Package: missSBM (via r-universe)

October 29, 2026

Type Package

Title Handling Missing Data in Stochastic Block Models

Version 1.1.0

Maintainer Julien Chiquet <julien.chiquet@inrae.fr>

Description When a network is partially observed (here, NAs in the adjacency matrix rather than 1 or 0 due to missing information between node pairs), it is possible to account for the underlying process that generates those NAs. 'missSBM', presented in 'Barbillon, Chiquet and Tabouy' (2022) <doi:10.18637/jss.v101.i12>, adjusts the popular stochastic block model from network data sampled under various missing data conditions, as described in 'Tabouy, Barbillon and Chiquet' (2019) <doi:10.1080/01621459.2018.1562934>.

URL https://grosssbm.github.io/missSBM/

BugReports https://github.com/grossSBM/missSBM/issues

License GPL-3

Encoding UTF-8

LazyData true

Depends R (>= 3.4.0)

Imports Rcpp, methods, igraph, ggplot2, future.apply, R6, rlang, sbm, magrittr, Matrix, RSpectra

LinkingTo Rcpp, RcppArmadillo

Collate 'utils_missSBM.R' 'R6Class-networkSampling.R' 'R6Class-networkSampling_fit.R' 'R6Class-simpleSBM_fit.R' 'R6Class-missSBM_fit.R' 'R6Class-missSBM_collection.R' 'R6Class-networkSampler.R' 'R6Class-partlyObservedNetwork.R' 'RcppExports.R' 'er_network.R' 'estimateMissSBM.R' 'frenchblog2007.R' 'kmeans.R' 'missSBM-package.R' 'observeNetwork.R' 'war.R'

Suggests aricode, blockmodels, corrplot, future, testthat (>= 2.1.0), covr, knitr, rmarkdown, spelling

VignetteBuilder knitr

Language en-US

Config/roxygen2/version 8.0.0

NeedsCompilation yes

Author Julien Chiquet [aut, cre] (ORCID: <https://orcid.org/0000-0002-3629-3429>), Pierre Barbillon [aut] (ORCID: <https://orcid.org/0000-0002-7766-7693>), Timothée Tabouy [aut], Jean-Benoist Léger [ctb] (provided C++ implementaion of K-means), François Gindraud [ctb] (provided C++ interface to NLopt), großBM team [ctb]

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

Date/Publication 2026-07-21 22:40:02 UTC

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

RemoteRef HEAD

RemoteSha a5400464afaa533706a5eba7cc034176a51d4033

Contents

blockDyadSampler . . . 3
blockDyadSampling_fit . . . 4
blockNodeSampler . . . 5
blockNodeSampling_fit . . . 6
coef.missSBM_fit . . . 7
covarDyadSampling_fit . . . 7
covarNodeSampling_fit . . . 8
degreeSampler . . . 9
degreeSampling_fit . . . 10
doubleStandardSampler . . . 11
doubleStandardSampling_fit . . . 12
dyadSampler . . . 13
dyadSampling_fit . . . 14
er_network . . . 15
estimateMissSBM . . . 15
fitted.missSBM_fit . . . 17
frenchblog2007 . . . 18
l1_similarity . . . 19
missSBM_collection . . . 19
missSBM_fit . . . 22
missSBM_param . . . 26
networkSampler . . . 28
networkSampling . . . 29
networkSamplingDyads_fit . . . 30
networkSamplingNodes_fit . . . 32
nodeSampler . . . 33
nodeSampling_fit . . . 34

observeNetwork 35
partlyObservedNetwork 37
plot.missSBM_fit 38
predicted.missSBM_fit 39
simpleDyadSampler 40
simpleNodeSampler 41
SimpleSBM_fit 42
SimpleSBM_fit_MNAR 43
SimpleSBM_fit_noCov 45
SimpleSBM_fit_withCov 46
snowballSampler 48
summary.missSBM_fit 48
war 49

Index **50**

`blockDyadSampler` *Class for defining a block dyad sampler*

Description

Class for defining a block dyad sampler

Super classes

`networkSampling` -> `networkSampler` -> `dyadSampler` -> `blockDyadSampler`

Active bindings

`df` the number of parameters of this sampling

Methods

Public methods:

- `blockDyadSampler$new()`
- `blockDyadSampler$clone()`

`blockDyadSampler$new()`: constructor for networkSampling

Usage:

```
blockDyadSampler$new(
  parameters = NA,
  nbNodes = NA,
  directed = FALSE,
  clusters = NA
)
```

Arguments:

`parameters` the vector of parameters associated to the sampling at play

`nbNodes` number of nodes in the network

`directed` logical, directed network of not

`clusters` a vector of class memberships

`blockDyadSampler$clone()`: The objects of this class are cloneable with this method.

Usage:

```
blockDyadSampler$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`blockDyadSampling_fit` *Class for fitting a block-dyad sampling*

Description

Class for fitting a block-dyad sampling

Super classes

`networkSampling` -> `networkSamplingDyads_fit` -> `blockDyadSampling_fit`

Active bindings

`vExpec` variational expectation of the sampling

`log_lambda` matrix, term for adjusting the imputation step which depends on the type of sampling

Methods

Public methods:

- `blockDyadSampling_fit$new()`
- `blockDyadSampling_fit$update_parameters()`
- `blockDyadSampling_fit$clone()`

`blockDyadSampling_fit$new()`: constructor

Usage:

```
blockDyadSampling_fit$new(partlyObservedNetwork, blockInit)
```

Arguments:

`partlyObservedNetwork` a object with class partlyObservedNetwork representing the observed data with possibly missing entries

`blockInit` n x Q matrix of initial block indicators

`blockDyadSampling_fit$update_parameters()`: a method to update the estimation of the parameters. By default, nothing to do (corresponds to MAR sampling)

Usage:

```
blockDyadSampling_fit$update_parameters(nu, Z)
```

Arguments:

`nu` the matrix of (uncorrected) imputation for missing entries

`Z` probabilities of block memberships

`blockDyadSampling_fit$clone()`: The objects of this class are cloneable with this method.

Usage:

```
blockDyadSampling_fit$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`blockNodeSampler` *Class for defining a block node sampler*

Description

Class for defining a block node sampler

Super classes

`networkSampling` -> `networkSampler` -> `nodeSampler` -> `blockNodeSampler`

Methods

Public methods:

- `blockNodeSampler$new()`
- `blockNodeSampler$clone()`

`blockNodeSampler$new()`: constructor for networkSampling

Usage:

```
blockNodeSampler$new(
  parameters = NA,
  nbNodes = NA,
  directed = FALSE,
  clusters = NA
)
```

Arguments:

`parameters` the vector of parameters associated to the sampling at play

`nbNodes` number of nodes in the network

`directed` logical, directed network of not

`clusters` a vector of class memberships

`blockNodeSampler$clone()`: The objects of this class are cloneable with this method.

Usage:

```
blockNodeSampler$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`blockNodeSampling_fit` *Class for fitting a block-node sampling*

Description

Class for fitting a block-node sampling

Super classes

`networkSampling` -> `networkSamplingNodes_fit` -> `blockNodeSampling_fit`

Active bindings

`vExpec` variational expectation of the sampling

`log_lambda` double, term for adjusting the imputation step which depends on the type of sampling

Methods

Public methods:

- `blockNodeSampling_fit$new()`
- `blockNodeSampling_fit$update_parameters()`
- `blockNodeSampling_fit$clone()`

`blockNodeSampling_fit$new()`: constructor

Usage:

```
blockNodeSampling_fit$new(partlyObservedNetwork, blockInit)
```

Arguments:

`partlyObservedNetwork` a object with class partlyObservedNetwork representing the observed data with possibly missing entries

`blockInit` n x Q matrix of initial block indicators

`blockNodeSampling_fit$update_parameters()`: a method to update the estimation of the parameters. By default, nothing to do (corresponds to MAR sampling)

Usage:

```
blockNodeSampling_fit$update_parameters(imputedNet, Z)
```

Arguments:

`imputedNet` an adjacency matrix where missing values have been imputed

`Z` indicator of blocks

`blockNodeSampling_fit$clone()`: The objects of this class are cloneable with this method.

Usage:

```
blockNodeSampling_fit$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`coef.missSBM_fit` *Extract model coefficients*

Description

Extracts model coefficients from objects `missSBM_fit` returned by `estimateMissSBM()`

Usage

```
## S3 method for class 'missSBM_fit'
coef(
  object,
  type = c("mixture", "connectivity", "covariates", "sampling"),
  ...
)
```

Arguments

`object`	an R6 object with class `missSBM_fit`
`type`	type of parameter that should be extracted. Either "mixture" (default), "connectivity", "covariates" or "sampling"
`...`	additional parameters for S3 compatibility. Not used

Value

A vector or matrix of coefficients extracted from the missSBM_fit model.

