Package: relevent (via r-universe)

September 16, 2024

Version 1.2-1

Date 2023-01-24

Title Relational Event Models

Author Carter T. Butts <`buttsc@uci.edu`>

Maintainer Carter T. Butts <`buttsc@uci.edu`>

Depends trust, sna (>= 2.0), coda

Description Tools to fit and simulate realizations from relational
event models.

License GPL (>= 2)

NeedsCompilation yes

Repository CRAN

Date/Publication 2023-01-24 08:10:02 UTC

Contents

`as.sociomatrix.eventlist` *Convert an Event List Into a Sociomatrix*

Description

Convert a dyadic event list into an adjacency matrix, such that the i, j cell value is the number of (i, j) events in the list.

Usage

```
as.sociomatrix.eventlist(eventlist, n = NULL)
```

Arguments

`eventlist`	a three-column numeric matrix (or equivalent), containing the event list to be converted.
`n`	the number of vertices. If omitted, this is assumed to be contained in an attribute called `"n"` attached to `eventlist`.

Details

An event list must be a three-column matrix (or something that can be treated as one), whose second and third columns must contain vertex IDs; these can be given as characters, but must be coercable with `as.numeric` to numeric form. Vertex IDs must be integers from `1:n`, where `n` is either supplied as an argument, or attached as an attribute of the `eventlist` object. The first column of an eventlist matrix conventionally contains the event time, and is ignored; the second and third should contain the IDs of the senders and receivers of events (respectively). Rows with missing values for one or both vertex IDs are removed during processing (but `NA`s in the first column have no effect, since the event timing information is not used).

The resulting output is an `n` by `n` adjacency matrix, whose `i,j` cell is the total number of events in `eventlist` from vertex `i` to vertex `j`. This can be useful for visualizing or otherwise analyzing the time-marginalized structure of a dyadic interaction network.

Value

A sociomatrix containing the time-aggregated event counts.

Author(s)

Carter T. Butts <`buttsc@uci.edu`>

See Also

`rem.dyad`

Examples

```
#Create a simple event list
el <- cbind(1:6, c(4,4,4,1,3,4), c(1,2,3,3,1,1))

#Convert to matrix form
as.sociomatrix.eventlist(el, 4)

#Can also store n as an attribute
attr(el, "n") <- 4
as.sociomatrix.eventlist(el)
```

rem *Fit a Relational Event Model to Single or Multiple Sequence Data*

Description

Fits a relational event model to general event sequence data, using either the ordinal or interval time likelihoods. Maximum likelihood and posterior mode methods are supported, as are local (per sequence) parameters and sequences with exogenous events.

Usage

```
rem(eventlist, statslist, supplist = NULL, timing = c("ordinal",
    "interval"), estimator = c("BPM", "MLE", "BMCMC", "BSIR"),
    prior.param = list(mu = 0, sigma = 1000, nu = 4), mcmc.draws = 1500,
    mcmc.thin = 25, mcmc.burn = 2000, mcmc.chains = 3, mcmc.sd = 0.05,
    mcmc.ind.int = 50, mcmc.ind.sd = 10, sir.draws = 1000,
    sir.expand = 10, sir.nu = 4, verbose = FALSE)
## S3 method for class 'rem'
print(x, ...)
## S3 method for class 'rem'
summary(object, ...)
```

Arguments

`eventlist`	a two-column matrix (or list thereof) containing the observed event sequence and timing information.
`statslist`	an event number by event type by statistic array (or list thereof) containing the sufficient statistics for the model to be estimated.
`supplist`	an event number by event type logical array (or list thereof) indicating which events were potentially observable at each point in the event history.
`timing`	the type of timing information to be used during estimation; `"ordinal"` indicates that only event order should be employed, while `"interval"` uses the exact inter-event times.
`estimator`	the type of estimator to be used; `"MLE"` selects maximum likelihood estimation, `"BPM"` selects Bayesian posterior mode estimation, `"BMCMC"` selects Bayesian posterior mean estimation via MCMC, and `"BSIR"` selects Bayesian posterior mean estimation via simulated importance resampling.
`prior.param`	for the Bayesian methods, the prior parameters to be employed; currently, these are the location, scale, and degrees of freedom parameters for independent t priors, and may be given as vectors (to set different priors for each parameter). (By default, a diffuse, heavy-tailed t distribution is used.)
`mcmc.draws`	total number of posterior draws to take when using the BMCMC method.
`mcmc.thin`	thinning interval for MCMC draws (BMCMC method).
`mcmc.burn`	number of burn-in iterations to use for each MCMC chain (BMCMC method).

