0 is the baseline rate across the relational event sequence and the intercept in the interval relational event models and is arbitrary in the ordinal relational event models. In addition, the standard error for each predicted value can be returned when `se.fit` is set to TRUE, and is based upon the delta method, where the standard error of the predicted values are:

$$SE(\hat{\lambda}) = \sqrt{(}\lambda^2 diag(X\Sigma X^T))$$

where $\hat{\lambda}$ is the vector of predicted event rates, X is the matrix of event covariates, and Σ is the covariance matrix of the relational event model.

References

Butts, Carter T. 2008. "A Relational Event Framework for Social Action." *Sociological Methodology* 38(1): 155-200.

Examples

```
#Creating a psuedo one-mode relational event sequence with ordinal timing
relational.seq <- simulate_rem_seq(n_actors = 8,
                                   n_events = 50,
                                   inertia = TRUE,
                                   inertia_p = 0.10,
                                   sender_outdegree = TRUE,
                                   sender_outdegree_p = 0.05)

#Creating a post-processing event sequence for the above relational sequence
post.processing <-  create_res(type = "one-mode",
                               ordinal = TRUE,
                                riskset = "fixed",
                               time = relational.seq$eventID,
                               sender = as.character(relational.seq$sender),
                               receiver = as.character(relational.seq$target),
                               case_control=TRUE,
                               n_controls = 5)

#Computing the sender-outdegree statistic for the above post-processing
#one-mode relational event sequence
post.processing <- dreamstats_degree(formation = "sender-outdegree",
                                   data = post.processing,
                                   halflife = 2)

#Computing the inertia/repetition statistic for the above post-processing
#one-mode relational event sequence
post.processing <- dreamstats_repetition(data = post.processing,
                                         halflife = 2)

#Fitting an ordinal timing relational event model to the above one-mode relational
#event sequence
rem <- estimate_rem(~ sender.outdegree + repetition,
                          data=post.processing)
summary(rem) #summary of the relational event model

#the predicted event rates
rates <- predict(rem)
hist(rates)


#adding a new model matrix specification (this is purely an example) based
#upon a `data.frame` object
changed.data <- data.frame(sender.outdegree = rnorm(100),
                           repetition = sample(0:1, 100, TRUE))
new.rates <- predict(rem, newdata = changed.data)
hist(new.rates)

#adding a new model matrix specification (this is purely an example) based
#upon the transformed `dream_sequence` object
new.sequence <- dreamstats_degree(formation = "sender-outdegree",
                                  data = post.processing,
```

```
                                    halflife = 4) #updating the halflife value
new.rates <- predict(rem, newdata = new.sequence)
hist(new.rates)
```

---

print.dream_gof *Print Method for* dream_gof *Model*

Description

Print Method for dream_gof Model

Usage

```
## S3 method for class 'dream_gof'
print(x, digits = 5, ...)
```

Arguments

x	An object of class dream_gof.
digits	The number of digits to print after the decimal point.
...	Additional arguments (currently unused).

Value

No return value. Prints out the main results of a 'dream_gof' object.

---

print.dream_rem *Print Method for dreamrem Model*

Description

Print Method for dreamrem Model

Usage

```
## S3 method for class 'dream_rem'
print(x, digits = 6, ...)
```

Arguments

x	An object of class "dream_rem".
digits	The number of digits to print after the decimal point.
...	Additional arguments (currently unused).

Value

No return value. Prints out the main results of a 'dream' object.

print.dream_sequence *Print Method for 'dream' object*

Description

Print Method for 'dream' object

Usage

```
## S3 method for class 'dream_sequence'
print(x, digits = 4, ...)
```

Arguments

x	An object of class 'dream_support' .
digits	The number of digits to print after the decimal point.
...	Additional arguments (currently unused).

Value

No return value. Prints out the main results of a 'dream' object.

print.summary.dream_gof *Print Method for* summary.dream_gof *Model*

Description

Print Method for summary.dream_gof Model

Usage

```
## S3 method for class 'summary.dream_gof'
print(x, digits = 5, ...)
```

Arguments

x	An object of class summary.dream_gof
digits	The number of digits to print after the decimal point.
...	Additional arguments (currently unused).

