Package 'btergm'

January 29, 2017

Version 1.8.2

Date 2017-01-28

Title Temporal Exponential Random Graph Models by Bootstrapped Pseudolikelihood

Description Temporal Exponential Random Graph Models (TERGM) estimated by maximum pseudolikelihood with bootstrapped confidence intervals or Markov Chain Monte Carlo maximum likelihood. Goodness of fit assessment for ERGMs, TERGMs, and SAOMs. Micro-level interpretation of ERGMs and TERGMs.

URL http://github.com/leifeld/btergm

Imports stats4, utils, methods, graphics, statnet (>= 2015.11.0), statnet.common (>= 3.3.0), network (>= 1.13.0), sna (>= 2.3.2), ergm (>= 3.5.1), texreg (>= 1.34.4), parallel, Matrix (>= 1.2.2), boot (>= 1.3.17), coda (>= 0.18.1), stats, ROCR (>= 1.0.7), speedglm (>= 0.3.1), igraph (>= 0.7.1), RSiena (>= 1.0.12.232)

Depends R (>= 2.14.0), xergm.common (>= 1.7.3)

Suggests xergm

License GPL (>= 2)

NeedsCompilation no

Author Philip Leifeld [aut, cre], Skyler J. Cranmer [ctb], Bruce A. Desmarais [ctb]

Maintainer Philip Leifeld <philip.leifeld@glasgow.ac.uk>

Repository CRAN

Date/Publication 2017-01-29 10:16:11

R topics documented:

`btergm-package` *Temporal Exponential Random Graph Models by Bootstrapped Pseudolikelihood*

Description

Temporal Exponential Random Graph Models (TERGM) estimated by maximum pseudolikelihood with bootstrapped confidence intervals or Markov Chain Monte Carlo maximum likelihood. Goodness of fit assessment for ERGMs, TERGMs, and SAOMs. Micro-level interpretation of ERGMs and TERGMs.

Details

The **btergm** package implements TERGMs with MPLE and bootstrapped confidence intervals (btergm function) or MCMC MLE (mtergm function). Use the preprocess, adjust, handleMissings, and timecov functions for data preparation for TERGMs. Goodness of fit assessment for ERGMs, TERGMs, SAOMs, and dyadic independence models is possible with the generic gof function and its various methods. New networks can be simulated from TERGMs using the simulate.btergm function. The package also implements micro-level interpretation for ERGMs and TERGMs using the interpret function. Furthermore, the package contains the chemnet and knecht datasets for estimating (T)ERGMs. To display citation information, type `citation("btergm")`.

Author(s)

Philip Leifeld (http://www.philipleifeld.de)

Skyler J. Cranmer (http://www.skylercranmer.net)

Bruce A. Desmarais (http://polisci.la.psu.edu/people/bruce-desmarais)

See Also

btergm mtergm preprocess timecov simulate.btergm gof interpret btergm-class checkdegeneracy

btergm *TERGM by bootstrapped pseudolikelihood or MCMC MLE*

Description

TERGM by bootstrapped pseudolikelihood or MCMC MLE.

Usage

```
btergm(formula, R = 500, offset = FALSE, parallel = c("no",
    "multicore", "snow"), ncpus = 1, cl = NULL,
    verbose = TRUE, ...)

mtergm(formula, constraints = ~ ., verbose = TRUE, ...)
```

Arguments

formula	Formula for the TERGM. Model construction works like in the **ergm** package with the same model terms etc. (for a list of terms, see help("ergm-terms")). The networks to be modeled on the left-hand side of the equation must be given either as a list of network objects with more recent networks last (i.e., chronological order) or as a list of matrices with more recent matrices at the end. dyadcov and edgecov terms accept time-independent covariates (as network or matrix objects) or time-varying covariates (as a list of networks or matrices with the same length as the list of networks to be modeled).
R	Number of bootstrap replications. The higher the number of replications, the more accurate but also the slower is the estimation.
offset	If offset = TRUE is set, a list of offset matrices (one for each time step) with structural zeros is handed over to the pseudolikelihood routine. The offset matrices contain structural zeros where either the dependent networks or any of the covariates have missing nodes (if auto.adjust = TRUE is used). All matrices and network objects are inflated to the dimensions of the largest object, and the offset matrices inform the estimation routine which dyads are constrained to be absent (that is, fixed at -Inf). If offset = FALSE is set (the default behavior) and auto.adjust is switched on, all nodes that are not present across all covariates and networks within a time step are removed completely from the respective object(s) before estimation begins. This is computationally more efficient than using an offset matrix.
parallel	Use multiple cores in a computer or nodes in a cluster to speed up bootstrapping computations. The default value "no" means parallel computing is switched off. If "multicore" is used, the mclapply function from the **parallel** package (formerly in the **multicore** package) is used for parallelization. This should run on any kind of system except MS Windows because it is based on forking. It is usually the fastest type of parallelization. If "snow" is used, the parLapply function from the **parallel** package (formerly in the **snow** package) is used for parallelization. This should run on any kind of system including cluster systems

and including MS Windows. It is slightly slower than the former alternative if the same number of cores is used. However, "snow" provides support for MPI clusters with a large amount of cores, which **multicore** does not offer (see also the cl argument). The backend for the bootstrapping procedure is the **boot** package.

ncpus — The number of CPU cores used for parallel computing (only if parallel is activated). If the number of cores should be detected automatically on the machine where the code is executed, one can set ncpus = detectCores() after loading the **parallel** package. On some HPC clusters, the number of available cores is saved as an environment variable; for example, if MOAB is used, the number of available cores can sometimes be accessed using Sys.getenv("MOAB_PROCCOUNT"), depending on the implementation.

cl — An optional **parallel** or **snow** cluster for use if parallel = "snow". If not supplied, a PSOCK cluster is created temporarily on the local machine.

constraints — Constraints of the ERGM. See ergm for details.

verbose — Print details about data preprocessing and estimation settings.

... — Further arguments to be handed over to subroutines. For example, one can specify arguments for the ergm function (in the case of an mtergm call) or the boot function (in the case of a btergm call).

Details

The btergm function computes temporal exponential random graph models (TERGM) by bootstrapped pseudolikelihood, as described in Desmarais and Cranmer (2012).

The mtergm function computes TERGMs by MCMC MLE (or MPLE with uncorrected standard errors) via blockdiagonal matrices and structural zeros. The btergm function is faster than the mtergm function.

References

Cranmer, Skyler J., Tobias Heinrich and Bruce A. Desmarais (2014): Reciprocity and the Structural Determinants of the International Sanctions Network. *Social Networks* 36(1): 5–22. http://dx.doi.org/10.1016/j.socnet.2013.01.001.

Desmarais, Bruce A. and Skyler J. Cranmer (2012): Statistical Mechanics of Networks: Estimation and Uncertainty. *Physica A* 391: 1865–1876. http://dx.doi.org/10.1016/j.physa.2011.10.018.

Desmarais, Bruce A. and Skyler J. Cranmer (2010): Consistent Confidence Intervals for Maximum Pseudolikelihood Estimators. *Neural Information Processing Systems 2010 Workshop on Computational Social Science and the Wisdom of Crowds*.

See Also

btergm-package simulate.btergm gof knecht btergm-class preprocess timecov checkdegeneracy