`covarDyadSampling_fit` *Class for fitting a dyad sampling with covariates*

Description

Class for fitting a dyad sampling with covariates

Super classes

`networkSampling` -> `networkSamplingDyads_fit` -> `covarDyadSampling_fit`

Active bindings

vExpec variational expectation of the sampling

Methods

Public methods:

- covarDyadSampling_fit$new()
- covarDyadSampling_fit$clone()

covarDyadSampling_fit$new(): constructor

Usage:

```
covarDyadSampling_fit$new(partialNet, ...)
```

Arguments:

partialNet a object with class partlyObservedNetwork representing the observed data with possibly missing entries

... used for compatibility

covarDyadSampling_fit$clone(): The objects of this class are cloneable with this method.

Usage:

```
covarDyadSampling_fit$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

---

covarNodeSampling_fit *Class for fitting a node-centered sampling with covariate*

Description

Class for fitting a node-centered sampling with covariate

Super classes

networkSampling -> networkSamplingNodes_fit -> covarNodeSampling_fit

Active bindings

vExpec variational expectation of the sampling

Methods

Public methods:

- covarNodeSampling_fit$new()
- covarNodeSampling_fit$clone()

covarNodeSampling_fit$new(): constructor

Usage:

```
covarNodeSampling_fit$new(partlyObservedNetwork, ...)
```

Arguments:

partlyObservedNetwork a object with class partlyObservedNetwork representing the observed data with possibly missing entries

... used for compatibility

covarNodeSampling_fit$clone(): The objects of this class are cloneable with this method.

Usage:

```
covarNodeSampling_fit$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

---

degreeSampler *Class for defining a degree sampler*

Description

Class for defining a degree sampler

Super classes

networkSampling -> networkSampler -> nodeSampler -> degreeSampler

Methods

Public methods:

- degreeSampler$new()
- degreeSampler$clone()

degreeSampler$new(): constructor for networkSampling

Usage:

```
degreeSampler$new(parameters = NA, degrees = NA, directed = FALSE)
```

Arguments:

parameters the vector of parameters associated to the sampling at play

degrees vector of nodes' degrees

`directed` logical, directed network of not

`degreeSampler$clone()`: The objects of this class are cloneable with this method.

Usage:

```
degreeSampler$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`degreeSampling_fit` *Class for fitting a degree sampling*

Description

Class for fitting a degree sampling

Super classes

`networkSampling` -> `networkSamplingNodes_fit` -> `degreeSampling_fit`

Active bindings

`vExpec` variational expectation of the sampling

Methods

Public methods:

- `degreeSampling_fit$new()`
- `degreeSampling_fit$update_parameters()`
- `degreeSampling_fit$update_imputation()`
- `degreeSampling_fit$clone()`

`degreeSampling_fit$new()`: constructor

Usage:

```
degreeSampling_fit$new(partlyObservedNetwork, blockInit, connectInit)
```

Arguments:

`partlyObservedNetwork` a object with class partlyObservedNetwork representing the observed data with possibly missing entries

`blockInit` n x Q matrix of initial block indicators

`connectInit` Q x Q matrix of initial block probabilities of connection

`degreeSampling_fit$update_parameters()`: a method to update the estimation of the parameters. By default, nothing to do (corresponds to MAR sampling)

Usage:

```
degreeSampling_fit$update_parameters(imputedNet, ...)
```

Arguments:

`imputedNet` an adjacency matrix where missing values have been imputed

`...` used for compatibility

`degreeSampling_fit$update_imputation()`: a method to update the imputation of the missing entries.

Usage:

```
degreeSampling_fit$update_imputation(PI, ...)
```

Arguments:

`PI` the matrix of inter/intra class probability of connection

`...` use for compatibility

`degreeSampling_fit$clone()`: The objects of this class are cloneable with this method.

Usage:

```
degreeSampling_fit$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`doubleStandardSampler` *Class for defining a double-standard sampler*

Description

Class for defining a double-standard sampler

Super classes

`networkSampling` -> `networkSampler` -> `dyadSampler` -> `doubleStandardSampler`

Methods

Public methods:

- `doubleStandardSampler$new()`
- `doubleStandardSampler$clone()`

`doubleStandardSampler$new()`: constructor for networkSampling

Usage:

```
doubleStandardSampler$new(parameters = NA, adjMatrix = NA, directed = FALSE)
```

Arguments:

`parameters` the vector of parameters associated to the sampling at play

`adjMatrix` matrix of adjacency

`directed` logical, directed network of not

`doubleStandardSampler$clone()`: The objects of this class are cloneable with this method.

Usage:

```
doubleStandardSampler$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`doubleStandardSampling_fit` *Class for fitting a double-standard sampling*

Description

Class for fitting a double-standard sampling

Super classes

`networkSampling` -> `networkSamplingDyads_fit` -> `doubleStandardSampling_fit`

Active bindings

`vExpec` variational expectation of the sampling

Methods

Public methods:

- `doubleStandardSampling_fit$new()`
- `doubleStandardSampling_fit$update_parameters()`
- `doubleStandardSampling_fit$update_imputation()`
- `doubleStandardSampling_fit$clone()`

`doubleStandardSampling_fit$new()`: constructor

Usage:

```
doubleStandardSampling_fit$new(partlyObservedNetwork, ...)
```

Arguments:

`partlyObservedNetwork` a object with class partlyObservedNetwork representing the observed data with possibly missing entries

`...` used for compatibility

`doubleStandardSampling_fit$update_parameters()`: a method to update the estimation of the parameters. By default, nothing to do (corresponds to MAR sampling)

Usage:

```
doubleStandardSampling_fit$update_parameters(nu, ...)
```

Arguments:

nu an adjacency matrix with imputed values (only)

... use for compatibility

`doubleStandardSampling_fit$update_imputation()`: a method to update the imputation of the missing entries.

Usage:

```
doubleStandardSampling_fit$update_imputation(nu)
```

Arguments:

nu the matrix of (uncorrected) imputation for missing entries

`doubleStandardSampling_fit$clone()`: The objects of this class are cloneable with this method.

Usage:

```
doubleStandardSampling_fit$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

`dyadSampler` *Virtual class for all dyad-centered samplers*

Description

Virtual class for all dyad-centered samplers

Super classes

`networkSampling` -> `networkSampler` -> `dyadSampler`

Methods

Public methods:

- `dyadSampler$new()`
- `dyadSampler$clone()`

`dyadSampler$new()`: constructor for networkSampling

Usage:

```
dyadSampler$new(type = NA, parameters = NA, nbNodes = NA, directed = FALSE)
```

Arguments:

type character for the type of sampling. must be in ("dyad", "covar-dyad", "node", "covar-node", "block-node", "block-dyad", "double-standard", "degree")

parameters the vector of parameters associated to the sampling at play

nbNodes number of nodes in the network

directed logical, directed network of not

`dyadSampler$clone()`: The objects of this class are cloneable with this method.

Usage:

```
dyadSampler$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`dyadSampling_fit` *Class for fitting a dyad sampling*

Description

Class for fitting a dyad sampling

Super classes

`networkSampling` -> `networkSamplingDyads_fit` -> `dyadSampling_fit`

Active bindings

`vExpec` variational expectation of the sampling

Methods

Public methods:

- `dyadSampling_fit$new()`
- `dyadSampling_fit$clone()`

`dyadSampling_fit$new()`: constructor

Usage:

```
dyadSampling_fit$new(partlyObservedNetwork, ...)
```

Arguments:

`partlyObservedNetwork` a object with class partlyObservedNetwork representing the observed data with possibly missing entries

`...` used for compatibility

`dyadSampling_fit$clone()`: The objects of this class are cloneable with this method.

Usage:

```
dyadSampling_fit$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

er_network *ER ego centered network*

Description

A dataset containing the weighted PPI network centered around the ESR1 (ER) protein

Usage

```
er_network
```

Format

A sparse symmetric matrix with 741 rows and 741 columns ESR1

Source

https://string-db.org/

Examples

```
data("er_network")
class(er_network)
```

estimateMissSBM *Estimation of simple SBMs with missing data*

Description

Variational EM inference of Stochastic Block Models indexed by block number from a partially observed network.