Value

No return value. Prints out the main results of a summary.dream_gof summary object.

print.summary.dream_rem

Print Method for dreamrem Model

Description

Print Method for dreamrem Model

Usage

```
## S3 method for class 'summary.dream_rem'
print(x, digits = 6, ...)
```

Arguments

x	An object of class "dream_rem".
digits	The number of digits to print after the decimal point.
...	Additional arguments (currently unused).

Value

No return value. Prints out the main results of a 'dream' summary object.

print.summary.dream_sequence

Print Method for dream Model

Description

Print Method for dream Model

Usage

```
## S3 method for class 'summary.dream_sequence'
print(x, digits = 3, ...)
```

Arguments

x	An object of class "dream_sequence".
digits	The number of digits to print after the decimal point.
...	Additional arguments (currently unused).

Value

No return value. Prints out the main results of a 'dream_sequence' summary object.

`residuals.dream_rem` *Model residuals for* `dream_rem` *relational event model fits*

Description

This function returns a set of model residuals based upon the realized events in a relational event sequence (Butts 2008). The residuals included are: unit deviance, residual deviance, standardized residual deviance, and for each covariate, the Schoenfeld residual (please see Schoenfeld 1982).

Usage

```
## S3 method for class 'dream_rem'
residuals(object, ...)
```

Arguments

`object`	An object of class "dream_rem".
`...`	Additional arguments for other methods.

Details

The residuals, based upon an estimated relational event model, included are: unit deviance, residual deviance, standardized residual deviance, and for each covariate, the Schoenfeld residual (please see Schoenfeld 1982).

Value

A `data.frame` object that contains the following results:

- `time` - The timing of each realized event.
- `unit.deviance` - The unit deviance for each realized event (the contribution of each realized event to the log-likelihood).
- `residual.deviance` - The residual deviance for each realized event (the square of the contribution of each realized event to the log-likelihood).
- `standardized.deviance` - The standardized residual deviance for each realized event (the square of the contribution of each realized event to the log-likelihood).
- `schof.resid` - The Schoenfeld residual for each included covariate.

References

Butts, Carter T. 2008. "A Relational Event Framework for Social Action." *Sociological Methodology* 38(1): 155-200.

Schoenfeld, David. 1982 "Partial Residuals for The Proportional Hazards Regression Model." *Biometrika* 69(1): 239-241.

Examples

```
#Creating a psuedo one-mode relational event sequence with ordinal timing
relational.seq <- simulate_rem_seq(n_actors = 8,
                                   n_events = 50,
                                   inertia = TRUE,
                                   inertia_p = 0.10,
                                   sender_outdegree = TRUE,
                                   sender_outdegree_p = 0.05)

#Creating a post-processing event sequence for the above relational sequence
post.processing <-  create_res(type = "one-mode",
                               ordinal = TRUE,
                               riskset = "fixed",
                               time = relational.seq$eventID,
                               sender = as.character(relational.seq$sender),
                               receiver = as.character(relational.seq$target),
                               case_control=TRUE,
                               n_controls = 5)

#Computing the sender-outdegree statistic for the above post-processing
#one-mode relational event sequence
post.processing <- dreamstats_degree(formation = "sender-outdegree",
                                   data = post.processing,
                                   halflife = 2)

#Computing the inertia/repetition statistic for the above post-processing
#one-mode relational event sequence
post.processing <- dreamstats_repetition(data = post.processing,
                                         halflife = 2)

#Fitting an ordinal timing relational event model to the above one-mode relational
#event sequence
rem <- estimate_rem(~ sender.outdegree + repetition,
                          data=post.processing)
summary(rem) #summary of the relational event model

#the model residuals
residuals(rem)
```

`simulate_rem_seq` *Simulate a Random One-Mode Relational Event Sequence*

Description

[Stable]

The function allows users to simulate a random one-mode relational event sequence between n actors for k events. This function follows the methods discussed in Butts (2008), Amati, Lomi, and Snijders (2024), and Scheter and Quintane (2021). See the details section for more information on this algorithm. Importanty, this function can be used to simulate a random event sequence to assess

the goodness of fit for ordinal timing relational event models (see Amati, Lomi, and Snijders 2024), and simulate random outcomes for relational outcome models.