Examples

```
# A simple toy example:

library("statnet")
set.seed(5)

networks <- list()
for(i in 1:10){            # create 10 random networks with 10 actors
  mat <- matrix(rbinom(100, 1, .25), nrow = 10, ncol = 10)
  diag(mat) <- 0           # loops are excluded
  nw <- network(mat)       # create network object
  networks[[i]] <- nw      # add network to the list
}

covariates <- list()
for (i in 1:10) {          # create 10 matrices as covariate
  mat <- matrix(rnorm(100), nrow = 10, ncol = 10)
  covariates[[i]] <- mat   # add matrix to the list
}

fit <- btergm(networks ~ edges + istar(2) +
    edgecov(covariates), R = 100)

summary(fit)               # show estimation results

## Not run:
# The same example using MCMC MLE:

fit2 <- mtergm(networks ~ edges + istar(2) +
    edgecov(covariates))

summary(fit2)

## End(Not run)

# For an example with real data, see help("knecht").


# Examples for parallel processing:

# Some preliminaries:
# - "Forking" means running the code on multiple cores in the same
#   computer. It's fast but consumes a lot of memory because all
#   objects are copied for each node. It's also restricted to
#   cores within a physical computer, i.e. no distribution over a
#   network or cluster. Forking does not work on Windows systems.
# - "MPI" is a protocol for distributing computations over many
#   cores, often across multiple physical computers/nodes. MPI
#   is fast and can distribute the work across hundreds of nodes
#   (but remember that R can handle a maximum of 128 connections,
#   which includes file access and parallel connections). However,
#   it requires that the Rmpi package is installed and that an MPI
```

```
#   server is running (e.g., OpenMPI).
# - "PSOCK" is a TCP-based protocol. It can also distribute the
#   work to many cores across nodes (like MPI). The advantage of
#   PSOCK is that it can as well make use of multiple nodes within
#   the same node or desktop computer (as with forking) but without
#   consuming too much additional memory. However, the drawback is
#   that it is not as fast as MPI or forking.
# The following code provides examples for these three scenarios.

# btergm works with clusters via the parallel package. That is, the
# user can create a cluster object (of type "PSOCK", "MPI", or
# "FORK") and supply it to the 'cl' argument of the 'btergm'
# function. If no cluster object is provided, btergm will try to
# create a temporary PSOCK cluster (if parallel = "snow") or it
# will use forking (if parallel = "multicore").

## Not run:

# To use a PSOCK cluster without providing an explicit cluster
# object:
require("parallel")
fit <- btergm(networks ~ edges + istar(2) + edgecov(covariates),
    R = 100, parallel = "snow", ncpus = 25)

# Equivalently, a PSOCK cluster can be provided as follows:
require("parallel")
cores <- 25
cl <- makeCluster(cores, type = "PSOCK")
fit <- btergm(networks ~ edges + istar(2) + edgecov(covariates),
    R = 100, parallel = "snow", ncpus = cores, cl = cl)
stopCluster(cl)

# Forking (without supplying a cluster object) can be used as
# follows.
require("parallel")
cores <- 25
fit <- btergm(networks ~ edges + istar(2) + edgecov(covariates),
    R = 100, parallel = "multicore", ncpus = cores)
stopCluster(cl)

# Forking (by providing a cluster object) works as follows:
require("parallel")
cores <- 25
cl <- makeCluster(cores, type = "FORK")
fit <- btergm(networks ~ edges + istar(2) + edgecov(covariates),
    R = 100, parallel = "snow", ncpus = cores, cl = cl)
stopCluster(cl)

# To use MPI, a cluster object MUST be created beforehand. In
# this example, a MOAB HPC server is used. It stores the number of
# available cores as a system option:
require("parallel")
cores <- as.numeric(Sys.getenv("MOAB_PROCCOUNT"))
```

```
cl <- makeCluster(cores, type = "MPI")
fit <- btergm(networks ~ edges + istar(2) + edgecov(covariates),
    R = 100, parallel = "snow", ncpus = cores, cl = cl)
stopCluster(cl)

# In the following example, the Rmpi package is used to create a
# cluster. This may not work on all systems; consult your local
# support staff or the help files on your HPC server to find out how
# to create a cluster object on your system.

# snow/Rmpi start-up
if (!is.loaded("mpi_initialize")) {
    library("Rmpi")
}
library(snow);

mpirank <- mpi.comm.rank (0)
if (mpirank == 0) {
   invisible(makeMPIcluster())
} else {
  sink (file="/dev/null")
  invisible(slaveLoop (makeMPImaster()))
  mpi.finalize()
  q()
}
# End snow/Rmpi start-up

cl <- getMPIcluster()

fit <- btergm(networks ~ edges + istar(2) + edgecov(covariates),
    R = 100, parallel = "snow", ncpus = 25, cl = cl)


## End(Not run)
```

---

btergm-class *Classes* "btergm" *and* "mtergm"

Description

btergm objects result from the estimation of a bootstrapped TERGM via the btergm function in the **xergm** package. btergm objects contain the coefficients, the bootstrapping samples of the coefficients, the number of replications, the number of observations, the number of time steps, the original formula, and the response, effects and weights objects that were fed into the glm call for estimating the model. mtergm objects result from MCMC-MLE-based estimation of a TERGM via the mtergm function. They contain the coefficients, standard errors, and p-values, among other details.

Usage

```
## S4 method for signature 'btergm'
summary(object, level = 0.95, type = "perc",
    invlogit = FALSE, ...)

## S4 method for signature 'mtergm'
summary(object, ...)

## S4 method for signature 'btergm'
show(object)

## S4 method for signature 'mtergm'
show(object)

## S4 method for signature 'btergm'
nobs(object)

## S4 method for signature 'mtergm'
nobs(object)

## S4 method for signature 'btergm'
coef(object, invlogit = FALSE, ...)

## S4 method for signature 'mtergm'
coef(object, invlogit = FALSE, ...)

## S4 method for signature 'btergm'
confint(object, parm, level = 0.95, type = "perc",
    invlogit = FALSE, ...)

btergm.se(object, print = FALSE)

timesteps.btergm(object)

timesteps.mtergm(object)
```

Arguments

object	A btergm object.
level	The significance level for computation of the confidence intervals. The default is 0.95 (that is, an alpha value of 0.05). Other common values include 0.999, 0.99, 0.9, and 0.5.
parm	Parameters (specified by integer position or character string).
type	Type of confidence interval, e.g., basic bootstrap interval (type = "basic"), percentile-based interval (type = "perc", which is the default option), or bias-adjusted and accelerated confidence interval (type = "bca"). All options from the type argument of the boot.ci function in the boot package can be used to generate confidence intervals.

invlogit	Apply inverse logit transformation to the estimates and/or confidence intervals? That is, `1 / (1 + exp(-x))`, where `x` is the respective value.
print	Should the formatted coefficient table be printed to the R console along with significance stars (`print = TRUE`), or should the plain coefficient matrix be returned (`print = FALSE`)?
...	Further arguments to be handed over to subroutines.

Details

Various generic methods are available for `btergm` objects: The `coef` and `show` methods return the coefficients; the `summary` method gives a model summary. The `nobs` method returns the number of observations. The confint method returns confidence intervals from the bootstrap replications of `btergm` objects, and the user can specify the confidence level. The method returns a matrix with three columns: the estimate, the lower bound, and the upper bound of the confidence interval for each model term.

The `btergm.se` function computes standard errors and p values for `btergm` objects. It returns a matrix with four columns: the estimate, the standard error, the z value, and the p value for each model term. If the argument `print = TRUE` is used, the matrix is printed to the R console as a formatted coefficient matrix with significance stars instead. Note that confidence intervals are the preferred way of interpretation for bootstrapped TERGMs; standard errors are only accurate if the bootstrapped data are normally distributed, which is not always the case. Various methods for checking for normality for each model term are available, for example quantile-quantile plots (e.g., `qqnorm(x@boot$t[, 1])` for the first model term in the `btergm` object called `x`).

The `timesteps.btergm` function extracts the number of time steps from a `btergm` object. The number of time steps is the number of networks being modeled on the left-hand side of the model formula.

Some of these functions or methods are also available for `mtergm` objects.

Slots

`coef`**:** Object of class `"numeric"`. The coefficients.

`bootsamp`**:** Object of class `"matrix"`. The bootstrapping sample.

`R`**:** Object of class `"numeric"`. Number of replications.

`nobs`**:** Object of class `"numeric"`. Number of observations.

`time.steps`**:** Object of class `"numeric"`. Number of time steps.

`formula`**:** Object of class `"formula"`. The original model formula (without indices for the time steps).

`response`**:** Object of class `"integer"`. The response variable.

`effects`**:** Object of class `"data.frame"`. The effects that went into the `glm` call.

`weights`**:** Object of class `"integer"`. The weights of the observations.

`auto.adjust`**:** Object of class `"logical"`. Indicates whether automatic adjustment of dimensions was done before estimation.

`offset`**:** Object of class `"logical"`. Indicates whether an offset matrix with structural zeros was used.

`directed`**:** Object of class `"logical"`. Are the dependent networks directed?

`bipartite`**:** Object of class `"logical"`. Are the dependent networks bipartite?

se Standard errors.

pval p-values.

estimate Estimate: either MCMC MLE or MPLE.

loglik The log likelihood.

aic Akaike's Information Criterion (AIC).

bic The Bayesian Information Criterion (BIC).

References

Cranmer, Skyler J., Tobias Heinrich and Bruce A. Desmarais (2014): Reciprocity and the Structural Determinants of the International Sanctions Network. *Social Networks* 36(1): 5–22. http://dx.doi.org/10.1016/j.socnet.2013.01.001.

Desmarais, Bruce A. and Skyler J. Cranmer (2012): Statistical Mechanics of Networks: Estimation and Uncertainty. *Physica A* 391: 1865–1876. http://dx.doi.org/10.1016/j.physa.2011.10.018.

Desmarais, Bruce A. and Skyler J. Cranmer (2010): Consistent Confidence Intervals for Maximum Pseudolikelihood Estimators. *Neural Information Processing Systems 2010 Workshop on Computational Social Science and the Wisdom of Crowds*.

See Also

btergm-package btergm simulate.btergm gof knecht getformula interpret mtergm

---

checkdegeneracy *Degeneracy check for btergm and mtergm objects*

Description

Assess degeneracy of btergm and mtergm models.