`mcmc.chains`	number of MCMC chains to use (BMCMC method).
`mcmc.sd`	standard deviation for the random walk Metropolis sampler (BMCMC method).
`mcmc.ind.int`	interval at which to take draws from the independence sampler (versus the random walk Metropolis sampler). (BMCMC method).
`mcmc.ind.sd`	standard deviation for the MCMC independence sampler (BMCMC method).
`sir.draws`	number of SIR draws to take (BSIR method).
`sir.expand`	expansion factor for the SIR sample; intitial sample size is `sir.draws` multiplied by `sir.expand`.
`sir.nu`	degrees of freedom parameter for the SIR sampling distribution.
`verbose`	logical; should verbose progress information be displayed?
`x`	an object of class `rem`.
`object`	an object of class `rem`.
`...`	additional arguments.

Details

`rem` fits a general relational event model to one or more event sequences (or "histories"), using either full interval or ordinal timing information. Although particularly applicable to "egocentric" relational event data, `rem` can be used to fit nearly any standard relational event model; the function depends heavily on user-supplied statistics, however, and thus lacks the built-in functionality of a routine like `rem.dyad`. Four estimation methods are currently supported: maximum likelihood estimation, Bayesian posterior mode estimation, Bayesian posterior mean estimation via MCMC, and Bayesian posterior mean estimation via sampling importance resampling (SIR). For the Bayesian methods, adjustable independent t priors are employed. For both mode-based methods, estimates of uncertainty (standard errors or posterior standard deviations) are approximated using the appropriate inverse hessian matrix; for the two simulation-based methods, posterior standard deviations are estimated from the resulting sample.

Irrespective of whether Bayesian or frequentist methods are used, the relevant likelihood is either based entirely on the order of events (`timing="ordinal"`) or on the realized event times (`timing="interval"`). In the latter case, all event times are understood to be relative to the onset of observation (i.e., observation starts at time 0), and the last event time given is taken to be the end of the observation period. (This should generally be marked as exogenous – see below.)

Event source/target/content are handled generically by `rem` via *event types*. Each event must be of a given type, and any number of types may be employed (up to limits of time and memory). Effects within the relational event model are associated with user-supplied statistics, of which any number may again be supplied (model identification notwithstanding). At each point in the event history, it is possible that only particular types of events may be realized; this constraint can be specified by means of an optional user-supplied support structure. Finally, it is also possible that an event sequence may be punctuated by *exogenous events,* which are unmodeled but which may affect the endogenous event dynamics. These are supported by means of a tacit "exogenous" event type, which is handled by the estimation routine as appropriate for the specified likelihood.

Observed event data is supplied to `rem` via the `eventlist` argument. For each event history, the observed events are indicated by a two-column matrix, whose ith row contains respectively the event type (as an integer ranging from 1 to the number of event types, inclusive) and the event time for the ith event in the history. (The second column may be omitted in the ordinal case, and will in any event be ignored.) Events must be given in ascending temporal order; if multiple histories are being modeled simultaneously (e.g., as with egocentric relational event samples), then `eventlist` should be a list with one matrix per event history. Exogenous events, if present, are indicated by specifying an event type of 0. (Note that the "type" of an exogenous event is irrelevant, since any such properties of exogenous events are handled via the model statistics.) If exact timing information is used, the hazard for the first event implicitly begins at time 0, and observation implicitly ends with the time of the last event (which should properly be coded as exogenous, unless the sampling design was based on observation of an endogenous terminal event). Where applicable, censoring due to the sampling interval is accounted for in the data likelihood (assuming that the user has set the model statistics appropriately).