Usage

```
simulate_rem_seq(
  n_actors,
  n_events,
  inertia = FALSE,
  inertia_p = 0,
  recip = FALSE,
  recip_p = 0,
  sender_outdegree = FALSE,
  sender_outdegree_p = 0,
  sender_indegree = FALSE,
  sender_indegree_p = 0,
  target_outdegree = FALSE,
  target_outdegree_p = 0,
  target_indegree = FALSE,
  target_indegree_p = 0,
  assort = FALSE,
  assort_p = 0,
  trans_trips = FALSE,
  trans_trips_p = 0,
  three_cycles = FALSE,
  three_cycles_p = 0,
  starting_events = NULL,
  returnStats = FALSE,
  rseed = 9999
)
```

Arguments

n_actors	The number of potential actors in the event sequence.
n_events	The number of simulated events for the relational event sequence.
inertia	TRUE/FALSE. True indicates the effect will be included (see the details section). FALSE indicates the effect will not be included.
inertia_p	If *inertia* = TRUE, the numerical value that corresponds to the parameter weight for the inertia statistic.
recip	TRUE/FALSE. True indicates the effect will be included (see the details section). FALSE indicates the effect will not be included.
recip_p	If *recip* = TRUE, the numerical value that corresponds to the parameter weight for the reciprocity statistic.
sender_outdegree	TRUE/FALSE. True indicates the effect will be included (see the details section). FALSE indicates the effect will not be included.

`sender_outdegree_p`
: If *sender_outdegree* = TRUE, the numerical value that corresponds to the parameter weight for the outdegree statistic.

`sender_indegree`
: TRUE/FALSE. True indicates the effect will be included (see the details section). FALSE indicates the effect will not be included.

`sender_indegree_p`
: If *sender_indegree* = TRUE, the numerical value that corresponds to the parameter weight for the indegree statistic.

`target_outdegree`
: TRUE/FALSE. True indicates the effect will be included (see the details section). FALSE indicates the effect will not be included.

`target_outdegree_p`
: If *target_outdegree* = TRUE, the numerical value that corresponds to the parameter weight for the outdegree statistic.

`target_indegree`
: TRUE/FALSE. True indicates the effect will be included (see the details section). FALSE indicates the effect will not be included.

`target_indegree_p`
: If *target_indegree* = TRUE, the numerical value that corresponds to the parameter weight for the indegree statistic.

`assort`
: Boolean. TRUE/FALSE. True indicates the effect will be included (see the details section). FALSE indicates the effect will not be included.

`assort_p`
: If *assort* = TRUE, the numerical value that corresponds to the parameter weight for the assortativity statistic.

`trans_trips`
: TRUE/FALSE. True indicates the effect will be included (see the details section). FALSE indicates the effect will not be included.

`trans_trips_p`
: If *trans_trips* = TRUE, the numerical value that corresponds to the parameter weight for the transitive triplets statistic.

`three_cycles`
: TRUE/FALSE. True indicates the effect will be included (see the details section). FALSE indicates the effect will not be included.

`three_cycles_p`
: If *three_cycles* = TRUE, the numerical value that corresponds to the parameter weight for the three cycles statistic.

`starting_events`
: A *n* x 2 dataframe with *n* starting events and 2 columns. The first column should be the sender and the second should be the target.

`returnStats`
: TRUE/FALSE. TRUE indicates that the requested network statistics will be returned alongside the simulated relational event sequence. FALSE indicates that only the simulated relational event sequence will be returned. Set to FALSE by default.

`rseed`
: A value for the starting seed for the random number generator. Set to 9999 by default.