Usage

```
## S4 method for signature 'mtergm'
checkdegeneracy(object, ...)

## S4 method for signature 'btergm'
checkdegeneracy(object, nsim = 1000,
    MCMC.interval = 1000, MCMC.burnin = 10000, verbose = FALSE)

## S3 method for class 'degeneracy'
print(x, center = FALSE, t = 1:length(x$sim),
    terms = 1:length(x$target.stats[[1]]), ...)
```

```
## S3 method for class 'degeneracy'
plot(x, center = TRUE, t = 1:length(x$sim),
    terms = 1:length(x$target.stats[[1]]), vbar = TRUE,
    main = NULL, xlab = NULL, target.col = "red",
    target.lwd = 3, ...)
```

Arguments

object	A btergm or mtergm object, as estimated using the btergm or mtergm function.
nsim	The number of networks to be simulated at each time step. This number should be sufficiently large for a meaningful comparison. If possible, much more than 1,000 simulations.
MCMC.burnin	Internally, this package uses the simulation facilities of the **ergm** package to create new networks against which to compare the original network(s) for goodness-of-fit assessment. This argument sets the MCMC burnin to be passed over to the simulation command. The default value is 10000. There is no general rule of thumb on the selection of this parameter, but if the results look suspicious (e.g., when the model fit is perfect), increasing this value may be helpful.
MCMC.interval	Internally, this package uses the simulation facilities of the **ergm** package to create new networks against which to compare the original network(s) for goodness-of-fit assessment. This argument sets the MCMC interval to be passed over to the simulation command. The default value is 1000, which means that every 1000th simulation outcome from the MCMC sequence is used. There is no general rule of thumb on the selection of this parameter, but if the results look suspicious (e.g., when the model fit is perfect), increasing this value may be helpful.
verbose	Print details?
x	A degeneracy object created by the checkdegeneracy function.
center	If TRUE, print/plot the simulated minus the target statistics, with an expected value of 0 in a non-degenerate model. If FALSE, print/plot the distribution of simulated statistics and show the target statistic separately.
t	Time indices to include, e.g., t = 2:4 for time steps 2 to 4.
terms	Indices of the model terms to include, e.g., terms = 1:3 includes the first three statistics.
vbar	Show vertical bar for target statistic in histogram.
main	Main title of the plot.
xlab	Label on the x-axis. Defaults to the name of the statistic.
target.col	Color of the vertical bar for the target statistic. Defaults to red.
target.lwd	Line width of the vertical bar for the target statistic. Defaults to 3.
...	Arbitrary further arguments.

Details

The methods for the generic degeneracy function implement a degeneracy check for btergm and mtergm objects. For btergm, this works by comparing the global statistics of simulated networks to those of the observed networks at each observed time step. If the global statistics differ significantly, this is indicated by small p-values. If there are many significant results, this indicates degeneracy. For mtergm, the mcmc.diagnostics function from the **ergm** package is used.

Author(s)

Philip Leifeld (http://www.philipleifeld.com)

See Also

btergm-package btergm gof

---

edgeprob *Compute all dyadic edge probabilities for an ERGM or TERGM.*

Description

edgeprob is a convenience function that creates a data frame with all dyads in the ERGM or TERGM along with their edge probabilities and their predictor values (i.e., change statistics). This is useful for creating marginal effects plots or contrasting multiple groups of dyads. This function works faster than the interpret function. See also the interpret help page.

Usage

```
edgeprob(object, verbose = FALSE)
```

Arguments

object	An ergm, btergm, or mtergm object.
verbose	Print details?

Value

The first variable in the resulting data frame contains the edge value (i.e., the dependent variable, which is usually binary). The next variables contain all the predictors from the ERGM or TERGM (i.e., the change statistics). The next three variables contain the indices of the sender (i), the receiver (j), and the time step (t). These three indices serve to identify the dyad. The last variable contains the computed edge probabilities.

Author(s)

Philip Leifeld

See Also

interpret btergm-package btergm

`getformula` *Extract the formula from a model.*

Description

Extract the model formula from ergm or btergm objects.

Usage

```
## S4 method for signature 'ergm'
getformula(x)

## S4 method for signature 'btergm'
getformula(x)

## S4 method for signature 'mtergm'
getformula(x)
```

Arguments

`x`	A model object, for example a `btergm` or an `ergm` object.

Details

Extract the model formula from ergm or btergm objects.

See Also

gof

`gof-methods` *Conduct Goodness-of-Fit Diagnostics on ERGMs, TERGMs, SAOMs, and logit models*

Description

Assess goodness of fit of btergm and other network models.

Usage

```
## S4 method for signature 'btergm'
gof(object, target = NULL, formula = getformula(object),
    nsim = 100, MCMC.interval = 1000, MCMC.burnin = 10000,
    parallel = c("no", "multicore", "snow"), ncpus = 1, cl = NULL,
    statistics = c(dsp, esp, deg, ideg, geodesic, rocpr,
    walktrap.modularity), verbose = TRUE, ...)

## S4 method for signature 'mtergm'
gof(object, target = NULL, formula = getformula(object),
    nsim = 100, MCMC.interval = 1000, MCMC.burnin = 10000,
    parallel = c("no", "multicore", "snow"), ncpus = 1, cl = NULL,
    statistics = c(dsp, esp, deg, ideg, geodesic, rocpr,
    walktrap.modularity), verbose = TRUE, ...)

## S4 method for signature 'ergm'
gof(object, target = NULL, formula = getformula(object),
    nsim = 100, MCMC.interval = 1000, MCMC.burnin = 10000,
    parallel = c("no", "multicore", "snow"), ncpus = 1, cl = NULL,
    statistics = c(dsp, esp, deg, ideg, geodesic, rocpr,
    walktrap.modularity), verbose = TRUE, ...)

## S4 method for signature 'matrix'
gof(object, covariates, coef, target = NULL, nsim = 100,
    mcmc = FALSE, MCMC.interval = 1000, MCMC.burnin = 10000,
    parallel = c("no", "multicore", "snow"), ncpus = 1, cl = NULL,
    statistics = c(dsp, esp, deg, ideg, geodesic, rocpr,
    walktrap.modularity), verbose = TRUE, ...)

## S4 method for signature 'network'
gof(object, covariates, coef, target = NULL,
    nsim = 100, mcmc = FALSE, MCMC.interval = 1000,
    MCMC.burnin = 10000, parallel = c("no", "multicore", "snow"),
    ncpus = 1, cl = NULL, statistics = c(dsp, esp, deg, ideg,
    geodesic, rocpr, walktrap.modularity), verbose = TRUE, ...)

## S4 method for signature 'sienaFit'
gof(object, period = NULL, parallel = c("no",
    "multicore", "snow"), ncpus = 1, cl = NULL, structzero = 10,
    statistics = c(esp, deg, ideg, geodesic, rocpr, walktrap.modularity),
    groupName = object$f$groupNames[[1]], varName = NULL,
    outofsample = FALSE, sienaData = NULL, sienaEffects = NULL,
    nsim = NULL, verbose = TRUE, ...)
```

Arguments

`cl`	An optional **parallel** or **snow** cluster for use if `parallel = "snow"`. If not supplied, a cluster on the local machine is created temporarily.

`coef` A vector of coefficients.

`covariates` A list of matrices or network objects that serve as covariates for the dependent network. The covariates in this list are automatically added to the formula as `edgecov` terms.

`formula` A model formula from which networks are simulated for comparison. By default, the formula from the `btergm` object `x` is used. It is possible to hand over a formula with only a single response network and/or dyad or edge covariates or with lists of response networks and/or covariates. It is also possible to use indices like `networks[[4]]` or `networks[3:5]` inside the formula.

`groupName` The group name used in the Siena model.

`mcmc` Should statnet's MCMC methods be used for simulating new networks? If `mcmc = FALSE`, new networks are simulated based on predicted tie probabilities of the regression equation.

`MCMC.burnin` Internally, this package uses the simulation facilities of the **ergm** package to create new networks against which to compare the original network(s) for goodness-of-fit assessment. This argument sets the MCMC burnin to be passed over to the simulation command. The default value is `10000`. There is no general rule of thumb on the selection of this parameter, but if the results look suspicious (e.g., when the model fit is perfect), increasing this value may be helpful.

`MCMC.interval` Internally, this package uses the simulation facilities of the **ergm** package to create new networks against which to compare the original network(s) for goodness-of-fit assessment. This argument sets the MCMC interval to be passed over to the simulation command. The default value is `1000`, which means that every 1000th simulation outcome from the MCMC sequence is used. There is no general rule of thumb on the selection of this parameter, but if the results look suspicious (e.g., when the model fit is perfect), increasing this value may be helpful.

`ncpus` The number of CPU cores used for parallel GOF assessment (only if `parallel` is activated). If the number of cores should be detected automatically on the machine where the code is executed, one can try the `detectCores()` function from the **parallel** package. On some HPC clusters, the number of available cores is saved as an environment variable; for example, if MOAB is used, the number of available cores can sometimes be accessed using `Sys.getenv("MOAB_PROCCOUNT")`, depending on the implementation. Note that the maximum number of connections in a single R session (i.e., to other cores or for opening files etc.) is 128, so fewer than 128 cores should be used at a time.

`nsim` The number of networks to be simulated at each time step. Example: If there are six time steps in the `formula` and `nsim = 100`, a total of 600 new networks is simulated. The comparison between simulated and observed networks is only done within time steps. For example, the first 100 simulations are compared with the first observed network, simulations 101-200 with the second observed network etc.

`object` A `btergm`, `ergm`, or `sienaFit` object (for the `btergm`, `ergm`, and `sienaFit` methods, respectively). Or a network object or matrix (for the `network` and `matrix` methods, respectively).

`outofsample` Should out-of-sample prediction be attempted? If so, some additional arguments must be provided: `sienaData`, `sienaEffects`, and `nsim`. The `sienaData` object must contain a base and a target network for out-of-sample prediction. The `sienaEffects` must contain the effects to be used for the simulations. The estimates will be taken from the estimated `object`, and they will be injected into a new SAOM and fixed during the sampling procedure. `nsim` determines how many simulations are used for the out-of-sample comparison.