Usage

```
estimateMissSBM(
  adjacencyMatrix,
  vBlocks,
  sampling,
  covariates = list(),
  control = missSBM_param()
)
```

Arguments

`adjacencyMatrix`	The N x N adjacency matrix of the network data. If `adjacencyMatrix` is symmetric, we assume an undirected network with no loop; otherwise the network is assumed to be directed.
`vBlocks`	The vector of number of blocks considered in the collection.
`sampling`	The model used to described the process that originates the missing data: MAR designs ("dyad", "node","covar-dyad","covar-node","snowball") and MNAR designs ("double-standard", "block-dyad", "block-node" , "degree") are available. See details.
`covariates`	An optional list with M entries (the M covariates). If the covariates are node-centered, each entry of `covariates` must be a size-N vector; if the covariates are dyad-centered, each entry of `covariates` must be N x N matrix.
`control`	a list-like structure for controlling advanced features, with default generated by `missSBM_param`. See the associated documentation for details.

Details

Internal functions use `future_lapply`, so set your plan to `'multisession'` or `'multicore'` to use several cores/workers.

The different sampling designs are split into two families in which we find dyad-centered and node-centered samplings. See doi:10.1080/01621459.2018.1562934 for a complete description.

- Missing at Random (MAR)
  - dyad parameter = p = Prob(Dyad(i,j) is observed)
  - node parameter = p = Prob(Node i is observed)
  - covar-dyad": parameter = beta in R^M, such that Prob(Dyad (i,j) is observed) = logistic(parameter' covarArray (i,j, .))
  - covar-node": parameter = nu in R^M such that Prob(Node i is observed) = logistic(parameter' covarMatrix (i,)
  - snowball": parameter = number of waves with Prob(Node i is observed in the 1st wave)
- Missing Not At Random (MNAR)
  - double-standard parameter = (p0,p1) with p0 = Prob(Dyad (i,j) is observed | the dyad is equal to 0), p1 = Prob(Dyad (i,j) is observed | the dyad is equal to 1)
  - block-node parameter = c(p(1),...,p(Q)) and p(q) = Prob(Node i is observed | node i is in cluster q)
  - block-dyad parameter = c(p(1,1),...,p(Q,Q)) and p(q,l) = Prob(Edge (i,j) is observed | node i is in cluster q and node j is in cluster l)

Value

Returns an R6 object with class `missSBM_collection`.

See Also

`observeNetwork`, `missSBM_collection`, `missSBM_fit` and `missSBM_param`.

Examples

```
## SBM parameters
N <- 100 # number of nodes
Q <- 3   # number of clusters
pi <- rep(1,Q)/Q     # block proportion
theta <- list(mean = diag(.45,Q) + .05 ) # connectivity matrix

## Sampling parameters
samplingParameters <- .75 # the sampling rate
sampling  <- "dyad"      # the sampling design

## generate a undirected binary SBM with no covariate
sbm <- sbm::sampleSimpleSBM(N, pi, theta)

## Uncomment to set parallel computing with future
## future::plan("multicore", workers = 2)

## Sample some dyads data + Infer SBM with missing data
collection <-
   observeNetwork(sbm$networkData, sampling, samplingParameters) %>%
   estimateMissSBM(vBlocks = 1:4, sampling = sampling)
plot(collection, "monitoring")
plot(collection, "icl")

collection$ICL
coef(collection$bestModel$fittedSBM, "connectivity")

myModel <- collection$bestModel
plot(myModel, "expected")
plot(myModel, "imputed")
plot(myModel, "meso")
coef(myModel, "sampling")
coef(myModel, "connectivity")
predict(myModel)[1:5, 1:5]
```

`fitted.missSBM_fit` *Extract model fitted values from object* `missSBM_fit`*, return by* `estimateMissSBM()`

Description

Extract model fitted values from object `missSBM_fit`, return by `estimateMissSBM()`

Usage

```
## S3 method for class 'missSBM_fit'
fitted(object, ...)
```

Arguments

object	an R6 object with class missSBM_fit
...	additional parameters for S3 compatibility.

Value

A matrix of estimated probabilities of connection

---

frenchblog2007 *Political Blogosphere network prior to 2007 French presidential election*

Description

French Political Blogosphere network dataset consists of a single day snapshot of over 200 political blogs automatically extracted the 14 October 2006 and manually classified by the "Observatoire Présidentielle" project. Originally part of the 'mixer' package

Usage

```
frenchblog2007
```

Format

An igraph object with 196 nodes. The vertex attribute "party" provides a possible clustering of the nodes.

Source

https://www.meltwater.com/en/suite/consumer-intelligence?utm_source=direct&utm_medium=linkfluence

Examples

```
data(frenchblog2007)
igraph::V(frenchblog2007)$party
igraph::plot.igraph(frenchblog2007,
  vertex.color = factor(igraph::V(frenchblog2007)$party),
  vertex.label = NA
 )
```

l1_similarity *L1-similarity*

Description

Compute l1-similarity between two vectors

Usage

```
l1_similarity(x, y)
```

Arguments

x	a vector
y	a vector

Value

a vector equal to -abs(x-y)

Examples

```
l1_similarity(1:5, 5:1)
```

missSBM_collection *An R6 class to represent a collection of SBM fits with missing data*

Description

The function `estimateMissSBM()` fits a collection of SBM with missing data for a varying number of block. These models with class `missSBM_fit` are stored in an instance of an object with class `missSBM_collection`, described here.

Fields are accessed via active binding and cannot be changed by the user.

This class comes with a set of R6 methods, some of them being useful for the user and exported as S3 methods. See the documentation for `show()` and `print()`

Active bindings

`models` a list of models

`ICL` the vector of Integrated Classification Criterion (ICL) associated to the models in the collection (the smaller, the better)

`bestModel` the best model according to the ICL, restricted to models without collapsed classes when at least one such model is available (see `$degenerate`)

`vBlocks` a vector with the number of blocks

`occupiedBlocks` a vector with the number of classes actually occupied in each model (see missSBM_fit's `occupiedBlocks`)

`degenerate` logical vector, `TRUE` for models with collapsed classes (`occupiedBlocks < vBlocks`, see missSBM_fit's `repair()`)

`optimizationSettings` the control list used by estimate()/polish()/explore() when not overridden per call (set at construction by `estimateMissSBM()`)

`optimizationStatus` a data.frame summarizing the optimization process for all models

Methods

Public methods:

- `missSBM_collection$new()`
- `missSBM_collection$estimate()`
- `missSBM_collection$estimate_chain()`
- `missSBM_collection$polish()`
- `missSBM_collection$explore()`
- `missSBM_collection$plot()`
- `missSBM_collection$show()`
- `missSBM_collection$print()`
- `missSBM_collection$clone()`

`missSBM_collection$new():` constructor for networkSampling

Usage:

```
missSBM_collection$new(partlyObservedNet, sampling, clusterInit, control)
```

Arguments:

`partlyObservedNet` An object with class `partlyObservedNetwork`.

`sampling` The sampling design for the modelling of missing data: MAR designs ("dyad", "node") and MNAR designs ("double-standard", "block-dyad", "block-node" ,"degree")

`clusterInit` Initial clustering: a list of vectors, each with size `ncol(adjacencyMatrix)`.

`control` a list of parameters controlling advanced features. Only 'trace' and 'useCov' are relevant here. See `estimateMissSBM()` for details.

`missSBM_collection$estimate():` method to launch the estimation of the collection of models

Usage:

```
missSBM_collection$estimate(control = NULL)
```

Arguments:

`control` optional list of parameters overriding the collection's stored control (set at construction by `estimateMissSBM()`, see its details for the full list). Default `NULL` uses the stored control as-is.

`missSBM_collection$estimate_chain():` alternative to `estimate()`: fits each model in increasing order of number of blocks, initializing `vBlocks[k]` by splitting (see missSBM_fit's `split()`/`candidates_split()`) the already-converged model at `vBlocks[k-1]` instead of an independent, cold spectral clustering. Meant to reduce VEM component collapse at higher numbers

of blocks (see `$degenerate`), at the cost of being sequential in the number of blocks (unlike `estimate()`, which fits every model in parallel) – can be slower in wall-clock time with many workers available. Falls back to this slot's own cold-started clustering (built at construction, same as `estimate()` would use) whenever nothing is splittable along the chain.

Usage:

```
missSBM_collection$estimate_chain(control = NULL)
```

Arguments:

`control` optional list of parameters overriding the collection's stored control (set at construction by `estimateMissSBM()`, see its details for the full list). Default `NULL` uses the stored control as-is.

`missSBM_collection$polish()`: method to node-swap-polish every model in the collection (see missSBM_fit's `polish()`); fixes individually misclassified nodes at each model's own number of blocks, unlike `explore()` which searches across blocks.

Usage:

```
missSBM_collection$polish(control = NULL)
```

Arguments:

`control` optional list of parameters overriding the collection's stored control (set at construction by `estimateMissSBM()`, see its details for the full list). Default `NULL` uses the stored control as-is.