Statistics for the relational event model are specified in a manner somewhat analogous to that of `eventlist`. Like the latter, `statslist` is generally a list with one element per event history, or a single element where only a single history is to be examined. Each element of `statslist` should be a list containing either one or two three-dimensional arrays, with the first dimension indexing event order (from first event to last, including exogenous events where applicable), the second indexing event type (in order corresponding to the integer values of `eventlist`), and the third indexing the model statistics. The ijkth cell of a `statslist` array is thus the value of the kth statistic prospectively impacting the hazard of observing an event of type j as the ith event in the history (given the previous $i-1$ realized events). Models estimated by *rem* are regular in the sense that one parameter is estimated per statistic; intuitively, a large value of a ijkth `statslist` cell associatd with a large (positive) parameter represents an increased hazard of observing a type j event at the ith point in the respective history, while the same statistic associated with a highly negative parameter represents a correspondingly diminished hazard of observing said event. (The total hazard of a given event type is equal to $\exp(\theta^T s_{ij})$, where θ is the vector of model parameters and s_{ij} is the corresponding vector of sufficient statistics for a type j event given the $i-1$ previously realized events; see the reference below for details.) It is up to the user to supply these statistics, and moreover to ensure that they are well-behaved (e.g., not linearly dependent). An array within a `statslist` element may be designated as *global* or *local* by assigning it to the appropriately named list element. Statistics belonging to a global array are assumed to correspond to parameters that are homogeneous across event histories, and are estimated in a pooled fashion; if global arrays are supplied, they must be given for every element of `statslist` (and must carry the same statistics and event types, although these statistics will not typically take the same values). Statistics belonging to a local array, on the other hand, are taken as idiosyncratic to the event history in question, and their corresponding parameters are estimated locally. Both local and global statistics may be employed simultaneously if desired, but at least one must be specified in any case. `rem` will return an error if passed a `statslist` with obvious inconsistencies.

If desired, support constraints for the event histories can be specified using `supplist`. `supplist` should be a list with one element per history, each of which should be an event order by event type logical matrix. The ijth cell of this matrix should be `TRUE` if an event of type `j` was a possible next event given the preceding code $i-1$ events, and `FALSE` otherwise. (By default, all events are assumed to be possible at all times.) As with the model statistics, the elements of the support list must be user supplied, and will often be history-dependent. (E.g., in a model for spell-based data, event types will come in onset/termination pairs, with terminal events necessarily being preceded by corresponding onset events.)

Given the above structure, `rem` will attempt to find a maximum likelihood or posterior estimate for the model parameters, as appropriate given `estimator`. In the latter case, the prior parameters for each parameter may be set using `prior.param`. Each parameter is taken to be *a priori* t distributed, with the indicated location, scale, and degree of freedom parameters; by default, a fairly diffuse and heavy-tailed prior is used. By specifying the elements of `prior.param` as vectors, it is possible to employ different priors for each model parameter. In this case, the vector elements are used in the order of the statistics (first global, then each local in order by event history). Standard errors or posterior standard deviation estimates are returned as appropriate, along with various goodness-of-fit indices. (Bear in mind that the "p-values" shown in the summary method for the posterior mode case are based on posterior quantiles (under an assumption of asymptotic normality), and should be interpreted in this fashion.)

For the MCMC sampling method, a combined independence and random walk Metropolis scheme is employed. Proposals are multivariate Gaussian, with standard deviations as set via the appropriate arguments. (These may be given as vectors, with one entry per parameter, if desired.) Gewke and Gelman-Rubin MCMC diagnostics (produced by the `coda` package) are computed, and are stored as elements `geweke` and `gelman.rubin` within the model fit object. The posterior draws themselves are stored as an element called `draws` within the model fit object, with corresponding log-posterior values `lp`.

The SIR method initially seeks the posterior mode (identically to the BPM method), and obtains approximate scale information using the Hessian of the log-posterior surface. This is used to generate a set of approximate posterior draws via a multivariate t distribution centered on the posterior mode, with degrees of freedom given by `sir.nu`. This crude sample is then refined by importance resampling, the final result of which is stored as element `draws` (with log-posterior vector `lp`) in the model fit object. As with the BMCMC procedure, posterior mean and standard deviations are estimated from the final sample, although the mode information is retained in elements `coef.mode` and `cov.hess`.

As a general matter, the MLE and BPM methods are most dependent upon asymptotic assumptions, but are also (usually) the least computationally complex. BMCMC requires no such assumptions, but can be extremely slow (and, like all MCMC methods, depends upon the quality of the MCMC sample). The BSIR method is something of a compromise between BPM and BMCMC, starting with a mode approximation but refining it in the direction of the true posterior surface; as one might expect, its cost is also intermediate between these extremes. For well-behaved models on large data sets, all methods are likely to produce nearly identical results. The simulation-based methods (particularly BMCMC) may be safer in less salutary circumstances. (Tests conducted by the author have so far obtained the best overall results from the BPM, particularly vis a vis estimates of uncertainty – this advice may or may not generalize, however.)