`parallel` Use multiple cores in a computer or nodes in a cluster to speed up the simulations. The default value `"no"` means parallel computing is switched off. If `"multicore"` is used (only available for `sienaAlgorithm` and `sienaModel` objects), the `mclapply` function from the **parallel** package (formerly in the **multicore** package) is used for parallelization. This should run on any kind of system except MS Windows because it is based on forking. It is usually the fastest type of parallelization. If `"snow"` is used, the `parLapply` function from the **parallel** package (formerly in the **snow** package) is used for parallelization. This should run on any kind of system including cluster systems and including MS Windows. It is slightly slower than the former alternative if the same number of cores is used. However, `"snow"` provides support for MPI clusters with a large amount of cores, which **multicore** does not offer (see also the `cl` argument). Note that `"multicore"` will only work if all cores are on the same node. For example, if there are three nodes with eight cores each, a maximum of eight CPUs can be used. Parallel computing is described in more detail on the help page of btergm.

`period` Which transition between time periods should be used for GOF assessment? By default, all transitions between all time periods are used. For example, if there are three consecutive networks, this will extract simulations from the transitions between 1 and 2 and between 2 and 3, respectively, and these simulations will be compared to the networks at time steps 2 and 3, respectively. The time period can be provided as a numeric, e.g., `period = 4` for extracting the simulations between time steps 4 and 5 (= the fourth transition) and predicting the fifth network. Values lower than 1 or larger than the number of consecutive networks minus 1 are therefore not permitted. This argument is only used if out-of-sample prediction is switched off.

`sienaData` An object of the class `siena`, which is usually created using the `sienaDataCreate` function in the **RSiena** package. This argument is only used for out-of-sample prediction. The object must be based on a `sienaDependent` object that contains two networks: the base network from which to simulate forward, and the target network which you want to predict out-of-sample. The object can contain further objects for storing covariates etc. that are necessary for estimating new networks. The best practice is to create an object that is identical to the `siena` object used for estimating the model, except that it contains the base and the target network instead of the dependent variable/networks.

`sienaEffects` An object of the class `sienaEffects`, which is usually created using the `getEffects()` and the `includeEffects()` functions in the `RSiena` package. The best practice is to provide a `sienaEffects` object that is identical to the object used to create the original model (that is, it should contain the same effects), except that it should be based on the `siena` object provided through the `sienaData` argument. In other words, the `sienaEffects` object should be based on the base and target network used for out-of-sample prediction, and it should contain the same

effects as those used for the original estimation. This argument is used only for out-of-sample prediction.

`statistics` A list of functions used for comparison of observed and simulated networks. Note that the list should contain the actual functions, not a character representation of them. See gof-statistics for details.

`target` A network or list of networks to which the simulations are compared. If left empty, the original networks from the `btergm` object `x` are used as observed networks.

`structzero` Which value was used for structural zeros (usually nodes that have dropped out of the network or have not yet joined the network) in the dependent variable/network? These nodes are removed from the observed network and the simulations before comparison. Usually, the value `10` is used for structural zeros in Siena.

`varName` The variable name that denotes the dependent networks in the Siena model.

`verbose` Print details?

`...` Arbitrary further arguments.

Details

The generic `gof` function provides goodness-of-fit measures and degeneracy checks for `btergm`, `mtergm`, `ergm`, `sienaFit`, and custom dyadic-independent models. The user can provide a list of network statistics for comparing simulated networks based on the estimated model with the observed network(s). See gof-statistics. The objects created by these methods can be displayed using various plot and print methods (see gof-plot).

In-sample GOF assessment is the default, which means that the same time steps are used for creating simulations and for comparison with the observed network(s). It is possible to do out-of-sample prediction by specifying a (list of) target network(s) using the `target` argument. If a formula is provided, the simulations are based on the networks and covariates specified in the formula. This is helpful in situations where complex out-of-sample predictions have to be evaluated. A usage scenario could be to simulate from a network at time `t` (provided through the `formula` argument) and compare to an observed network at time `t + 1` (the `target` argument). This can be done, for example, to assess predictive performance between time steps of the original networks, or to check whether the model performs well with regard to a newly measured network given the old data from the previous time step.

Predictive fit can also be assessed for stochastic actor-oriented models (SAOM) as implemented in the **RSiena** package. After compiling the usual objects (model, data, effects), one of the time steps can be predicted based on the previous time step and the SAOM using the `sienaFit` method of the `gof` function. By default, however, within-sample fit is used for SAOMs, just like for (T)ERGMs.

The `gof` methods for networks and matrices serve to assess the goodness of fit of a dyadic-independence model. To do this, the method requires a vector of coefficients (one coefficient for the intercept or `edges` term and one coefficient for each covariate), a list of covariates (in matrix or network shape), and a dependent network or matrix. This is useful for assessing the goodness of fit of QAP-adjusted logistic regression models (as implemented in the `netlogit` function in the **sna** package) or other dyadic-independence models, such as models fitted using `glm`. Note that this method only works with cross-sectional models and does not accept lists of networks as input data.

Author(s)

Philip Leifeld (http://www.philipleifeld.com)

See Also

btergm-package btergm simulate.btergm simulate.formula gof gof-statistics gof-plot

Examples

```
## Not run:
# First, create data and fit a TERGM...
networks <- list()
for(i in 1:10){            # create 10 random networks with 10 actors
  mat <- matrix(rbinom(100, 1, .25), nrow = 10, ncol = 10)
  diag(mat) <- 0           # loops are excluded
  nw <- network(mat)       # create network object
  networks[[i]] <- nw      # add network to the list
}

covariates <- list()
for (i in 1:10) {          # create 10 matrices as covariate
  mat <- matrix(rnorm(100), nrow = 10, ncol = 10)
  covariates[[i]] <- mat   # add matrix to the list
}

fit <- btergm(networks ~ edges + istar(2) +
    edgecov(covariates), R = 100)

# Then assess the goodness of fit:
g <- gof(fit, statistics = c(triad.directed, esp, maxmod.modularity,
    rocpr), nsim = 50)
g
plot(g)  # see ?"gof-plot" for details

## End(Not run)
```

---

gof-plot *Plot and print methods for gof output.*

Description

Plot and print methods for goodness-of-fit output for network models.

Usage

```
## S3 method for class 'boxplot'
plot(x, relative = TRUE, transform = function(x) x,
    xlim = NULL, main = x$label, xlab = x$label, ylab = "Frequency",
    border = "darkgray", boxplot.lwd = 0.8, outline = FALSE,
```

```
    median = TRUE, median.col = "black", median.lty = "solid",
    median.lwd = 2, mean = TRUE, mean.col = "black",
    mean.lty = "dashed", mean.lwd = 1, ...)

## S3 method for class 'gof'
plot(x, mfrow = TRUE, ...)

## S3 method for class 'pr'
plot(x, add = FALSE, main = x$label, avg = c("none",
    "horizontal", "vertical", "threshold"), spread.estimate =
    c("boxplot", "stderror", "stddev"), lwd = 3, rgraph = TRUE,
    col = "#5886be", random.col = "#5886be44", pr.poly = 0, ...)

## S3 method for class 'roc'
plot(x, add = FALSE, main = x$label, avg = c("none",
    "horizontal", "vertical", "threshold"), spread.estimate =
    c("boxplot", "stderror", "stddev"), lwd = 3, rgraph = TRUE,
    col = "#bd0017", random.col = "#bd001744", ...)

## S3 method for class 'rocpr'
plot(x, main = x$label, roc.avg = c("none",
    "horizontal", "vertical", "threshold"),
    roc.spread.estimate = c("boxplot", "stderror", "stddev"),
    roc.lwd = 3, roc.rgraph = TRUE, roc.col = "#bd0017",
    roc.random.col = "#bd001744", pr.avg = c("none", "horizontal",
    "vertical", "threshold"), pr.spread.estimate = c("boxplot",
    "stderror", "stddev"), pr.lwd = 3, pr.rgraph = TRUE,
    pr.col = "#5886be", pr.random.col = "#5886be44", pr.poly = 0,
    ...)

## S3 method for class 'univariate'
plot(x, main = x$label, sim.hist = TRUE,
    sim.bar = TRUE, sim.density = TRUE, obs.hist = FALSE,
    obs.bar = TRUE, obs.density = TRUE, sim.adjust = 1,
    obs.adjust = 1, sim.lwd = 2, obs.lwd = 2, sim.col = "black",
    obs.col = "red", ...)

## S3 method for class 'boxplot'
print(x, ...)

## S3 method for class 'gof'
print(x, ...)

## S3 method for class 'pr'
print(x, ...)

## S3 method for class 'roc'
print(x, ...)
```

```
## S3 method for class 'rocpr'
print(x, ...)

## S3 method for class 'univariate'
print(x, ...)
```

Arguments

add	Add the ROC and/or PR curve to an existing plot?
avg	Averaging pattern for the ROC and PR curve(s) if multiple target time steps were used. Allowed values are "none" (plot all curves separately), "horizontal" (horizontal averaging), "vertical" (vertical averaging), and "threshold" (threshold (= cutoff) averaging). Note that while threshold averaging is always feasible, vertical and horizontal averaging are not well-defined if the graph cannot be represented as a function x->y and y->x, respectively. More information can be obtained from the help pages of the **ROCR** package, the functions of which are employed here.
border	Color of the borders of the boxplots.
boxplot.lwd	Line width of boxplot.
col	Color of the ROC or PR curve.
lwd	Line width.
main	Main title of a GOF plot.
mean	Plot the mean curve for the observed network?
mean.col	Color of the mean of the observed network statistic.
mean.lty	Line type of mean line. For example "dashed" or "solid".
mean.lwd	Line width of mean line.
median	Plot the median curve for the observed network?
median.col	Color of the median of the observed network statistic.
median.lty	Line type of median line. For example "dashed" or "solid".
median.lwd	Line width of median line.
mfrow	Should the GOF plots come out separately (mfrow = FALSE), or should all statistics be aligned in a single diagram (mfrow = TRUE)? Returning the plots separately can be helpful if the output is redirected to a multipage PDF or TIFF file.
obs.adjust	Bandwidth adjustment parameter for the density curve.
obs.bar	Draw a bar for the median of the statistic for the observed networks?
obs.col	Color for the observed network(s).
obs.density	Draw a density curve fot the statistic for the observed networks?
obs.hist	Draw a histogram for the observed networks?
obs.lwd	Line width for the observed network(s).
outline	Print outliers in the boxplots?