Details

Following the authors listed in the descriptions section, the probability of selecting a new event for *t+1* based on the past relational history, H_t, from $0 < t < t + 1$ is given by:

$$p(e_t) = \frac{\lambda ij(t;\theta)}{\sum_{(u,v)\in R_t} \lambda_{uv}(t;\theta)}$$

where *(i,j,t)* is the triplet that corresponds to the dyadic pair with sender *i* and target *j* at time *t* contained in the full risk set, R_t, based on the past relational history. $\lambda_{ij}(t;\theta)$ is formulated as:

$$\lambda_{ij}(t;\theta) = e^{\sum_p \theta_p X_{ijp}(H_t)}$$

where θ_p corresponds to the specific parameter weight given by the user, and X_{ijp} represents the value of the specific statistic based on the current past relational history H_t.

Following Scheter and Quintane (2021) and Amati, Lomi, and Snijders (2024), the algorithm for simulating the random relational sequence for *k* events is:

- 1. Initialize the full risk set, R_t, which is the full Cartesian plot of actors.
- 2. Randomly sample the first event e_1 and add that event into the relational history, H_t.
- 3. Until *i* = *k*, compute the sufficient statistics for each event in the risk set, sample a new event e_i based on the probability function specified above, and add that element into the relational history.
- 4. End when *i* > *k*.

Currently, the function supports 6 statistics for one-mode networks. These are:

- Inertia: n_{ijt}
- Reciprocity: n_{jit}
- Target Indegree: $\sum_k n_{kjt}$
- Target Outdegree: $\sum_k n_{jkt}$
- Sender Outdegree: $\sum_k n_{ikt}$
- Sender Indegree: $\sum_k n_{kit}$
- Assortativity: $\sum_k n_{kit} \cdot \sum_k n_{ikt}$
- Transitive Triplets: $\sum_k n_{ikt} \cdot n_{kjt}$
- Three Cycles: $\sum_k n_{jkt} \cdot n_{kit}$

Where *n* represents the counts of past events, *i* is the event sender, and *j* is the event target. See Scheter and Quintane (2021) and Butts (2008) for a further discussion of these statistics.

Users are allowed to insert a starting event sequence to base the simulation on. A few things are worth nothing. The starting event sequence should be a matrix with *n* rows indicating the number of starting events and 2 columns, with the first representing the event senders and the second column representing the event receivers. Internally, the number of actors is ignored, as the number of possible actors in the risk set is based only on the actors present in the starting event sequence. Finally, the sender and receiver actor IDs should be numerical values.

Value

A data frame that contains the simulated relational event sequence with the sufficient statistics (if requested).

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Amati, Viviana, Alessandro Lomi, and Tom A.B. Snijders. 2024. "A goodness of fit framework for relational event models." *Journal of the Royal Statistical Society Series A: Statistics in Society* 187(4): 967-988.

Butts, Carter T. "A Relational Framework for Social Action." *Sociological Methodology* 38: 155-200.

Schecter, Aaron and Eric Quintane. 2021 "The Power, Accuracy, and Precision of the Relational Event Model." *Organizational Research Methods* 24(4): 802-829.

Examples

```
#Creating a random relational sequence with 5 actors and 25 events
rem1<- simulate_rem_seq(n_actors = 25,
                     n_events = 1000,
                     inertia = TRUE,
                     inertia_p = 0.12,
                     recip = TRUE,
                     recip_p = 0.08,
                     sender_outdegree = TRUE,
                     sender_outdegree_p = 0.09,
                     target_indegree = TRUE,
                     target_indegree_p = 0.05,
                     assort = TRUE,
                     assort_p = -0.01,
                     trans_trips = TRUE,
                     trans_trips_p = 0.09,
                     three_cycles = TRUE,
                     three_cycles_p = 0.04,
                     starting_events = NULL,
                     returnStats = TRUE)
rem1

#Creating a random relational sequence with 100 actors and 1000 events with
#only inertia and reciprocity
rem2 <- simulate_rem_seq(n_actors = 100,
                     n_events = 1000,
                     inertia = TRUE,
                     inertia_p = 0.12,
                     recip = TRUE,
                     recip_p = 0.08,
                     returnStats = TRUE)
```

```
rem2

#Creating a random relational sequence based on the starting sequence with
#only inertia and reciprocity
rem3 <- simulate_rem_seq(n_actors = 100, #does not matter can be any value, this is
                                         #overridden by the starting event sequence
                     n_events = 100,
                     inertia = TRUE,
                     inertia_p = 0.12,
                     recip = TRUE,
                     recip_p = 0.08,
                     #a random starting event sequence
                     starting_events = matrix(c(1:10, 10:1),
                     nrow = 10, ncol = 2,  byrow = FALSE),
                     returnStats = TRUE)
rem3
```