`missSBM_collection$explore()`: method for performing exploration of the ICL (split/merge search across numbers of blocks, see missSBM_fit's `candidates_split()`/`candidates_merge()`). Uses the collection's stored control by default; `iterates` lets the caller override it for this call only, without altering the stored control – handy to alternate `explore()`/`polish()` calls without having to reconstruct a full control list each time. `iterates <= 0` is a no-op.

Usage:

```
missSBM_collection$explore(control = NULL, iterates = NULL, direction = "both")
```

Arguments:

`control` optional list of parameters overriding the collection's stored control (set at construction by `estimateMissSBM()`, see its details for the full list). Default `NULL` uses the stored control as-is.

`iterates` optional integer overriding `control$iterates` for this call only.

`direction` character ("forward", "backward", "both" or "none") controlling which directions are searched. Default "both".

`missSBM_collection$plot()`: plot method for missSBM_collection

Usage:

```
missSBM_collection$plot(type = c("icl", "elbo", "monitoring"))
```

Arguments:

`type` the type specifies the field to plot, either "icl", "elbo" or "monitoring". Default is "icl"

`missSBM_collection$show()`: show method for missSBM_collection

Usage:

```
missSBM_collection$show()
```

`missSBM_collection$print()`: User friendly print method

Usage:

```
missSBM_collection$print()
```

`missSBM_collection$clone()`: The objects of this class are cloneable with this method.

Usage:

```
missSBM_collection$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Uncomment to set parallel computing with future
## future::plan("multicore", workers = 2)

## Sample 75% of dyads in  French political Blogosphere's network data
adjacencyMatrix <- missSBM::frenchblog2007 %>%
  igraph::delete.vertices(1:100) %>%
  igraph::as_adjacency_matrix() %>%
  missSBM::observeNetwork(sampling = "dyad", parameters = 0.75)
collection <- estimateMissSBM(adjacencyMatrix, 1:5, sampling = "dyad")
class(collection)
```

`missSBM_fit` *An R6 class to represent an SBM fit with missing data*

Description

The function `estimateMissSBM()` fits a collection of SBM for varying number of block. Each fitted SBM is an instance of an R6 object with class `missSBM_fit`, described here.

Fields are accessed via active binding and cannot be changed by the user.

This class comes with a set of R6 methods, some of them being useful for the user and exported as S3 methods. See the documentation for `show()`, `print()`, `fitted()`, `predict()`, `plot()`.

Active bindings

`fittedSBM` the fitted SBM with class `SimpleSBM_fit_noCov`, `SimpleSBM_fit_withCov` or `SimpleSBM_fit_MNAR` inheriting from class `sbm::SimpleSBM_fit`

`fittedSampling` the fitted sampling, inheriting from class `networkSampling` and corresponding fits

`imputedNetwork` The network data as a matrix with NAs values imputed with the current model

`monitoring` a list carrying information about the optimization process

`occupiedBlocks` the number of classes actually occupied by at least one node; can be less than `fittedSBM$nbBlocks` for an over-specified fit whose VEM has collapsed one or more classes (see `repair()`)

`degenerate` `TRUE` if `occupiedBlocks` < `fittedSBM$nbBlocks`

`entropyImputed` the entropy of the distribution of the imputed dyads

`entropy` the entropy due to the distribution of the imputed dyads and of the clustering

`vExpec` double: variational expectation of the complete log-likelihood

`penalty` double, value of the penalty term in ICL

`loglik` double: approximation of the log-likelihood (variational lower bound) reached

`ICL` double: value of the integrated classification log-likelihood

Methods

Public methods:

- `missSBM_fit$new()`
- `missSBM_fit$doVEM()`
- `missSBM_fit$split()`
- `missSBM_fit$candidates_split()`
- `missSBM_fit$merge()`
- `missSBM_fit$candidates_merge()`
- `missSBM_fit$repair()`
- `missSBM_fit$polish()`
- `missSBM_fit$show()`
- `missSBM_fit$print()`
- `missSBM_fit$clone()`

`missSBM_fit$new()`: constructor for networkSampling

Usage:

```
missSBM_fit$new(partlyObservedNet, netSampling, clusterInit, useCov = TRUE)
```

Arguments:

`partlyObservedNet` An object with class `partlyObservedNetwork`.

`netSampling` The sampling design for the modelling of missing data: MAR designs ("dyad", "node") and MNAR designs ("double-standard", "block-dyad", "block-node" ,"degree")

`clusterInit` Initial clustering: a vector with size `ncol(adjacencyMatrix)`, providing a user-defined clustering. The number of blocks is deduced from the number of levels in with `clusterInit`.

`useCov` logical. If covariates are present in partlyObservedNet, should they be used for the inference or of the network sampling design, or just for the SBM inference? default is TRUE.

`missSBM_fit$doVEM()`: a method to perform inference of the current missSBM fit with variational EM

Usage:

```
missSBM_fit$doVEM(
  control = list(threshold = 0.01, maxIter = 100, fixPointIter = 3, trace = TRUE)
)
```

Arguments:

`control` a list of VEM control parameters (see `estimateMissSBM()`)

`missSBM_fit$split()`: clone of the current fit after splitting cluster `index` in two, via a spectral bipartition of the sub-network it induces. Builds but does not fit the candidate (see `candidates_split()`).

Usage:

```
missSBM_fit$split(index, in_place = FALSE, base_net = NULL)
```

Arguments:

`index` index (integer) of the cluster to split

`in_place` replace `self`'s own fit (`TRUE`) or return a new object (`FALSE`, the default)?

`base_net` optional precomputed network to bipartition (as built internally at the top of this method); lets `candidates_split()` avoid recomputing it once per candidate.

Returns: a new `missSBM_fit` with one more block, or `NULL` if `index` cannot be split (its induced sub-network has zero variance)

`missSBM_fit$candidates_split()`: generate and cheaply trial-fit candidates obtained by splitting each splittable cluster in two (see `split()`). A cluster is splittable if it has at least 4 members and non-zero variance in its induced sub-network.

Usage:

```
missSBM_fit$candidates_split(
  control = list(threshold = 0.01, maxIter = 100, fixPointIter = 3, trace = TRUE),
   trial_niter = 2
)
```

Arguments:

`control` a list of VEM control parameters (see `estimateMissSBM()`); `maxIter` is overridden by `trial_niter`

`trial_niter` number of VEM iterations used for the trial fits. Default is 2.

Returns: a list of trial-fitted `missSBM_fit` candidates (one per splittable cluster)

`missSBM_fit$merge()`: clone of the current fit after merging clusters `indices[1]` and `indices[2]` into one. Builds but does not fit the candidate (see `candidates_merge()`).

Usage:

```
missSBM_fit$merge(indices, in_place = FALSE)
```

Arguments:

`indices` indices (couple of integers) of the clusters to merge

`in_place` replace `self`'s own fit (`TRUE`) or return a new object (`FALSE`, the default)?

Returns: a new `missSBM_fit` with one fewer block

`missSBM_fit$candidates_merge()`: generate and cheaply trial-fit candidates obtained by merging pairs of clusters (see `merge()`). Beyond `max_candidates` pairs (quadratic in the number of blocks), only the most similar-connectivity pairs are tried.

Usage:

```
missSBM_fit$candidates_merge(
  control = list(threshold = 0.01, maxIter = 100, fixPointIter = 3, trace = TRUE),
   max_candidates = 30,
   trial_niter = 2
)
```

Arguments:

`control` a list of VEM control parameters (see `estimateMissSBM()`); `maxIter` is overridden by `trial_niter`

`max_candidates` cap on the number of pairs tried. Default is 30.

`trial_niter` number of VEM iterations used for the trial fits. Default is 2.

Returns: a list of trial-fitted `missSBM_fit` candidates

`missSBM_fit$repair()`: recovers a degenerate fit (fewer occupied classes than its structural `nbBlocks`, e.g. after a VEM component collapse) by filling the empty classes (see `repair_empty_classes()`) and refitting the full VEM. Mutates `self` in place; a no-op if the fit is not degenerate.

Usage:

```
missSBM_fit$repair(
  control = list(threshold = 0.01, maxIter = 100, fixPointIter = 3, trace = TRUE)
)
```

Arguments:

`control` a list of VEM control parameters (see `estimateMissSBM()`)

Returns: invisibly, `self`

`missSBM_fit$polish()`: discrete node-swap polishing (Kernighan-Lin / greedy-ICL style): after VEM convergence, tau is near-hard and its fixed point cannot relocate a single misclassified node (only `split()`/`merge()` fix group-level mistakes). Each sweep computes, for every node, the closed-form complete-data log-likelihood gain of moving it to its best alternative class (theta/pi held fixed), applies the improving, non-class-emptying moves, then runs a full VEM to resettle. Stops as soon as a sweep fails to improve the ICL, mutates `self` in place, and never leaves the ICL worse than before the call.