`pr.avg` Averaging pattern for the PR curve(s) if multiple target time steps were used. Allowed values are `"none"` (plot all curves separately), `"horizontal"` (horizontal averaging), `"vertical"` (vertical averaging), and `"threshold"` (threshold (= cutoff) averaging). Note that while threshold averaging is always feasible, vertical and horizontal averaging are not well-defined if the graph cannot be represented as a function x->y and y->x, respectively. More information can be obtained from the help pages of the **ROCR** package, the functions of which are employed here.

`pr.col` Color of the PR curve.

`pr.lwd` Line width.

`pr.poly` If a value of `0` is set, nothing special happens. If a value of `1` is set, a straight line is fitted through the PR curve and displayed. Values between `2` and `9` fit higher-order polynomial curves through the PR curve and display the resulting curve. This argument allows to check whether the imputation of the first precision value in the PR curve yielded a reasonable result (in case the value had to be imputed).

`pr.random.col` Color of the PR curve of the random graph prediction.

`pr.rgraph` Should an PR curve also be drawn for a random graph? This serves as a baseline against which to compare the actual PR curve.

`pr.spread.estimate`
When multiple target time steps are used and curve averaging is enabled, the variation around the average curve can be visualized as standard error bars (`"stderror"`), standard deviation bars (`"stddev"`), or by using box plots (`"boxplot"`). Note that the function plotCI, which is used internally by the **ROCR** package to draw error bars, might raise a warning if the spread of the curves at certain positions is 0. More details can be found in the documentation of the **ROCR** package, the functions of which are employed here.

`random.col` Color of the ROC or PR curve of the random graph prediction.

`relative` Print relative frequencies (as opposed to absolute frequencies) of a statistic on the y axis?

`rgraph` Should an ROC or PR curve also be drawn for a random graph? This serves as a baseline against which to compare the actual ROC or PR curve.

`roc.avg` Averaging pattern for the ROC curve(s) if multiple target time steps were used. Allowed values are `"none"` (plot all curves separately), `"horizontal"` (horizontal averaging), `"vertical"` (vertical averaging), and `"threshold"` (threshold (= cutoff) averaging). Note that while threshold averaging is always feasible, vertical and horizontal averaging are not well-defined if the graph cannot be represented as a function x->y and y->x, respectively. More information can be obtained from the help pages of the **ROCR** package, the functions of which are employed here.

`roc.col` Color of the ROC curve.

`roc.lwd` Line width.

`roc.random.col` Color of the ROC curve of the random graph prediction.

`roc.rgraph` Should an ROC curve also be drawn for a random graph? This serves as a baseline against which to compare the actual ROC curve.

`roc.spread.estimate`
: When multiple target time steps are used and curve averaging is enabled, the variation around the average curve can be visualized as standard error bars (`"stderror"`), standard deviation bars (`"stddev"`), or by using box plots (`"boxplot"`). Note that the function plotCI, which is used internally by the **ROCR** package to draw error bars, might raise a warning if the spread of the curves at certain positions is 0. More details can be found in the documentation of the **ROCR** package, the functions of which are employed here.

`sim.adjust`
: Bandwidth adjustment parameter for the density curve.

`sim.bar`
: Draw a bar for the median of the statistic for the simulated networks?

`sim.col`
: Color for the simulated networks.

`sim.density`
: Draw a density curve fot the statistic for the simulated networks?

`sim.hist`
: Draw a histogram for the simulated networks?

`sim.lwd`
: Line width for the simulated networks.

`spread.estimate`
: When multiple target time steps are used and curve averaging is enabled, the variation around the average curve can be visualized as standard error bars (`"stderror"`), standard deviation bars (`"stddev"`), or by using box plots (`"boxplot"`). Note that the function plotCI, which is used internally by the **ROCR** package to draw error bars, might raise a warning if the spread of the curves at certain positions is 0. More details can be found in the documentation of the **ROCR** package, the functions of which are employed here.

`transform`
: A function which transforms the y values used for the boxplots. For example, if some of the values become very large and make the output illegible, `transform = function(x) x^0.1` or a similar transformation of the values can be used. Note that logarithmic transformations often produce infinite values because `log(0) = -Inf`, so one should rather use something like `transform = function(x) log1p` to avoid infinite values.

`x`
: An object created by one of the `gof` methods.

`xlab`
: Label of the x-axis of a GOF plot.

`xlim`
: Horizontal limit of the boxplots. Only the maximum value must be provided, e.g., `xlim = 8`.

`ylab`
: Label of the y-axis of a GOF plot.

`...`
: Arbitrary further arguments.

Details

These plot and print methods serve to display the output generated by the `gof` function and its methods. See the help page of gof-methods for details on how to compute gof.

Author(s)

Philip Leifeld (`http://www.philipleifeld.com`)

See Also

btergm-package gof gof-methods gof-statistics

gofstatistics *Statistics for goodness-of-fit assessment of network models*

Description

Statistics for goodness-of-fit assessment of network models.

Usage

```
b1deg(mat)

b1star(mat)

b2deg(mat)

b2star(mat)

comemb(vec)

deg(mat)

dsp(mat)

edgebetweenness.modularity(mat)

edgebetweenness.pr(sim, obs)

edgebetweenness.roc(sim, obs)

esp(mat)

fastgreedy.modularity(mat)

fastgreedy.pr(sim, obs)

fastgreedy.roc(sim, obs)

geodesic(mat)

ideg(mat)

istar(mat)

kcycle(mat)

kstar(mat)
```

```
maxmod.modularity(mat)

maxmod.pr(sim, obs)

maxmod.roc(sim, obs)

nsp(mat)

odeg(mat)

ostar(mat)

pr(sim, obs)

roc(sim, obs)

rocpr(sim, obs)

rocprgof(sim, obs, pr.impute = "poly4")

spinglass.modularity(mat)

spinglass.pr(sim, obs)

spinglass.roc(sim, obs)

triad.directed(mat)

triad.undirected(mat)

walktrap.modularity(mat)

walktrap.pr(sim, obs)

walktrap.roc(sim, obs)
```

Arguments

vec	A vector of community memberships in order to create a community co-membership matrix.
mat	A sparse network matrix as created by the Matrix function in the **Matrix** package.
sim	A list of simulated networks. Each element in the list should be a sparse matrix as created by the Matrix function in the **Matrix** package.
obs	A list of observed (= target) networks. Each element in the list should be a sparse matrix as created by the Matrix function in the **Matrix** package.
pr.impute	In some cases, the first precision value of the precision-recall curve is undefined. The pr.impute argument serves to impute this missing value to ensure that the

AUC-PR value is not severely biased. Possible values are "no" for no imputation, "one" for using a value of 1.0, "second" for using the next (= adjacent) precision value, "poly1" for fitting a straight line through the remaining curve to predict the first value, "poly2" for fitting a second-order polynomial curve etc. until "poly9". Warning: this is a pragmatic solution. Please double-check whether the imputation makes sense. This can be checked by plotting the resulting object and using the pr.poly argument to plot the predicted curve on top of the actual PR curve.

Details

These functions can be plugged into the statistics argument of the gof methods in order to compare observed with simulated networks (see the gof-methods help page). There are three types of statistics:

(1) Univariate statistics, which aggregate a network into a single quantity. For example, modularity measures or density. The distribution of statistics can be displayed using histograms, density plots, and median bars. Univariate statistics take a sparse matrix (mat) as an argument and return a single numeric value that summarize a network matrix.

(2) Multivariate statistics, which aggregate a network into a vector of quantities. For example, the distribution of geodesic distances, edgewise shared partners, or indegree. These statistics typically have multiple values, e.g., esp(1), esp(2), esp(3) etc. The results can be displayed using multiple boxplots for simulated networks and a black curve for the observed network(s). Multivariate statistics take a sparse matrix (mat) as an argument and return a vector of numeric values that summarize a network matrix.

(3) Tie prediction statistics, which predict dyad states the observed network(s) by the dyad states in the simulated networks. For example, receiver operating characteristics (ROC) or precision-recall curves (PR) of simulated networks based on the model, or ROC or PR predictions of community co-membership matrices of the simulated vs. the observed network(s). Tie prediction statistics take a list of simulated sparse network matrices and another list of observed sparse network matrices (possibly containing only a single sparse matrix) as arguments and return a rocpr, roc, or pr object (as created by the respective functions rocpr, rocprgof, roc, and pr).