---

simulated.res *Simulated Ordinal Timing One-Mode Relational Event Sequence*

Description

Simulated Ordinal Timing One-Mode Relational Event Sequence

Usage

```
data(simulated.res)
```

Format

`simulated.res`**:**

A simulated one-mode ordinal timing relational event sequence with 500 realized relational events and 10 actors. The simulated one-mode relational event sequence was generated with the `simulate_rem_seq` function the following effects specified: sender outdegree ($\theta = 0.09$), receiver indegree ($\theta = 0.05$), and transitive triplets/outgoing two-paths ($\theta = 0.09$).

---

southern.women *Davis Southern Women's Dataset*

Description

Davis Southern Women's Dataset

Usage

```
data(southern.women)
```

Format

`southern.women`:

Two-Mode affliation matrix from Davis et al.(1941) Southern Women study. 18 women x 14 events. Dataset is taken from the networkdata R package (Almquist 2014)

Source

Almquist, Zach. 2014. *networkdata: Lin Freeman's Network Data Collection*. R package version 0.01, https://github.com/Z-co/networkdata.

Brieger, Ronald. 1974. "Duality of Persons and Groups." *Social Forces* 53(2): 181-190.

Davis, Allison, Burleigh B. Gardner, and Mary R. Gardner. 1941. *Deep South: A Social Anthropological Study of Caste and Class*. University of Chicago Press.

---

summary.dream_gof *Summary Method for* dream_gof *Objects*

Description

Summarizes the results of a goodness-of-fit assessment for estimated relational event models.

Usage

```
## S3 method for class 'dream_gof'
summary(object, digits = 5, ...)
```

Arguments

object	An object of class dream_gof.
digits	The number of digits to print after the decimal point.
...	Additional arguments (currently unused).

Value

A list of summary statistics for the goodness-of-fit assessment.

summary.dream_rem *Summary Method for dreamrem Objects*

Description

Summarizes the results of an ordinal timing relational event model.

Usage

```
## S3 method for class 'dream_rem'
summary(object, digits = 6, ...)
```

Arguments

object	An object of class "dream_rem".
digits	The number of digits to print after the decimal point.
...	Additional arguments (currently unused).

Value

A list of summary statistics for the relational event model including parameter estimates, (null) likelihoods, and tests of significance for likelihood ratios and estimated parameters.

summary.dream_sequence *Summary Method for dream_sequence Objects*

Description

Summarizes the main components of a processed relational event sequence.

Usage

```
## S3 method for class 'dream_sequence'
summary(object, digits = 4, ...)
```

Arguments

object	An object of class "dream_sequence".
digits	The number of digits to print after the decimal point.
...	Additional arguments (currently unused).

Value

A list of descripitive statistics for a processed relational event sequence.

vcov.dream_rem *Extract variance-covariance matrix from Relational Event Model Fits*

Description

This function extracts the variance-covariance matrix from estimated relational event model fits.

Usage

```
## S3 method for class 'dream_rem'
vcov(object, ...)
```

Arguments

object	An object of class "dream_rem".
...	Additional arguments for other methods.