Value

An object of class `rem`, for which `print` and `summary` methods currently exist.

Author(s)

Carter T. Butts <`buttsc@uci.edu`>

References

Butts, C.T. (2008). "A Relational Event Framework for Social Action." *Sociological Methodology*, 38(1).

See Also

`rem.dyad`

rem.dyad *Fit a Relational Event Model to Dyadic Data*

Description

Fits a relational event model to dyadic edgelist data, using either the ordinal or temporal likelihood. Maximum likelihood, posterior mode, and posterior importance resampling methods are supported.

Usage

```
rem.dyad(edgelist, n, effects = NULL, ordinal = TRUE, acl = NULL,
    cumideg = NULL, cumodeg = NULL, rrl = NULL, covar = NULL, ps = NULL,
    tri = NULL, optim.method = "BFGS", optim.control = list(),
    coef.seed = NULL, hessian = FALSE, sample.size = Inf, verbose = TRUE,
    fit.method = c("BPM", "MLE", "BSIR"), conditioned.obs = 0,
    prior.mean = 0, prior.scale = 100, prior.nu = 4, sir.draws = 500,
    sir.expand = 10, sir.nu = 4, gof = TRUE)
## S3 method for class 'rem.dyad'
print(x, ...)
## S3 method for class 'rem.dyad'
summary(object, ...)
## S3 method for class 'rem.dyad'
simulate(object, nsim = object$m, seed = NULL,
    coef = NULL, covar = NULL, edgelist = NULL, redraw.timing = FALSE,
    redraw.events = FALSE, verbose = FALSE, ...)
```

Arguments

`edgelist`	a three-column edgelist matrix, with each row containing (in order) the time/order, sender, and receiver for the event in question, or `NULL` to create a model skeleton (useful for simulation).
`n`	number of senders/receivers.
`effects`	a character vector indicating which effects to use; see below for specification.
`ordinal`	logical; should the ordinal likelihood be used? (If `FALSE`, the temporal likelihood is used instead.)
`acl`	optionally, a pre-computed acl structure.
`cumideg`	optionally, a pre-computed cumulative indegree structure.
`cumodeg`	optionally, a pre-computed cumulative outdegree stucture.
`rrl`	optionally, a pre-computed recency-ranked communications list.
`covar`	an optional list of sender/receiver/event covariates.
`ps`	optionally, a pre-computed p-shift matrix.
`tri`	optionally, a pre-computed triad statistic structure.
`optim.method`	the method to be used by `optim`.

`optim.control`	additional control parameters to `optim`.
`coef.seed`	an optional vector of coefficients to use as the starting point for the optimization process; if `edgelist==NULL`, this is the vector of embedded coefficients for the model skeleton.
`hessian`	logical; compute the hessian of the log-likelihood/posterior surface?
`sample.size`	sample size to use when estimating the sum of event rates.
`verbose`	logical; deliver progress reports?
`fit.method`	method to use when fitting the model.
`conditioned.obs`	the number of initial observations on which to condition when fitting the model (defaults to 0).
`prior.mean`	for Bayesian estimation, location vector for prior distribution (multivariate-t). (Can be a single value.)
`prior.scale`	for Bayesian estimation, scale vector for prior distribution. (Can be a single value.)
`prior.nu`	for Bayesian estimation, degrees of freedom for prior distribution. (Setting this to `Inf` results in a Gaussian prior.)
`sir.draws`	for sampling importance resampling method, the number of posterior draws to take (post-resampling).
`sir.expand`	for sampling importance resampling method, the expansion factor to use in the initial (pre-resampling) sample; sample size is `sir.expand*sir.draws`.
`sir.nu`	for sampling importance resampling method, the degrees of freedom for the t distribution used to obtain initial (pre-resampling) sample.
`gof`	logical; calculate goodness-of-fit information?
`x`	an object of class `rem.dyad`.
`object`	an object of class `rem.dyad`.
`nsim`	number of events to simulate (defaults to the observed sequence length in the fitted model).
`seed`	random number seed to use for simulation.
`coef`	optional vector of coefficients to override those in the fitted model object, for simulation purposes.
`redraw.timing`	logical; should any prespecified events in `edgelist` have their timings redrawn during simulation?
`redraw.events`	logical; should any prespecified events in `edgelist` have their senders and receivers redrawn during simulation?
`...`	additional arguments.