Users can create their own statistics for use with the codegof methods. To do so, one needs to write a function that accepts and returns the respective objects described in the enumeration above. It is advisable to look at the definitions of some of the existing functions to add custom functions. It is also possible to add an attribute called label to the return object, which describes what is being returned by the function. This label will be used as a descriptive label in the plot and for verbose output during computations. The examples section contains an example of a custom user statistic.

To aid the development of custom statistics, several helper functions are available: The roc, pr, and rocpr functions accept lists of simulated and observed sparse network matrices and compute ROC and precision recall curves as well as the area under the curve that can be used as network statistics. These functions are used internally for a number of functions related to community structure, where the community structure in the simulated networks is compared to the community structure in the observed network(s) by means of tie prediction. The rocprgof function provides the same functionality as the rocpr function, but it has an additional argument for controlling imputation of the first PR value. Another helper function is comemb, which accepts a vector of community memberships and converts it to a co-membership matrix. This function is also used internally by statistics like walktrap.roc and others.

Network statistics

The following built-in functions can be handed over to the `statistics` argument. See the usage section for their respective arguments.

(1) Univariate statistics:

`walktrap.modularity(mat)` Modularity distribution as computed by the Walktrap algorithm.

`fastgreedy.modularity(mat)` Modularity distribution as computed by the fast and greedy algorithm. Only sensible with undirected networks.

`maxmod.modularity(mat)` Optimal modularity distribution.

`edgebetweenness.modularity(mat)` Modularity distribution as computed by the Girvan-Newman edge betweenness community detection method.

`spinglass.modularity(mat)` Modularity distribution as computed by the Spinglass algorithm.

(2) Multivariate statistics:

`dsp` Dyad-wise shared partner distribution.

`esp(mat)` Edge-wise shared partner distribution.

`nsp(mat)` Non-edge-wise shared partner distribution.

`deg(mat)` Degree distribution (for undirected networks).

`ideg(mat)` Indegree distribution (for directed networks).

`odeg(mat)` Outdegree distribution (for directed networks).

`b1deg(mat)` Degree distribution (for the first mode in a two-mode network).

`b2deg(mat)` Degree distribution (for the second mode in a two-mode network).

`kstar(mat)` k-star distribution (for undirected networks).

`istar(mat)` in-star distribution (for directed networks).

`ostar(mat)` out-star distribution (for directed networks).

`b1star(mat)` k-star distribution (for the first mode in a two-mode network).

`b2star(mat)` k-star distribution (for the second mode in a two-mode network).

`kcycle(mat)` k-cycle distribution (for undirected networks).

`geodesic(mat)` Geodesic distance (or shortest path) distribution.

`triad.directed(mat)` Triad census (directed networks).

`triad.undirected(mat)` Triad census (undirected networks).

(3) Tie prediction statistics:

`walktrap.roc(sim, obs)` Receiver-operating characteristics of predicting the community structure in the observed network(s) by the community structure in the simulated networks, as computed by the Walktrap algorithm.

`walktrap.pr(sim, obs)` Precision-recall curve for predicting the community structure in the observed network(s) by the community structure in the simulated networks, as computed by the Walktrap algorithm.

`fastgreedy.roc(sim, obs)` Receiver-operating characteristics of predicting the community structure in the observed network(s) by the community structure in the simulated networks, as computed by the fast and greedy algorithm. Only sensible with undirected networks.

`fastgreedy.pr(sim, obs)` Precision-recall curve for predicting the community structure in the observed network(s) by the community structure in the simulated networks, as computed by the fast and greedy algorithm. Only sensible with undirected networks.

`maxmod.roc(sim, obs)` Receiver-operating characteristics of predicting the community structure in the observed network(s) by the community structure in the simulated networks, as computed by the modularity maximization algorithm.

`maxmod.pr(sim, obs)` Precision-recall curve for predicting the community structure in the observed network(s) by the community structure in the simulated networks, as computed by the modularity maximization algorithm.

`edgebetweenness.roc(sim, obs)` Receiver-operating characteristics of predicting the community structure in the observed network(s) by the community structure in the simulated networks, as computed by the Girvan-Newman edge betweenness community detection method.

`edgebetweenness.pr(sim, obs)` Precision-recall curve for predicting the community structure in the observed network(s) by the community structure in the simulated networks, as computed by the Girvan-Newman edge betweenness community detection method.

`spinglass.roc(sim, obs)` Receiver-operating characteristics of predicting the community structure in the observed network(s) by the community structure in the simulated networks, as computed by the Spinglass algorithm.

`spinglass.pr(sim, obs)` Precision-recall curve for predicting the community structure in the observed network(s) by the community structure in the simulated networks, as computed by the Spinglass algorithm.

`roc(sim, obs)` Receiver-operating characteristics. Prediction of the dyad states of the observed network(s) by the dyad states of the simulated networks.

`pr(sim, obs)` Precision-recall curve. Prediction of the dyad states of the observed network(s) by the dyad states of the simulated networks.

`rocpr(sim, obs)` Both receiver-operating characteristics and precision-recall curve. Prediction of the dyad states of the observed network(s) by the dyad states of the simulated networks.

Author(s)

Philip Leifeld (http://www.philipleifeld.com)

See Also

btergm-package gof gof-methods

Examples

```
# To see how these statistics are used, look at the examples section of
# ?"gof-methods". The following example illustrates how custom
# statistics can be created. Suppose one is interested in the density
# of a network. Then a univariate statistic can be created as follows.
```

```
dens <- function(mat) {             # univariate: one argument
  mat <- as.matrix(mat)             # sparse matrix -> normal matrix
  d <- sna::gden(mat)               # compute the actual statistic
  attributes(d)$label <- "Density"  # add a descriptive label
  return(d)                         # return the statistic
}

# Now the statistic can be used in the statistics argument of one of
# the gof methods.

# For illustrative purposes, let us consider an existing statistic, the
# indegree distribution, a multivariate statistic. It also accepts a
# single argument. Note that the sparse matrix is converted to a
# normal matrix object when it is used. First, statnet's summary
# method is used to compute the statistic. Names are attached to the
# resulting vector for the different indegree values. Then the vector
# is returned.

ideg <- function(mat) {
  d <- summary(mat ~ idegree(0:(nrow(mat) - 1)))
  names(d) <- 0:(length(d) - 1)
  attributes(d)$label <- "Indegree"
  return(d)
}

# See the gofstatistics.R file in the package for more complex examples.
```

`interpret` *Interpretation functions for ergm and btergm objects*

Description

Interpretation functions for ergm and btergm objects.

Usage

```
## S4 method for signature 'ergm'
interpret(object, formula = getformula(object),
    coefficients = coef(object), target = NULL, type = "tie", i, j)

## S4 method for signature 'btergm'
interpret(object, formula = getformula(object),
    coefficients = coef(object), target = NULL, type = "tie", i, j,
    t = 1:object@time.steps)

## S4 method for signature 'mtergm'
interpret(object, formula = getformula(object),
    coefficients = coef(object), target = NULL, type = "tie", i, j,
    t = 1:object@time.steps)
```

Arguments

object	An ergm, btergm, or mtergm object.
formula	The formula to be used for computing probabilities. By default, the formula embedded in the model object is retrieved and used.
coefficients	The estimates on which probabilities should be based. By default, the coefficients from the model object are retrieved and used. Custom coefficients can be handed over, for example, in order to compare versions of the model where the reciprocity term is fixed at 0 versus versions of the model where the reciprocity term is left as in the empirical result. This is one of the examples described in Desmarais and Cranmer (2012).
target	The response network on which probabilities are based. Depending on whether the function is applied to an ergm or btergm/mtergm object, this can be either a single network or a list of networks. By default, the (list of) network(s) provided as the left-hand side of the (T)ERGM formula is used.
type	If type = "tie" is used, probabilities at the edge level are computed. For example, what is the probability of a specific node i to be connected to a specific node j given the rest of the network and given the model? If type = "dyad" is used, probabilities at the dyad level are computed. For example, what is the probability that node i is connected to node j but not vice-versa, or what is the probability that nodes i and j and mutually connected in a directed network? If type = "node" is used, probabilities at the node level are computed. For example, what is the probability that node i is connected to a set of three other j nodes given the rest of the network and the model?
i	A single (sender) node i or a set of (sender) nodes i. If type = "node" is used, this can be more than one node and should be provided as a vector. The i argument can be either provided as the index of the node in the sociomatrix (e.g., the fourth node would be i = 4) or the row name of the node in the sociomatrix (e.g., i = "Peter"). If more than one node is provided and type = "node", there can be only one (receiver) node j. The i and j arguments are used to specify for which nodes probabilities should be computed. For example, what is the probability that i = 4 is connected to i = 7?
j	A single (receiver) node j or a set of (receiver) nodes j. If type = "node" is used, this can be more than one node and should be provided as a vector. The j argument can be either provided as the index of the node in the sociomatrix (e.g., the fourth node would be j = 4) or the row name of the node in the sociomatrix (e.g., j = "Mary"). If more than one node is provided and type = "node", there can be only one (sender) node i. The i and j arguments are used to specify for which nodes probabilities should be computed. For example, what is the probability that i = 4 is connected to i = 7?
t	A vector of (numerical) time steps for which the probabilities should be computed. This only applies to btergm objects because ergm objects are by definition based on a single time step. By default, all available time steps are used. It is, for example, possible to compute probabilities only for a single time step by specifying, e.g., t = 5 in order to compute probabilities for the fifth response network.