Usage:

```
missSBM_fit$polish(
  control = list(threshold = 0.01, maxIter = 100, fixPointIter = 3, trace = TRUE),
   max_sweeps = 10
)
```

Arguments:

`control` a list of VEM control parameters (see `estimateMissSBM()`)

`max_sweeps` maximum number of swap sweeps. Default is 10.

Returns: invisibly, `self`

`missSBM_fit$show()`: show method for missSBM_fit

Usage:

```
missSBM_fit$show()
```

`missSBM_fit$print()`: User friendly print method

Usage:

```
missSBM_fit$print()
```

`missSBM_fit$clone()`: The objects of this class are cloneable with this method.

Usage:

```
missSBM_fit$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
## Sample 75% of dyads in  French political Blogosphere's network data
adjMatrix <- missSBM::frenchblog2007 %>%
  igraph::as_adjacency_matrix(sparse = FALSE) %>%
  missSBM::observeNetwork(sampling = "dyad", parameters = 0.75)
collection <- estimateMissSBM(adjMatrix, 3:5, sampling = "dyad")
my_missSBM_fit <- collection$bestModel
class(my_missSBM_fit)
plot(my_missSBM_fit, "imputed")
```

`missSBM_param` *Control of a missSBM fit*

Description

Helper to define the list of parameters that controls `estimateMissSBM()`. All arguments have defaults.

Usage

```
missSBM_param(
  threshold = 0.01,
  maxIter = 50,
  fixPointIter = 3,
  imputation = c("median", "average", "zero"),
  similarity = l1_similarity,
  useCov = TRUE,
  clusterInit = NULL,
  polish = TRUE,
  iterates = 1,
  maxMergeCandidates = 30,
  stopOnDegenerate = TRUE,
  maxConsecutiveDegenerate = 2,
  warmChain = FALSE,
  trace = TRUE
)
```

Arguments

`threshold`	V-EM algorithm stops when an optimization step changes the objective function or the parameters by less than threshold. Default is 1e-2.
`maxIter`	V-EM algorithm stops when the number of iteration exceeds maxIter. Default is 50.
`fixPointIter`	number of fix-point iterations in the V-E step. Default is 3.
`imputation`	character, the imputation strategy used to build the initial clustering (see `partlyObservedNetwork`'s `imputation()`). Either "median" (default), "average" or "zero".
`similarity`	an R x R -> R function to compute similarities between node covariates. Default is `l1_similarity`, that is, -abs(x - y). Only relevant when the covariates are node-centered (i.e. `covariates` is a list of size-N vectors).
`useCov`	logical. If `covariates` is not empty, should they be used for the SBM inference (or just for the sampling)? Default is TRUE.
`clusterInit`	initial clustering: `NULL` (default) for a cold spectral clustering, or a list with `length(vBlocks)` vectors, each with size `ncol(adjacencyMatrix)`, providing a user-defined clustering.
`polish`	logical, should each model be node-swap-polished (see missSBM_fit's `polish()`) after the initial VEM fit, fixing individually misclassified nodes at that model's own number of blocks? Cheap relative to exploration (`iterates`), since it does not search across numbers of blocks. Default is TRUE.
`iterates`	integer, the number of forward/backward exploration passes searching for a better number of blocks by splitting/merging clusters (unlike `polish`, which only refines each model at its own number of blocks): more expensive than `polish`. `0` disables exploration entirely (only the initial VEM fit, plus `polish` if enabled, is returned). Default is 1.
`maxMergeCandidates`	integer, caps the number of cluster-pair merge candidates tried during backward exploration (quadratic in the number of blocks otherwise). Beyond this cap, only the pairs with the most similar fitted connectivity profiles are tried, since merging two blocks with very different connectivity is rarely competitive anyway. Default is 30.
`stopOnDegenerate`	logical. A requested number of blocks can be higher than what the network actually supports: VEM then collapses one or more classes (see missSBM_fit's `repair()`), and even a fair recovery attempt (`repair()`, then a full exploration pass letting `explore_forward()` try to split a healthy neighbor into that range) can still collapse right back down. When this persists over `maxConsecutiveDegenerate` consecutive (increasing) values of `vBlocks` after a pass, forward (split) exploration stops growing further into that range on subsequent passes (only relevant when `iterates` > 1); the affected models stay as fitted, flagged via `$degenerate`, and a warning is issued. Default is TRUE.
`maxConsecutiveDegenerate`	integer, the run length that triggers `stopOnDegenerate`. Default is 2.
`warmChain`	logical (**work in progress**). If `TRUE`, each model is initialized by splitting (see missSBM_fit's `split()`) the already-converged, smaller neighbor in `vBlocks`

	instead of an independent cold spectral clustering (see missSBM_collection's `estimate_chain()`) – meant to reduce VEM component collapse at higher numbers of blocks. Sequential in the number of blocks, unlike the default (`FALSE`), which fits every model in parallel: can be slower in wall-clock time when several workers are available. Default is FALSE.
`trace`	logical for verbosity. Default is TRUE.

Value

a list of parameters configuring the fit, with class `missSBM_param`.

See Also

`estimateMissSBM()`

Examples

```
my_control <- missSBM_param(iterates = 2, polish = FALSE)
my_control$iterates
```

---

networkSampler *Definition of R6 Class 'networkSampling_sampler'*

Description

This class is use to define a sampling model for a network. Inherits from 'networkSampling'. Owns a rSampling method which takes an adjacency matrix as an input and send back an object with class partlyObservedNetwork.

Super class

`networkSampling` -> `networkSampler`

Active bindings

`samplingMatrix` a matrix of logical indicating observed entries

Methods

Public methods:

- `networkSampler$new()`
- `networkSampler$rSamplingMatrix()`
- `networkSampler$clone()`

`networkSampler$new()`: constructor for networkSampling

Usage:

```
networkSampler$new(type = NA, parameters = NA, nbNodes = NA, directed = FALSE)
```

Arguments:

`type` character for the type of sampling. must be in ("dyad", "covar-dyad", "node", "covar-node", "block-node", "block-dyad", "double-standard", "degree")

`parameters` the vector of parameters associated to the sampling at play

`nbNodes` number of nodes in the network

`directed` logical, directed network of not

`networkSampler$rSamplingMatrix():` a method for drawing a sampling matrix according to the current sampling design

Usage:

```
networkSampler$rSamplingMatrix()
```

`networkSampler$clone():` The objects of this class are cloneable with this method.

Usage:

```
networkSampler$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`partlyObservedNetwork`

`networkSampling` *Definition of R6 Class 'networkSampling'*

Description

this virtual class is the mother of all subtypes of networkSampling (either sampler or fit) It is used to define a sampling model for a network. It has a rSampling method which takes an adjacency matrix as an input and send back an object with class partlyObservedNetwork.

Active bindings

`type` a character for the type of sampling

`parameters` the vector of parameters associated with the sampling at play

`df` the number of entries in the vector of parameters

Methods

Public methods:

- `networkSampling$new()`
- `networkSampling$show()`
- `networkSampling$print()`
- `networkSampling$clone()`

`networkSampling$new()`: constructor for networkSampling

Usage:

```
networkSampling$new(type = NA, parameters = NA)
```

Arguments:

`type` character for the type of sampling. must be in ("dyad", "covar-dyad", "node", "covar-node", "block-node", "block-dyad", "double-standard", "degree")

`parameters` the vector of parameters associated to the sampling at play

`networkSampling$show()`: show method

Usage:

```
networkSampling$show(
  type = paste0(private$name, "-model for network sampling\n")
)
```

Arguments:

`type` character used to specify the type of sampling

`networkSampling$print()`: User friendly print method

Usage:

```
networkSampling$print()
```

`networkSampling$clone()`: The objects of this class are cloneable with this method.

Usage:

```
networkSampling$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`networkSamplingDyads_fit` *Virtual class used to define a family of networkSamplingDyads_fit*

Description

Virtual class used to define a family of networkSamplingDyads_fit

Super class

`networkSampling` -> `networkSamplingDyads_fit`

Active bindings

`penalty` double, value of the penalty term in ICL

`log_lambda` double, term for adjusting the imputation step which depends on the type of sampling

Methods

Public methods:

- networkSamplingDyads_fit$new()
- networkSamplingDyads_fit$show()
- networkSamplingDyads_fit$update_parameters()
- networkSamplingDyads_fit$update_imputation()
- networkSamplingDyads_fit$clone()

`networkSamplingDyads_fit$new()`: constructor for networkSampling_fit

Usage:

```
networkSamplingDyads_fit$new(partlyObservedNetwork, name)
```

Arguments:

`partlyObservedNetwork` a object with class partlyObservedNetwork representing the observed data with possibly missing entries

`name` a character for the name of sampling to fit on the partlyObservedNetwork

`networkSamplingDyads_fit$show()`: show method

Usage:

```
networkSamplingDyads_fit$show()
```

`networkSamplingDyads_fit$update_parameters()`: a method to update the estimation of the parameters. By default, nothing to do (corresponds to MAR sampling)

Usage:

```
networkSamplingDyads_fit$update_parameters(...)
```

Arguments:

`...` use for compatibility

`networkSamplingDyads_fit$update_imputation()`: a method to update the imputation of the missing entries.