Details

`rem.dyad` fits a (dyadic) relational event model to an event sequence, using either the full temporal or ordinal data likelihoods. Three estimation methods are currently supported: maximum likelihood estimation, Bayesian posterior mode estimation, and Bayesian sampling importance resampling. For the Bayesian methods, an adjustable multivariate-t (or, if `prior.nu==Inf`, Gaussian) prior is employed. In the case of Bayesian sampling importance resampling, the posterior mode (and the hessian of the posterior about it) is used as the basis for a multivariate-t sample, which is then resampled via SIR methods to obtain an approximate set of posterior draws. While this approximation is not guaranteed to work well, it is generally more robust than pure mode approximations (or, in the case of the MLE, estimates of uncertainty derived from the inverse hessian matrix).

Whether Bayesian or frequentist methods are used, the relevant likelihood is either based entirely on the order of events (`ordinal=TRUE`) or on the realized event times (`ordinal=FALSE`). In the latter case, all event times are understood to be relative to the onset of observation (i.e., observation starts at time 0), and the last event time given is taken to be the end of the observation period. (If an event is also specified, this event is ignored.)

Effects to be fit by `rem.dyad` are determined by the eponymous `effects` argument, a character vector which lists the effects to be used. These are as follows:

- `NIDSnd`: Normalized indegree of v affects v's future sending rate
- `NIDRec`: Normalized indegree of v affects v's future receiving rate
- `NODSnd`: Normalized outdegree of v affects v's future sending rate
- `NODRec`: Normalized outdegree of v affects v's future receiving rate
- `NTDegSnd`: Normalized total degree of v affects v's future sending rate
- `NTDegRec`: Normalized total degree of v affects v's future receiving rate
- `FrPSndSnd`: Fraction of v's past actions directed to v' affects v's future rate of sending to v'
- `FrRecSnd`: Fraction of v's past receipt of actions from v' affects v's future rate of sending to v'
- `RRecSnd`: Recency of receipt of actions from v' affects v's future rate of sending to v'
- `RSndSnd`: Recency of sending to v' affects v's future rate of sending to v'
- `CovSnd`: Covariate effect for outgoing actions (requires a `covar` entry of the same name)
- `CovRec`: Covariate effect for incoming actions (requires a `covar` entry of the same name)
- `CovInt`: Covariate effect for both outgoing and incoming actions (requires a `covar` entry of the same name)
- `CovEvent`: Covariate effect for each (v, v') action (requires a `covar` entry of the same name)
- `OTPSnd`: Number of outbound two-paths from v to v' affects v's future rate of sending to v'
- `ITPSnd`: Number of incoming two-paths from v' to v affects v's future rate of sending to v'
- `OSPSnd`: Number of outbound shared partners for v and v' affects v's future rate of sending to v'
- `ISPSnd`: Number of inbound shared partners for v and v' affects v's future rate of sending to v'
- `FESnd`: Fixed effects for outgoing actions
- `FERec`: Fixed effects for incoming actions
- `FEInt`: Fixed effects for both outgoing and incoming actions
- `PSAB-BA`: P-Shift effect (turn receiving) – AB->BA (dyadic)
- `PSAB-B0`: P-Shift effect (turn receiving) – AB->B0 (non-dyadic)
- `PAAB-BY`: P-Shift effect (turn receiving) – AB->BY (dyadic)

- `PSA0-X0`: P-Shift effect (turn claiming) – A0->X0 (non-dyadic)
- `PSA0-XA`: P-Shift effect (turn claiming) – A0->XA (non-dyadic)
- `PSA0-XY`: P-Shift effect (turn claiming) – A0->XY (non-dyadic)
- `PSAB-X0`: P-Shift effect (turn usurping) – AB->X0 (non-dyadic)
- `PSAB-XA`: P-Shift effect (turn usurping) – AB->XA (dyadic)
- `PSAB-XB`: P-Shift effect (turn usurping) – AB->XB (dyadic)
- `PSAB-XY`: P-Shift effect (turn usurping) – AB->XY (dyadic)
- `PSA0-AY`: P-Shift effect (turn continuing) – A0->AY (non-dyadic)
- `PSAB-A0`: P-Shift effect (turn continuing) – AB->A0 (non-dyadic)
- `PSAB-AY`: P-Shift effect (turn continuing) – AB->AY (dyadic)

Note that not all effects may lead to identified models in all cases - it is up to the user to ensure that the postulated model makes sense.