Details

The `interpret` function facilitates interpretation of ERGMs and TERGMs at the micro level via block Gibbs sampling, as described in Desmarais and Cranmer (2012). There are generic methods for `ergm` objects, `btergm` objects, and `mtergm` objects. The function can be used to interpret these models at the tie or edge level, dyad level, and block level.

For example, what is the probability that two specific nodes i (the sender) and node j (the receiver) are connected given the rest of the network and given the model? Or what is the probability that any two nodes are tied at t = 2 if they were tied (or disconnected) at t = 1 (i.e., what is the amount of tie stability)? These tie- or edge-level questions can be answered if the `type = "tie"` argument is used.

Another example: What is the probability that node i has a tie to node j but not vice-versa? Or that i and j maintain a reciprocal tie? Or that they are disconnected? How much more or less likely are i and j reciprocally connected if the `mutual` term in the model is fixed at 0 (compared to the model that includes the estimated parameter for reciprocity)? See example below. These dyad-level questions can be answered if the `type = "dyad"` argument is used.

Or what is the probability that a specific node i is connected to nodes j1 and j2 but not to j5 and j7? And how likely is any node i to be connected to exactly four j nodes? These node-level questions (focusing on the ties of node i or node j) can be answered by using the `type = "node"` argument.

The typical procedure is to manually enumerate all dyads or sender-receiver-time combinations with certain properties and repeat the same thing with some alternative properties for contrasting the two groups. Then apply the `interpret` function to the two groups of dyads and compute a measure of central tendency (e.g., mean or median) and possibly some uncertainy measure (i.e., confidence intervals) from the distribution of dyadic probabilities in each group. For example, if there is a gender attribute, one can sample male-male or female-female dyads, compute the distributions of edge probabilities for the two sets of dyads, and create boxplots or barplots with confidence intervals for the two types of dyads in order to contrast edge probabilities for male versus female same-sex dyads.

See also the edgeprob function for automatic computation of all dyadic edge probabilities.

References

Desmarais, Bruce A. and Skyler J. Cranmer (2012): Micro-Level Interpretation of Exponential Random Graph Models with Application to Estuary Networks. *The Policy Studies Journal* 40(3): 402–434.

See Also

edgeprob btergm-package btergm timesteps.btergm

Examples

```
##### The following example is a TERGM adaptation of the #####
##### dyad-level example provided in figure 5(c) on page #####
##### 424 of Desmarais and Cranmer (2012) in the PSJ. At #####
##### each time step, it compares dyadic probabilities   #####
##### (no tie, unidirectional tie, and reciprocal tie    #####
##### probability) between a fitted model and a model    #####
##### where the reciprocity effect is fixed at 0 based   #####
```

```
##### on 20 randomly selected dyads per time step. The   #####
##### results are visualized using a grouped bar plot.   #####

## Not run:
# create toy dataset and fit a model
networks <- list()
for (i in 1:3) {           # create 3 random networks with 10 actors
  mat <- matrix(rbinom(100, 1, 0.25), nrow = 10, ncol = 10)
  diag(mat) <- 0           # loops are excluded
  nw <- network(mat)       # create network object
  networks[[i]] <- nw      # add network to the list
}
fit <- btergm(networks ~ edges + istar(2) + mutual, R = 200)

# extract coefficients and create null hypothesis vector
null <- coef(fit)  # estimated coefs
null[3] <- 0       # set mutual term = 0

# sample 20 dyads per time step and compute probability ratios
probabilities <- matrix(nrow = 9, ncol = length(networks))
# nrow = 9 because three probabilities + upper and lower CIs
colnames(probabilities) <- paste("t =", 1:length(networks))
for (t in 1:length(networks)) {
  d <- dim(as.matrix(networks[[t]]))  # how many row and column nodes?
  size <- d[1] * d[2]                 # size of the matrix
  nw <- matrix(1:size, nrow = d[1], ncol = d[2])
  nw <- nw[lower.tri(nw)]             # sample only from lower triangle b/c
  samp <- sample(nw, 20)              # dyadic probabilities are symmetric
  prob.est.00 <- numeric(0)
  prob.est.01 <- numeric(0)
  prob.est.11 <- numeric(0)
  prob.null.00 <- numeric(0)
  prob.null.01 <- numeric(0)
  prob.null.11 <- numeric(0)
  for (k in 1:20) {
    i <- arrayInd(samp[k], d)[1, 1]   # recover 'i's and 'j's from sample
    j <- arrayInd(samp[k], d)[1, 2]
    # run interpretation function with estimated coefs and mutual = 0:
    int.est <- interpret(fit, type = "dyad", i = i, j = j, t = t)
    int.null <- interpret(fit, coefficients = null, type = "dyad",
        i = i, j = j, t = t)
    prob.est.00 <- c(prob.est.00, int.est[[1]][1, 1])
    prob.est.11 <- c(prob.est.11, int.est[[1]][2, 2])
    mean.est.01 <- (int.est[[1]][1, 2] + int.est[[1]][2, 1]) / 2
    prob.est.01 <- c(prob.est.01, mean.est.01)
    prob.null.00 <- c(prob.null.00, int.null[[1]][1, 1])
    prob.null.11 <- c(prob.null.11, int.null[[1]][2, 2])
    mean.null.01 <- (int.null[[1]][1, 2] + int.null[[1]][2, 1]) / 2
    prob.null.01 <- c(prob.null.01, mean.null.01)
  }
  prob.ratio.00 <- prob.est.00 / prob.null.00  # ratio of est. and null hyp
  prob.ratio.01 <- prob.est.01 / prob.null.01
  prob.ratio.11 <- prob.est.11 / prob.null.11
```

```
  probabilities[1, t] <- mean(prob.ratio.00)   # mean estimated 00 tie prob
  probabilities[2, t] <- mean(prob.ratio.01)   # mean estimated 01 tie prob
  probabilities[3, t] <- mean(prob.ratio.11)   # mean estimated 11 tie prob
  ci.00 <- t.test(prob.ratio.00, conf.level = 0.99)$conf.int
  ci.01 <- t.test(prob.ratio.01, conf.level = 0.99)$conf.int
  ci.11 <- t.test(prob.ratio.11, conf.level = 0.99)$conf.int
  probabilities[4, t] <- ci.00[1]              # lower 00 conf. interval
  probabilities[5, t] <- ci.01[1]              # lower 01 conf. interval
  probabilities[6, t] <- ci.11[1]              # lower 11 conf. interval
  probabilities[7, t] <- ci.00[2]              # upper 00 conf. interval
  probabilities[8, t] <- ci.01[2]              # upper 01 conf. interval
  probabilities[9, t] <- ci.11[2]              # upper 11 conf. interval
}

# create barplots from probability ratios and CIs
require("gplots")
bp <- barplot2(probabilities[1:3, ], beside = TRUE, plot.ci = TRUE,
    ci.l = probabilities[4:6, ], ci.u = probabilities[7:9, ],
    col = c("tan", "tan2", "tan3"), ci.col = "grey40",
    xlab = "Dyadic tie values", ylab = "Estimated Prob./Null Prob.")
mtext(1, at = bp, text = c("(0,0)", "(0,1)", "(1,1)"), line = 0, cex = 0.5)


##### The following examples illustrate the behavior of  #####
##### the interpret function with undirected and/or      #####
##### bipartite graphs with or without structural zeros. #####

library("statnet")
library("btergm")

# micro-level interpretation for undirected network with structural zeros
set.seed(12345)
mat <- matrix(rbinom(400, 1, 0.1), nrow = 20, ncol = 20)
mat[1, 5] <- 1
mat[10, 7] <- 1
mat[15, 3] <- 1
mat[18, 4] < 1
nw <- network(mat, directed = FALSE, bipartite = FALSE)
cv <- matrix(rnorm(400), nrow = 20, ncol = 20)
offsetmat <- matrix(rbinom(400, 1, 0.1), nrow = 20, ncol = 20)
offsetmat[1, 5] <- 1
offsetmat[10, 7] <- 1
offsetmat[15, 3] <- 1
offsetmat[18, 4] < 1
model <- ergm(nw ~ edges + kstar(2) + edgecov(cv) + offset(edgecov(offsetmat)),
    offset.coef = -Inf)
summary(model)

# tie-level interpretation (note that dyad interpretation would not make any
# sense in an undirected network):
interpret(model, type = "tie", i = 1, j = 2)  # 0.28 (= normal dyad)
interpret(model, type = "tie", i = 1, j = 5)  # 0.00 (= structural zero)
```

```
# node-level interpretation; note the many 0 probabilities due to the
# structural zeros; also note the warning message that the probabilities may
# be slightly imprecise because -Inf needs to be approximated by some large
# negative number (-9e8):
interpret(model, type = "node", i = 1, j = 3:5)

# repeat the same exercise for a directed network
nw <- network(mat, directed = TRUE, bipartite = FALSE)
model <- ergm(nw ~ edges + istar(2) + edgecov(cv) + offset(edgecov(offsetmat)),
    offset.coef = -Inf)
interpret(model, type = "tie", i = 1, j = 2)  # 0.13 (= normal dyad)
interpret(model, type = "tie", i = 1, j = 5)  # 0.00 (= structural zero)
interpret(model, type = "dyad", i = 1, j = 2)  # results for normal dyad
interpret(model, type = "dyad", i = 1, j = 5)  # results for i->j struct. zero
interpret(model, type = "node", i = 1, j = 3:5)

# micro-level interpretation for bipartite graph with structural zeros
set.seed(12345)
mat <- matrix(rbinom(200, 1, 0.1), nrow = 20, ncol = 10)
mat[1, 5] <- 1
mat[10, 7] <- 1
mat[15, 3] <- 1
mat[18, 4] < 1
nw <- network(mat, directed = FALSE, bipartite = TRUE)
cv <- matrix(rnorm(200), nrow = 20, ncol = 10)  # some covariate
offsetmat <- matrix(rbinom(200, 1, 0.1), nrow = 20, ncol = 10)
offsetmat[1, 5] <- 1
offsetmat[10, 7] <- 1
offsetmat[15, 3] <- 1
offsetmat[18, 4] < 1
model <- ergm(nw ~ edges + b1star(2) + edgecov(cv)
    + offset(edgecov(offsetmat)), offset.coef = -Inf)
summary(model)

# tie-level interpretation; note the index for the second mode starts with 21
interpret(model, type = "tie", i = 1, j = 21)

# dyad-level interpretation does not make sense because network is undirected;
# node-level interpretation prints warning due to structural zeros, but
# computes the correct probabilities (though slightly imprecise because -Inf
# is approximated by some small number:
interpret(model, type = "node", i = 1, j = 21:25)

# compute all dyadic probabilities
dyads <- edgeprob(model)
dyads

## End(Not run)
```