Usage:

```
networkSamplingDyads_fit$update_imputation(nu)
```

Arguments:

`nu` the matrix of (uncorrected) imputation for missing entries

`networkSamplingDyads_fit$clone()`: The objects of this class are cloneable with this method.

Usage:

```
networkSamplingDyads_fit$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

networkSamplingNodes_fit *Virtual class used to define a family of networkSamplingNodes_fit*

Description

Virtual class used to define a family of networkSamplingNodes_fit

Super class

networkSampling -> networkSamplingNodes_fit

Active bindings

penalty double, value of the penalty term in ICL

log_lambda double, term for adjusting the imputation step which depends on the type of sampling

Methods

Public methods:

- networkSamplingNodes_fit$new()
- networkSamplingNodes_fit$show()
- networkSamplingNodes_fit$update_parameters()
- networkSamplingNodes_fit$update_imputation()
- networkSamplingNodes_fit$clone()

networkSamplingNodes_fit$new(): constructor

Usage:

```
networkSamplingNodes_fit$new(partlyObservedNetwork, name)
```

Arguments:

partlyObservedNetwork a object with class partlyObservedNetwork representing the observed data with possibly missing entries

name a character for the name of sampling to fit on the partlyObservedNetwork

networkSamplingNodes_fit$show(): show method

Usage:

```
networkSamplingNodes_fit$show()
```

networkSamplingNodes_fit$update_parameters(): a method to update the estimation of the parameters. By default, nothing to do (corresponds to MAR sampling)

Usage:

```
networkSamplingNodes_fit$update_parameters(...)
```

Arguments:

... use for compatibility

`networkSamplingNodes_fit$update_imputation()`: a method to update the imputation of the missing entries.

Usage:

```
networkSamplingNodes_fit$update_imputation(nu)
```

Arguments:

`nu` the matrix of (uncorrected) imputation for missing entries

`networkSamplingNodes_fit$clone()`: The objects of this class are cloneable with this method.

Usage:

```
networkSamplingNodes_fit$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`nodeSampler` *Virtual class for all node-centered samplers*

Description

Virtual class for all node-centered samplers

Super classes

`networkSampling` -> `networkSampler` -> `nodeSampler`

Methods

Public methods:

- `nodeSampler$clone()`

`nodeSampler$clone()`: The objects of this class are cloneable with this method.

Usage:

```
nodeSampler$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

nodeSampling_fit *Class for fitting a node sampling*

Description

Class for fitting a node sampling

Super classes

networkSampling -> networkSamplingNodes_fit -> nodeSampling_fit

Active bindings

vExpec variational expectation of the sampling

Methods

Public methods:

- nodeSampling_fit$new()
- nodeSampling_fit$clone()

nodeSampling_fit$new(): constructor

Usage:

```
nodeSampling_fit$new(partlyObservedNetwork, ...)
```

Arguments:

partlyObservedNetwork a object with class partlyObservedNetwork representing the observed data with possibly missing entries

... used for compatibility

nodeSampling_fit$clone(): The objects of this class are cloneable with this method.

Usage:

```
nodeSampling_fit$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

`observeNetwork` *Observe a network partially according to a given sampling design*

Description

This function draws observations in an adjacency matrix according to a given network sampling design.

Usage

```
observeNetwork(
  adjacencyMatrix,
  sampling,
  parameters,
  clusters = NULL,
  covariates = list(),
  similarity = l1_similarity,
  intercept = 0
)
```

Arguments

`adjacencyMatrix`	The N x N adjacency matrix of the network to sample. The diagonal is expected to be NA (no self-loops); any other pre-existing NA entry is treated as an absent edge (coded 0) before sampling is applied on top of it, with a warning.
`sampling`	The sampling design used to observe the adjacency matrix, see details.
`parameters`	The sampling parameters (adapted to each sampling, see details).
`clusters`	An optional clustering membership vector of the nodes. Only necessary for block samplings.
`covariates`	An optional list with M entries (the M covariates). If the covariates are node-centered, each entry of `covariates`. must be a size-N vector; if the covariates are dyad-centered, each entry of `covariates` must be N x N matrix.
`similarity`	An optional function to compute similarities between node covariates. Default is `l1_similarity`, that is, -abs(x-y). Only relevant when the covariates are node-centered.
`intercept`	An optional intercept term to be added in case of the presence of covariates. Default is 0.

Details

Internal functions use `future_lapply`, so set your plan to `'multisession'` or `'multicore'` to use several cores/workers.

The different sampling designs are split into two families in which we find dyad-centered and node-centered samplings. See doi:10.1080/01621459.2018.1562934 for a complete description.

- Missing at Random (MAR)
  - dyad parameter = p = Prob(Dyad(i,j) is observed)
  - node parameter = p = Prob(Node i is observed)
  - covar-dyad": parameter = beta in R^M, such that Prob(Dyad (i,j) is observed) = logistic(parameter' covarArray (i,j, .))
  - covar-node": parameter = nu in R^M such that Prob(Node i is observed) = logistic(parameter' covarMatrix (i,)
  - snowball": parameter = number of waves with Prob(Node i is observed in the 1st wave)
- Missing Not At Random (MNAR)
  - double-standard parameter = (p0,p1) with p0 = Prob(Dyad (i,j) is observed | the dyad is equal to 0), p1 = Prob(Dyad (i,j) is observed | the dyad is equal to 1)
  - block-node parameter = c(p(1),...,p(Q)) and p(q) = Prob(Node i is observed | node i is in cluster q)
  - block-dyad parameter = c(p(1,1),...,p(Q,Q)) and p(q,l) = Prob(Edge (i,j) is observed | node i is in cluster q and node j is in cluster l)

Value

an adjacency matrix with the same dimension as the input, yet with additional NAs.

Examples

```
## SBM parameters
N <- 300 # number of nodes
Q <- 3   # number of clusters
pi <- rep(1,Q)/Q     # block proportion
theta <- list(mean = diag(.45,Q) + .05 ) # connectivity matrix

## simulate an unidirected binary SBM without covariate
sbm <- sbm::sampleSimpleSBM(N, pi, theta)

## Sample network data

# some sampling design and their associated parameters
sampling_parameters <- list(
   "dyad" = .3,
   "node" = .3,
   "double-standard" = c(0.4, 0.8),
   "block-node" = c(.3, .8, .5),
   "block-dyad" = theta$mean,
   "degree" = c(.01, .01),
   "snowball" = c(2,.1)
 )

observed_networks <- list()

for (sampling in names(sampling_parameters)) {
  observed_networks[[sampling]] <-
     missSBM::observeNetwork(
```

```
        adjacencyMatrix = sbm$networkData,
        sampling        = sampling,
        parameters      = sampling_parameters[[sampling]],
        clusters        = sbm$memberships
      )
}
```

`partlyObservedNetwork` *An R6 Class used for internal representation of a partially observed network*

Description

This class is not exported to the user

Active bindings

`samplingRate` The percentage of observed dyads

`nbNodes` The number of nodes

`nbDyads` The number of dyads

`is_directed` logical indicating if the network is directed or not

`networkData` The adjacency matrix of the network

`covarArray` the array of covariates

`covarMatrix` the matrix of covariates

`samplingMatrix` matrix of observed and non-observed edges

`samplingMatrixBar` matrix of observed and non-observed edges

`observedNodes` a vector of observed and non-observed nodes (observed means at least one non NA value)

Methods

Public methods:

- `partlyObservedNetwork$new()`
- `partlyObservedNetwork$clustering()`
- `partlyObservedNetwork$imputation()`
- `partlyObservedNetwork$clone()`

`partlyObservedNetwork$new()`: constructor

Usage:

```
partlyObservedNetwork$new(
  adjacencyMatrix,
  covariates = list(),
  similarity = l1_similarity
)
```

Arguments:

`adjacencyMatrix` The adjacency matrix of the network

`covariates` A list with M entries (the M covariates), each of whom being either a size-N vector or N x N matrix.

`similarity` An R x R -> R function to compute similarities between node covariates. Default is `l1_similarity`, that is, -abs(x-y).