Data to be used by `rem.dyad` must consist of an edgelist matrix, whose rows contain information on successive events. This matrix must have three columns, containing (respectively) the event times, sender IDs (as integers from 1 to `n`), and receiver IDs (also from 1 to `n`). As already noted, event times should be relative to onset of observation where the temporal likelihood is being used; otherwise, only event order is employed. In the temporal likelihood case, the last row should contain the time for the termination of the observation period – any event on this row is ignored. If `conditioned.obs>0`, the relevant number of initial observations is taken as fixed, and the likelihood of the remaining sequence is calculated conditional on these values; this can be useful when analyzing an event history with no clear starting point.

If covariates effects are indicated, then appropriate covariate values must be supplied as a list in argument `covar`. The elements of `covar` should be given the same name as the effect type to which they correspond (e.g., `CovSnd`, `CovRec`, etc.); any other elements will be ignored. The format of a given covariate element depends both on the effect type and on the number of covariates specified. The basic cases are as follows:

- Single covariate, time invariant: For `CovSnd`, `CovRec`, or `CovInt`, a vector or single-column matrix/array. For `CovEvent`, an `n` by `n` matrix or array.
- Multiple covariates, time invariant: For `CovSnd`, `CovRec`, or `CovInt`, a two-dimensional `n` by `p` matrix/array whose columns contain the respective covariates. For `CovEvent`, a `p` by `n` by `n` array, whose first dimension indexes the covariate matrices.
- Single or multiple covariates, time varying: For `CovSnd`, `CovRec`, or `CovInt`, an `m` by `p` by `n` array whose respective dimensions index time (i.e., event number), covariate, and actor. For `CovEvent`, a `m` by `p` by `n` by `n` array, whose dimensions are analogous to the previous case.

Note that "time varying" covariates may only change values when events transpire; thus, they should be regarded as temporally endogenous. (See the reference below for details.)

If called with `edgelist==NULL`, `rem.dyad` will produce a "model skeleton" object containing the effects and other information, but no model fit. (The seed coefficients, if given, are entered as the coefficients in the model, or else an uninteresting default set is used.) The main purpose for this object is to set up an *ab initio* simulation, as described below: once the skeleton is created, the `simulate` method can be used to generate draws from that model (without fitting to a data set).

A simulate method is provided for rem.dyad objects, which allows simulation of new event sequences from a fitted or skeleton model. By default, a new sequence of length equal to the original sequence to which the model object was fitted is simulated (if applicable), but other lengths may be chosen using nsim. Although the coefficients in the model object are used by default, this may also be altered by specifying coef. Note that any covariates used must be passed to the simulate command via covar (using the same format as in the original model); this is in part because rem.dyad objects do not currently save their input data, and in part because dynamic covariates must always be the length of the simulated sequence (and hence must be factored when a non-default nsim value is used). For models fit using ordinal=TRUE, the overall pacing of events will be arbitrary (more specifically, the simulation will tacitly assume that each event has a unit base hazard), but the relative timing is not. See below for examples of both simulation using a fitted model object and *ab initio* simulation without fitting a model to data.

For simulation, it is possible to fix the first portion of the event history by passing an event list matrix to the edgelist argument; this must be compatible with the target model (i.e., the vertex IDs must match), and it cannot contain NA values. (Thus, if starting with an exact timing seqence with a last line containing NAs, this must be removed.) If the input event list contains m events, then these are assumed to supply the first m events of the target sequence; if m>nsim, then any excess events are discarded. By default, the input events are taken as fixed. However, specifying redraw.timing=TRUE will lead the event timings to be redrawn, and redraw.events will lead the sender/reciver pairs to be redrawn. This allows e.g. for an observed ordinal time sequence to be given a simulated exact time realization, by setting nsim to the event list length and setting redraw.timing=TRUE. The more obvious use case is to simply extend an observed sequence, in which case one should use nsim greater than the input sequence length (i.e., the input length plus the number of new events to generate) and leave the redraw paraeters set to FALSE.