`simulate.btergm` *Simulate new networks from btergm objects*

Description

Simulate new networks from btergm objects.

Usage

```
## S3 method for class 'btergm'
simulate(object, nsim = 1, seed = NULL,
    index = NULL, formula = getformula(object),
    coef = object@coef, verbose = TRUE, ...)
```

Arguments

object	A btergm object, resulting from a call of the btergm function.
nsim	The number of networks to be simulated. Note that for values greater than one, a network.list object is returned, which can be indexed just like a list object, for example mynetworks[[1]] for the first simulated network in the object mynetworks.
seed	Random number integer seed. See set.seed.
formula	A model formula from which the new network(s) should be simulated. By default, the formula is taken from the btergm object.
index	Index of the network from which the new network(s) should be simulated. The index refers to the list of response networks on the left-hand side of the model formula. Note that more recent networks are located at the end of the list. By default, the first (= oldest) network is used.
coef	A vector of parameter estimates. By default, the coefficients are extracted from the given btergm object.
verbose	Print additional details while running the simulations?
...	Arbitrary further arguments are handed over to the simulate.formula function. For details, refer to the help page of the simulate.formula function.

Details

The simulate.btergm function is a wrapper for the simulate.formula function in the **ergm** package (see help("simulate.formula")). It can be used to simulate new networks from a btergm object. The index argument specifies from which of the original networks the new network(s) should be simulated. For example, if object is an estimation based on cosponsorship networks from the 99th to the 107th Congress (as in Desmarais and Cranmer 2012), and the cosponsorship network in the 108th Congress should be predicted using the simulate.btergm function, then the argument index = 9 should be passed to the function because the network should be based on the 9th network in the list (that is, the latest network, which is the cosponsorship network for the 107th Congress). Note that all relevant objects (the networks and the covariates) must be present in the workspace (as was the case during the estimation of the model).

References

Desmarais, Bruce A. and Skyler J. Cranmer (2012): Statistical Mechanics of Networks: Estimation and Uncertainty. *Physica A* 391: 1865–1876.

See Also

btergm-package btergm gof

`tergm-terms` *Temporal dependencies for TERGMs*

Description

Network statistics that span multiple time points.

Details

In addition to the ERGM user terms that can be estimated within a single network (see ergm-terms), the **btergm** package provides additional model terms that can be used within a formula. These additional statistics span multiple time periods and are therefore called "temporal dependencies." Examples include memory terms (i.e., positive autoregression, dyadic stability, edge innovation, or edge loss), delayed reciprocity or mutuality, and time covariates (i.e., functions of time or interactions with time):

`memory(type = "stability", lag = 1)` Memory terms control for the impact of a previous network on the current network. Four different types of memory terms are available: positive autoregression (`type = "autoregression"`) checks whether previous ties are carried over to the current network; dyadic stability (`type = "stability"`) checks whether both edges and non-edges are stable between the previous and the current network; edge loss (`type = "loss"`) checks whether ties in the previous network have been dissolved and no longer exist in the current network; and edge innovation (`type = "innovation"`) checks whether previously unconnected nodes have the tendency to become tied in the current network. The `lag` argument accepts integer values and controls whether the comparison is made with the previous network (`lag = 1`), the pre-previous network (`lag = 2`) etc. Note that as `lag` increases, the number of time steps on the dependent variable decreases.

`delrecip(mutuality = FALSE, lag = 1)` The `delrecip` term checks for delayed reciprocity. For example, if node j is tied to node i at t = 1, does this lead to a reciprocation of that tie back from i to j at t = 2? If `mutuality = TRUE` is set, this extends not only to ties, but also non-ties. That is, if i is not tied to j at t = 1, will this lead to j not being tied to i at t = 2, in addition to positively reciprocal patterns over time? The `lag` argument controls the size of the temporal lag: with `lag = 1`, reciprocity over one consecutive time period is checked. Note that as `lag` increases, the number of time steps on the dependent variable decreases.

`timecov(x = NULL, minimum = 1, maximum = NULL, transform = function(t) 1 + (0 * t) + (0 * t^2))` The `timecov` model term checks for linear or non-linear time trends with regard to edge formation. Optionally, this can be combined with a covariate to create an interaction effect between a dyadic covariate and time in order to test whether the importance of a covariate increases or decreases over time. In the default case, time is modeled as being constant over time. By tweaking the `transform` function, arbitrary functional forms of time can be tested. For example, `transform = sqrt` (for a geometrically decreasing time effect), `transform = function(x) x^2` (for a geometrically increasing time effect), `transform = function(t) t` (for a linear time trend) or polynomial functional forms can be used. For time steps below the

minimum value and above the maximum value, the time covariate is set to 0. These arguments can be used to create stepping effects, for example to use a value of 0 up to an external event and 1 from that event onwards in order to control for external influences. The model term works in a similar way as the separate timecov function.

See Also

btergm-package btergm preprocess timecov ergm-terms

`timecov` *Create interaction terms between covariates and (transformations of) time for TERGMs*

Description

Create interaction terms between covariates and (transformations of) time for TERGMs.

Usage

```
timecov(covariate, minimum = 1, maximum = length(covariate),
    transform = function(t) 1 + (0 * t) + (0 * t^2),
    onlytime = FALSE)
```

Arguments

`covariate`	A list of matrices or network objects (a time-varying dyadic covariate). Note that nodal covariates can be manually converted into dyadic covariates to use them with this function.
`minimum`	The first matrix or network in the covariate list which should be non-zero. All matrices or networks before this time step are filled with zeros.
`maximum`	The last matrix or network in the covariate list which should be non-zero. All matrices or networks after this time step are filled with zeros.
`transform`	A function that transforms the time axis. By default, all time steps are weighted equally. For example, if there are five consecutive matrices, all of them are equally important, i.e., time is represented by weights of `1-1-1-1-1`. A linearly increasing importance of time can be achieved by using a multiplication with time, for example `function(t) = 2 * t` leads to a multiplication of all values in the covariates by the values `2-4-6-8-10`, respectively by time step. Quadratic or higher-order polynomials are possible using more complex transformations of time.
`onlytime`	If `onlytime = TRUE` is set, the function returns a list of matrices where all entries within a matrix are identical and reflect the transformation of time. This is useful when a function of time per se should be included in the model as a covariate, e.g., when the tie formation probability increases or decreases over time. If `onlytime = FALSE` is set, the aforementioned list of matrices is interacted with the covariate list, i.e., each value in the matrix at time step `t` is multiplied by the transformation of time at that time step.

Details

IMPORTANT: This helper function is usually not needed because the `timecov` model term can be used directly in the TERGM formula. See tergm-terms for further information on temporal dependency terms for TERGMs.

The `timecov` function takes a list of matrices or networks (a varying dyadic covariate) and creates an interaction term with time. For example, if the rules of network formation are expected to change after some time steps (say, after the third out of six time steps), one can model the effect of the covariate for the first three time points and for the remaining three time points using separate model terms. To achieve this, the covariate matrix at each time step is multiplied by zeros or ones, depending on whether the time step should be incorporated in that model term. In this situation, one would create two dyadic covariates, one where the first three time periods are present and the remaining three are set to zero, and the other one where the first three time periods are set to zero and the remaining ones are present.

Another usage scenario is that time per se may have a polynomial effect (of any shape) on the probability of forming a tie. In this case, the `timecov` function can be used to create a covariate list of matrices where the entries of the matrix correspond to polynomial functions of time. For example, if tie formation becomes increasingly likely (as expressed by a linear relationship), a list of matrices with linearly increasing entries over time can be created. Such time effects can also be interacted with other covariates. For example, the further time progresses, the more (or less) the property captured by the covariate becomes important for tie formation.

See Also

btergm-package btergm preprocess

Index