`partlyObservedNetwork$clustering():` method to cluster network data with missing value

Usage:

```
partlyObservedNetwork$clustering(
  vBlocks,
  imputation = ifelse(is.null(private$phi), "median", "average")
)
```

Arguments:

`vBlocks` The vector of number of blocks considered in the collection.

`imputation` character indicating the type of imputation among "median", "average"

`partlyObservedNetwork$imputation():` basic imputation from existing clustering

Usage:

```
partlyObservedNetwork$imputation(type = c("median", "average", "zero"))
```

Arguments:

`type` a character, the type of imputation. Either "median" or "average"

`partlyObservedNetwork$clone():` The objects of this class are cloneable with this method.

Usage:

```
partlyObservedNetwork$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`plot.missSBM_fit` *Visualization for an object* `missSBM_fit`

Description

Plot function for the various fields of a `missSBM_fit`: the fitted SBM (network or connectivity), and a plot monitoring the optimization.

Usage

```
## S3 method for class 'missSBM_fit'
plot(
  x,
  type = c("imputed", "expected", "meso", "monitoring"),
  dimLabels = list(row = "node", col = "node"),
  ...
)
```

Arguments

x	an object with class missSBM_fit
type	the type specifies the field to plot, either "imputed", "expected", "meso", or "monitoring"
dimLabels	: a list of two characters specifying the labels of the nodes. Default to list(row='node',col = 'node'))
...	additional parameters for S3 compatibility. Not used

Value

a ggplot object

---

predicted.missSBM_fit *Prediction of a missSBM_fit (i.e. network with imputed missing dyads)*

Description

Prediction of a missSBM_fit (i.e. network with imputed missing dyads)

Usage

```
## S3 method for class 'missSBM_fit'
predict(object, ...)
```

Arguments

object	an R6 object with class missSBM_fit
...	additional parameters for S3 compatibility.

Value

an adjacency matrix between pairs of nodes. Missing dyads are imputed with their expected values, i.e. by there estimated probabilities of connection under the missing SBM.

simpleDyadSampler *Class for defining a simple dyad sampler*

Description

Class for defining a simple dyad sampler

Super classes

networkSampling -> networkSampler -> dyadSampler -> simpleDyadSampler

Methods

Public methods:

- simpleDyadSampler$new()
- simpleDyadSampler$clone()

simpleDyadSampler$new(): constructor for networkSampling

Usage:

```
simpleDyadSampler$new(
  parameters = NA,
  nbNodes = NA,
  directed = FALSE,
  covarArray = NULL,
  intercept = 0
)
```

Arguments:

parameters the vector of parameters associated to the sampling at play

nbNodes number of nodes in the network

directed logical, directed network of not

covarArray an array of covariates used

intercept double, intercept term used to compute the probability of sampling in the presence of covariates. Default 0.

simpleDyadSampler$clone(): The objects of this class are cloneable with this method.

Usage:

```
simpleDyadSampler$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

simpleNodeSampler *Class for defining a simple node sampler*

Description

Class for defining a simple node sampler

Super classes

networkSampling -> networkSampler -> nodeSampler -> simpleNodeSampler

Methods

Public methods:

- simpleNodeSampler$new()
- simpleNodeSampler$clone()

simpleNodeSampler$new(): constructor for networkSampling

Usage:

```
simpleNodeSampler$new(
  parameters = NA,
  nbNodes = NA,
  directed = FALSE,
  covarMatrix = NULL,
  intercept = 0
)
```

Arguments:

parameters the vector of parameters associated to the sampling at play

nbNodes number of nodes in the network

directed logical, directed network of not

covarMatrix a matrix of covariates used

intercept double, intercept term used to compute the probability of sampling in the presence of covariates. Default 0.

simpleNodeSampler$clone(): The objects of this class are cloneable with this method.

Usage:

```
simpleNodeSampler$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

SimpleSBM_fit *Base internal class for adjusting a binary Stochastic Block Model in the context of missSBM.*

Description

It is not designed to be called directly by the user; see the concrete variants SimpleSBM_fit_noCov, SimpleSBM_fit_withCov and SimpleSBM_fit_MNAR.

Super classes

sbm::SBM -> sbm::SimpleSBM -> SimpleSBM_fit

Active bindings

type the type of SBM (distribution of edges values, network type, presence of covariates)

penalty double, value of the penalty term in ICL

entropy double, value of the entropy due to the clustering distribution

loglik double: approximation of the log-likelihood (variational lower bound) reached

ICL double: value of the integrated classification log-likelihood

Methods

Public methods:

- SimpleSBM_fit$new()
- SimpleSBM_fit$doVEM()
- SimpleSBM_fit$reorder()
- SimpleSBM_fit$get_state()
- SimpleSBM_fit$set_state()
- SimpleSBM_fit$clone()

SimpleSBM_fit$new(): constructor for simpleSBM_fit for missSBM purpose

Usage:

```
SimpleSBM_fit$new(networkData, clusterInit, covarList = list())
```

Arguments:

networkData a structure to store network under missing data condition: either a matrix possibly with NA, or a missSBM:::partlyObservedNetwork

clusterInit Initial clustering: a vector with size ncol(adjacencyMatrix), providing a user-defined clustering with nbBlocks levels.

covarList An optional list with M entries (the M covariates).

SimpleSBM_fit$doVEM(): method to perform estimation via variational EM

Usage:

```
SimpleSBM_fit$doVEM(
  threshold = 0.01,
  maxIter = 100,
  fixPointIter = 3,
  trace = FALSE
)
```

Arguments:

`threshold` stop when an optimization step changes the objective function by less than threshold. Default is 1e-4.

`maxIter` V-EM algorithm stops when the number of iteration exceeds maxIter. Default is 10

`fixPointIter` number of fix-point iterations in the Variational E step. Default is 5.

`trace` logical for verbosity. Default is `FALSE`.

`SimpleSBM_fit$reorder()`: permute group labels by order of decreasing probability

Usage:

```
SimpleSBM_fit$reorder()
```

`SimpleSBM_fit$get_state()`: a lightweight snapshot of the mutable VEM state (as opposed to `clone()`, which duplicates the whole object, including the – possibly large – network data)

Usage:

```
SimpleSBM_fit$get_state()
```

`SimpleSBM_fit$set_state()`: restore a state previously returned by `get_state()`

Usage:

```
SimpleSBM_fit$set_state(state)
```

Arguments:

`state` a state, as returned by `get_state()`

`SimpleSBM_fit$clone()`: The objects of this class are cloneable with this method.

Usage:

```
SimpleSBM_fit$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`SimpleSBM_fit_MNAR` *Internal class for a binary SBM fit under MNAR sampling designs (double-standard, block-node, block-dyad).*

Description

It is not designed to be called directly by the user.

Super classes

sbm::SBM -> sbm::SimpleSBM -> SimpleSBM_fit -> SimpleSBM_fit_noCov -> SimpleSBM_MNAR_noCov

Active bindings

vExpec double: variational approximation of the expectation complete log-likelihood

Methods

Public methods:

- SimpleSBM_MNAR_noCov$new()
- SimpleSBM_MNAR_noCov$update_parameters()
- SimpleSBM_MNAR_noCov$update_blocks()
- SimpleSBM_MNAR_noCov$polish_log_tau()
- SimpleSBM_MNAR_noCov$clone()

`SimpleSBM_MNAR_noCov$new()`: constructor for simpleSBM_fit for missSBM purpose

Usage:

```
SimpleSBM_MNAR_noCov$new(networkData, clusterInit)
```

Arguments:

`networkData` a structure to store network under missing data condition: either a matrix possibly with NA, or a missSBM:::partlyObservedNetwork

`clusterInit` Initial clustering: a vector with size `ncol(adjacencyMatrix)`, providing a user-defined clustering with `nbBlocks` levels.

`SimpleSBM_MNAR_noCov$update_parameters()`: update parameters estimation (M-step)

Usage:

```
SimpleSBM_MNAR_noCov$update_parameters(nu = NULL)
```

Arguments:

`nu` currently imputed values

`SimpleSBM_MNAR_noCov$update_blocks()`: update variational estimation of blocks (VE-step)

Usage:

```
SimpleSBM_MNAR_noCov$update_blocks(log_lambda = 0)
```

Arguments:

`log_lambda` additional term sampling dependent used to de-bias estimation of tau

`SimpleSBM_MNAR_noCov$polish_log_tau()`: for each node, the complete-data log-likelihood it would contribute to each class if hard-assigned there (theta/pi held fixed), used to decide node-swap moves in `missSBM_fit$polish()`. Unlike `update_blocks()`'s tau update (which combines the observed- and imputed-part E-steps as-is), this subtracts one copy of `log(pi)`: both E-step calls add it, so summing them double-counts it relative to `vExpec`'s convention – verified numerically to matter (see git history).