Value

For rem.dyad, an object of class rem.dyad. For the simulate method, an event list.

Author(s)

Carter T. Butts <buttsc@uci.edu>

References

Butts, C.T. (2008). “A Relational Event Framework for Social Action.” *Sociological Methodology*, 38(1).

See Also

rem

Examples

```
## Not run:
#Generate some simple sample data based on fixed effects
roweff<-rnorm(10)                                     #Build rate matrix
roweff<-roweff-roweff[1]                   #Adjust for later convenience
coleff<-rnorm(10)
coleff<-coleff-coleff[1]
```

```
lambda<-exp(outer(roweff,coleff,"+"))
diag(lambda)<-0
ratesum<-sum(lambda)
esnd<-as.vector(row(lambda))                  #List of senders/receivers
erec<-as.vector(col(lambda))
time<-0
edgelist<-vector()
while(time<15){                   # Observe the system for 15 time units
  drawsr<-sample(1:100,1,prob=as.vector(lambda))        #Draw from model
  time<-time+rexp(1,ratesum)
  if(time<=15)                                             #Censor at 15
    edgelist<-rbind(edgelist,c(time,esnd[drawsr],erec[drawsr]))
  else
    edgelist<-rbind(edgelist,c(15,NA,NA))
}

#Fit the model, ordinal BPM
effects<-c("FESnd","FERec")
fit.ord<-rem.dyad(edgelist,10,effects=effects,hessian=TRUE)
summary(fit.ord)
par(mfrow=c(1,2))                                #Check the coefficients
plot(roweff[-1],fit.ord$coef[1:9],asp=1)
abline(0,1)
plot(coleff[-1],fit.ord$coef[10:18],asp=1)
abline(0,1)

#Now, find the temporal BPM
fit.time<-rem.dyad(edgelist,10,effects=effects,ordinal=FALSE,hessian=TRUE)
summary(fit.time)
plot(fit.ord$coef,fit.time$coef,asp=1)                  #Similar results
abline(0,1)

#Finally, try the BSIR method (note: a much larger expansion factor
#is recommended in practice)
fit.bsir<-rem.dyad(edgelist,10,effects=effects,fit.method="BSIR",
    sir.draws=100,sir.expand=5)
summary(fit.bsir)
par(mfrow=c(3,3))   #Examine the approximate posterior marginals
for(i in 1:9){
  hist(fit.bsir$post[,i],main=names(fit.bsir$coef)[i],prob=TRUE)
  abline(v=roweff[i+1],col=2,lwd=3)
}
for(i in 10:18){
  hist(fit.bsir$post[,i],main=names(fit.bsir$coef)[i],prob=TRUE)
  abline(v=coleff[i-8],col=2,lwd=3)
}

#Simulate an event sequence from the temporal model
sim<-simulate(fit.time,nsim=50000) #Simulate 50000 events
head(sim)                          #Show the event list
par(mfrow=c(1,2))                  #Check the behavior
esnd<-exp(c(0,fit.time$coef[1:9]))
esnd<-esnd/sum(esnd)*5e4           #Expected sending count
```

```
erec<-exp(c(0,fit.time$coef[10:18]))
erec<-erec/sum(erec)*5e4           #Expected sending count
plot(esnd,tabulate(sim[,2]),xlab="Expected Out-events",ylab="Out-events")
abline(0,1,col=2)
plot(erec,tabulate(sim[,3]),xlab="Expected In-events",ylab="In-events")
abline(0,1,col=2)

#Keep the first 10 events of the simulated sequence, and produce 10 more
sim.pre<-sim[1:10,]
sim2<-simulate(fit.time,nsim=20,edgelist=sim.pre)
sim.pre                            #See the first 10 events
sim2                               #First 10 events preserved
all(sim2[1:10,]==sim.pre)          #All TRUE

#Repeat, but redrawing part of the input sequence
sim2.t<-simulate(fit.time,nsim=20,edgelist=sim.pre,redraw.timing=TRUE)
sim2.e<-simulate(fit.time,nsim=20,edgelist=sim.pre,redraw.events=TRUE)
sim2.t                             #Events kept, timings not
sim2.t[1:10,]==sim.pre             #Second two columns TRUE
sim2.e                             #Timing kept, events not
sim2.e[1:10,]==sim.pre             #(Note: some events may repeat by chance!)

## End(Not run)
```

Index