Examples

```
#Creating a psuedo one-mode relational event sequence with ordinal timing
relational.seq <- simulate_rem_seq(n_actors = 8,
                                   n_events = 50,
                                   inertia = TRUE,
                                   inertia_p = 0.10,
                                   sender_outdegree = TRUE,
                                   sender_outdegree_p = 0.05)

#Creating a post-processing event sequence for the above relational sequence
post.processing <-  create_res(type = "one-mode",
                               riskset = "fixed",
                               ordinal = TRUE,
                               time = relational.seq$eventID,
                               sender = as.character(relational.seq$sender),
                               receiver = as.character(relational.seq$target),
                               case_control=TRUE,
                               n_controls = 5)

#Computing the sender-outdegree statistic for the above post-processing
#one-mode relational event sequence
post.processing <- dreamstats_degree(formation = "sender-outdegree",
                                     data = post.processing,
                                     halflife = 2)

#Computing the inertia/repetition statistic for the above post-processing
#one-mode relational event sequence
post.processing <- dreamstats_repetition(data = post.processing,
                                         halflife = 2)

#Fitting an ordinal timing relational event model to the above one-mode relational
```

```
#event sequence
rem <- estimate_rem(~ sender.outdegree + repetition,
                        data=post.processing)
vcov(rem)
```

`WikiEvent2018.first100k`

Wikipedia Edit Event Sequence 2018

Description

The first 100,000 events of the (two-mode) Wikipedia edit event sequence, where an event is described as a Wikipedia user editing a Wikipedia article. The user column represents the unique event senders, the article column represents the unique event receivers (targets), and the time variable is in milliseconds.

Usage

```
data(WikiEvent2018.first100k)
```

Format

`WikiEvent2018.first100k`**:**

The first 100,000 events of the Wikipedia edit event sequence, where an event is described as a Wikipedia user editing a Wikipedia article. The user column represents the unique event senders, the article column represents the unique event receivers (targets), and the time variable is in milliseconds.

user the column that represents the unique event senders.

article the article column represents the unique event receivers.

time the event time variable in milliseconds.

eventID the numerical id for each event in the event sequence

Source

https://zenodo.org/records/1626323

Lerner, Jurgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: how to fit a relational event model to 360 million dyadic events." *Network Science* 8(1):97-135. (DOI: https://doi.org/10.1017/nws.2019.57)

Index

∗ **datasets**
simulated.res, 100
southern.women, 100
WikiEvent2018.first100k, 104

AIC, *62*
as.data.frame, *3*
as.data.frame.dream_sequence, 3

clogit, *58, 59*
coef.dream_rem, 5, *62*
coxph, *59*
create_res, 6, *15*

dream_information, 13
dream_sequence, 14
dreamstats_actor, 17
dreamstats_actorfe, 19
dreamstats_degree, 21
dreamstats_dyadcut, *23*, 25, *35*, *48*, *51*, *56*
dreamstats_dyadfe, 28
dreamstats_dyadic, 29
dreamstats_event, 32
dreamstats_fourcycles, 34
dreamstats_persistence, 37
dreamstats_prefattachment, 40
dreamstats_recency, 42
dreamstats_reciprocity, 46
dreamstats_repetition, 50
dreamstats_triads, 53

estimate_rem, *7*, 58

glm, *59*
gof_rem, *62*, 64

lm, *59*
logLik.dream_rem, *62*, 66

netstats_om_constraint, 67
netstats_om_effective, 69
netstats_om_nwalks, 71
netstats_om_pib, 72
netstats_tm_constraint, 74
netstats_tm_degreecent, 76
netstats_tm_density, 78
netstats_tm_effective, 79
netstats_tm_egodistance, 81
netstats_tm_homfourcycles, 83
netstats_tm_redundancy, 84

optim, *59, 60*

plot.dream_gof, 86
plot.dream_rem, *62*, 87
predict.dream_rem, *62*, 89
print.dream_gof, 91
print.dream_rem, 91
print.dream_sequence, 92
print.summary.dream_gof, 92
print.summary.dream_rem, 93
print.summary.dream_sequence, 93

remstimate, *7*, *60*
residuals.dream_rem, *62*, 94

simulate_rem_seq, 95, *100*
simulated.res, *14*, 100
southern.women, 100
summary.dream_gof, 101
summary.dream_rem, 102
summary.dream_sequence, 102

vcov.dream_rem, *62*, 103

WikiEvent2018.first100k, *14*, 104