Usage:

`SimpleSBM_MNAR_noCov$polish_log_tau(log_lambda = 0)`

Arguments:

`log_lambda` additional sampling-design-dependent term, added as-is

Returns: an N x Q matrix

`SimpleSBM_MNAR_noCov$clone():` The objects of this class are cloneable with this method.

Usage:

`SimpleSBM_MNAR_noCov$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

`SimpleSBM_fit_noCov` *Internal class for a binary SBM fit under MAR sampling designs without covariates.*

Description

It is not designed to be called directly by the user.

Super classes

`sbm::SBM` -> `sbm::SimpleSBM` -> `SimpleSBM_fit` -> `SimpleSBM_fit_noCov`

Active bindings

`imputation` the matrix of imputed values

`vExpec` double: variational approximation of the expectation complete log-likelihood

`vExpec_corrected` double: variational approximation of the expectation complete log-likelihood with correction to be comparable with MNAR criteria

Methods

Public methods:

- `SimpleSBM_fit_noCov$update_parameters()`
- `SimpleSBM_fit_noCov$update_blocks()`
- `SimpleSBM_fit_noCov$polish_log_tau()`
- `SimpleSBM_fit_noCov$clone()`

`SimpleSBM_fit_noCov$update_parameters():` update parameters estimation (M-step)

Usage:

`SimpleSBM_fit_noCov$update_parameters(...)`

Arguments:

`...` additional arguments, only required for MNAR cases

`SimpleSBM_fit_noCov$update_blocks()`: update variational estimation of blocks (VE-step)

Usage:

```
SimpleSBM_fit_noCov$update_blocks(...)
```

Arguments:

`...` additional arguments, only required for MNAR cases

`SimpleSBM_fit_noCov$polish_log_tau()`: for each node, the complete-data log-likelihood it would contribute to each class if hard-assigned there (theta/pi held fixed), used to decide node-swap moves in `missSBM_fit$polish()`.

Usage:

```
SimpleSBM_fit_noCov$polish_log_tau(log_lambda = 0)
```

Arguments:

`log_lambda` additional sampling-design-dependent term, added as-is

Returns: an N x Q matrix

`SimpleSBM_fit_noCov$clone()`: The objects of this class are cloneable with this method.

Usage:

```
SimpleSBM_fit_noCov$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`SimpleSBM_fit_withCov` *Internal class for a binary SBM fit under MAR sampling designs with covariates.*

Description

It is not designed to be called directly by the user.

Super classes

`sbm::SBM` -> `sbm::SimpleSBM` -> `SimpleSBM_fit` -> `SimpleSBM_fit_withCov`

Active bindings

`imputation` the matrix of imputed values

`vExpec` double: variational approximation of the expectation complete log-likelihood

`vExpec_corrected` double: variational approximation of the expectation complete log-likelihood with correction to be comparable with MNAR criteria

Methods

Public methods:

- SimpleSBM_fit_withCov$update_parameters()
- SimpleSBM_fit_withCov$update_blocks()
- SimpleSBM_fit_withCov$polish_log_tau()
- SimpleSBM_fit_withCov$clone()

`SimpleSBM_fit_withCov$update_parameters()`: update parameters estimation (M-step) via Newton-Raphson: the M-step objective is a weighted logistic regression (concave), so Newton converges in a handful of iterations – no external optimizer is required.

Usage:

```
SimpleSBM_fit_withCov$update_parameters(...)
```

Arguments:

`...` use for compatibility

`SimpleSBM_fit_withCov$update_blocks()`: update variational estimation of blocks (VE-step)

Usage:

```
SimpleSBM_fit_withCov$update_blocks(...)
```

Arguments:

`...` use for compatibility

`SimpleSBM_fit_withCov$polish_log_tau()`: for each node, the complete-data log-likelihood it would contribute to each class if hard-assigned there (theta/beta/pi held fixed), used to decide node-swap moves in `missSBM_fit$polish()`.

Usage:

```
SimpleSBM_fit_withCov$polish_log_tau(log_lambda = 0)
```

Arguments:

`log_lambda` additional sampling-design-dependent term, added as-is

Returns: an N x Q matrix

`SimpleSBM_fit_withCov$clone()`: The objects of this class are cloneable with this method.

Usage:

```
SimpleSBM_fit_withCov$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

snowballSampler *Class for defining a snowball sampler*

Description

Class for defining a snowball sampler

Super classes

networkSampling -> networkSampler -> nodeSampler -> snowballSampler

Methods

Public methods:

- snowballSampler$new()
- snowballSampler$clone()

snowballSampler$new(): constructor for networkSampling

Usage:

```
snowballSampler$new(parameters = NA, adjacencyMatrix = NA, directed = FALSE)
```

Arguments:

parameters the vector of parameters associated to the sampling at play

adjacencyMatrix the adjacency matrix of the network

directed logical, directed network of not

snowballSampler$clone(): The objects of this class are cloneable with this method.

Usage:

```
snowballSampler$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

summary.missSBM_fit *Summary method for a missSBM_fit*

Description

Summary method for a missSBM_fit

Usage

```
## S3 method for class 'missSBM_fit'
summary(object, ...)
```

Arguments

object	an R6 object with class missSBM_fit
...	additional parameters for S3 compatibility.

Value

a basic printing output

---

war *War data set*

Description

This dataset contains two networks where the nodes are countries and an edge in network "belligerent" means that the two countries have been at least once at war between years 1816 to 2007 while an edge in network "alliance" means that the two countries have had a formal alliance between years 1816 to 2012. The network belligerent have less nodes since countries which have not been at war are not considered.

Usage

```
war
```

Format

A list with 2 two igraph objects, alliance and belligerent. Each graph have three attributes: 'name' (the country name), 'power' (a score related to military power: the higher, the better) and 'trade' (a score related to the trade effort between pairs of countries).

Source

networks were extracted from https://correlatesofwar.org/

References

Sarkees, Meredith Reid and Frank Wayman (2010). Resort to War: 1816 - 2007. Washington DC: CQ Press.

Gibler, Douglas M. 2009. International military alliances, 1648-2008. CQ Press

Examples

```
data(war)
class(war$belligerent)
igraph::gorder(war$alliance)
igraph::gorder(war$belligerent)
igraph::edges(war$alliance)
igraph::get.graph.attribute(war$alliance)
```

Index

∗ **datasets**
    er_network, 15
    frenchblog2007, 18
    war, 49

blockDyadSampler, 3
blockDyadSampling_fit, 4
blockNodeSampler, 5
blockNodeSampling_fit, 6

coef.missSBM_fit, 7
covarDyadSampling_fit, 7
covarNodeSampling_fit, 8

degreeSampler, 9
degreeSampling_fit, 10
doubleStandardSampler, 11
doubleStandardSampling_fit, 12
dyadSampler, *3*, *11*, 13, *40*
dyadSampling_fit, 14

er_network, 15
estimateMissSBM, 15
estimateMissSBM(), *7*, *17*, *19–22*, *24–26*, *28*

fitted(), *22*
fitted.missSBM_fit, 17
frenchblog2007, 18

l1_similarity, 19, *27*, *35*

missSBM_collection, *16*, 19, *19*, *28*
missSBM_fit, *7*, *16–21*, 22, *22*, *24, 25*, *27*, *38, 39*, *48, 49*
missSBM_param, *16*, 26

networkSampler, *3*, *5*, *9*, *11*, *13*, 28, *33*, *40, 41*, *48*
networkSampling, *3–14*, *22*, *28*, 29, *30*, *32–34*, *40, 41*, *48*
networkSamplingDyads_fit, *4*, *7*, *12*, *14*, 30
networkSamplingNodes_fit, *6*, *8*, *10*, 32, *34*
nodeSampler, *5*, *9*, 33, *41*, *48*
nodeSampling_fit, 34

observeNetwork, *16*, 35

partlyObservedNetwork, *20*, *23*, *27*, *29*, 37
plot(), *22*
plot.missSBM_fit, 38
predict(), *22*
predict.missSBM_fit *(*predicted.missSBM_fit*)*, 39
predicted.missSBM_fit, 39
print(), *19*, *22*

sbm::SBM, *42*, *44–46*
sbm::SimpleSBM, *42*, *44–46*
sbm::SimpleSBM_fit, *22*
show(), *19*, *22*
simpleDyadSampler, 40
simpleNodeSampler, 41
SimpleSBM_fit, 42, *44–46*
SimpleSBM_fit_MNAR, *22*, *42*, 43
SimpleSBM_fit_noCov, *22*, *42*, *44*, 45
SimpleSBM_fit_withCov, *22*, *42*, 46
snowballSampler, 48
summary.missSBM_fit, 48

war, 49