Package ‘fastRG’

June 30, 2022

Title Sample Generalized Random Dot Product Graphs in Linear Time

Version 0.3.1

Description Samples generalized random product graph, a generalization of a broad class of network models. Given matrices X, S, and Y with with non-negative entries, samples a matrix with expectation X S Y^T and independent Poisson or Bernoulli entries using the fastRG algorithm of Rohe et al. (2017) <https://www.jmlr.org/papers/v19/17-128.html>. The algorithm first samples the number of edges and then puts them down one-by-one. As a result it is O(m) where m is the number of edges, a dramatic improvement over element-wise algorithms that which require O(n^2) operations to sample a random graph, where n is the number of nodes.

URL https://rohelab.github.io/fastRG/,
https://github.com/RoheLab/fastRG

BugReports https://github.com/RoheLab/fastRG/issues

Depends Matrix

Imports ellipsis, glue, igraph, RSpectra, stats, tibble, tidygraph

Suggests covr, dplyr, ggplot2, knitr, magrittr, rmarkdown, testthat (>= 3.0.0)

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.2.0.9000

NeedsCompilation no

Author Alex Hayes [aut, cre, cph] (<https://orcid.org/0000-0002-4985-5160>),
Karl Rohe [aut, cph],
Jun Tao [aut],
Xintian Han [aut],
Norbert Binkiewicz [aut]

Maintainer Alex Hayes <alexpghayes@gmail.com>

Repository CRAN

Date/Publication 2022-06-30 07:30:12 UTC

R topics documented:

---

chung_lu *Create an undirected Chung-Lu object*

Description

To specify a Chung-Lu graph, you must specify the degree-heterogeneity parameters (via `n` or `theta`). We provide reasonable defaults to enable rapid exploration or you can invest the effort for more control over the model parameters. We **strongly recommend** setting the `expected_degree` or `expected_density` argument to avoid large memory allocations associated with sampling large, dense graphs.

Usage

```
chung_lu(
  n = NULL,
  theta = NULL,
  ...,
  sort_nodes = TRUE,
  poisson_edges = TRUE,
  allow_self_loops = TRUE,
  force_identifiability = FALSE
)
```

Arguments

`n` (degree heterogeneity) The number of nodes in the graph. Use when you don't want to specify the degree-heterogeneity parameters `theta` by hand. When `n` is specified, `theta` is randomly generated from a `LogNormal(2, 1)` distribution. This is subject to change, and may not be reproducible. `n` defaults to `NULL`. You must specify either `n` or `theta`, but not both.

`theta` (degree heterogeneity) A numeric vector explicitly specifying the degree heterogeneity parameters. This implicitly determines the number of nodes in the resulting graph, i.e. it will have `length(theta)` nodes. Must be positive. Setting to a vector of ones recovers an erdos renyi graph. Defaults to `NULL`. You must specify either `n` or `theta`, but not both.

`...` Arguments passed on to `undirected_factor_model`

- `expected_degree` If specified, the desired expected degree of the graph. Specifying `expected_degree` simply rescales `S` to achieve this. Defaults to `NULL`. Do not specify both `expected_degree` and `expected_density` at the same time.
- `expected_density` If specified, the desired expected density of the graph. Specifying `expected_density` simply rescales `S` to achieve this. Defaults to `NULL`. Do not specify both `expected_degree` and `expected_density` at the same time.

`sort_nodes` Logical indicating whether or not to sort the nodes so that they are grouped by block and by `theta`. Useful for plotting. Defaults to `TRUE`.

`poisson_edges` Logical indicating whether or not multiple edges are allowed to form between a pair of nodes. Defaults to `TRUE`. When `FALSE`, sampling proceeds as usual, and duplicate edges are removed afterwards. Further, when `FALSE`, we assume that `S` specifies a desired between-factor connection probability, and back-transform this `S` to the appropriate Poisson intensity parameter to approximate Bernoulli factor connection probabilities. See Section 2.3 of Rohe et al. (2017) for some additional details.

`allow_self_loops` Logical indicating whether or not nodes should be allowed to form edges with themselves. Defaults to `TRUE`. When `FALSE`, sampling proceeds allowing self-loops, and these are then removed after the fact.

`force_identifiability` Logical indicating whether or not to normalize `theta` such that it sums to one within each block. Defaults to `FALSE`, since this behavior can be surprise when `theta` is set to a vector of all ones to recover the DC-SBM case.

Value

An `undirected_chung_lu` S3 object, a subclass of `dcsbm()`.

See Also

Other undirected graphs: `dcsbm()`, `erdos_renyi()`, `mmsbm()`, `overlapping_sbm()`, `planted_partition()`, `sbm()`

Examples

```
set.seed(27)

cl <- chung_lu(n = 1000, expected_density = 0.01)
cl

theta <- round(stats::rlnorm(100, 2))

cl2 <- chung_lu(
  theta = theta,
  expected_degree = 5
)

cl2

edgelist <- sample_edgelist(cl)
edgelist
```

---

dcsbm *Create an undirected degree corrected stochastic blockmodel object*

Description

To specify a degree-corrected stochastic blockmodel, you must specify the degree-heterogeneity parameters (via `n` or `theta`), the mixing matrix (via `k` or `B`), and the relative block probabilities (optional, via `pi`). We provide defaults for most of these options to enable rapid exploration, or you can invest the effort for more control over the model parameters. We **strongly recommend** setting the `expected_degree` or `expected_density` argument to avoid large memory allocations associated with sampling large, dense graphs.

Usage

```
dcsbm(
  n = NULL,
  theta = NULL,
  k = NULL,
  B = NULL,
  ...,
  pi = rep(1/k, k),
  sort_nodes = TRUE,
  force_identifiability = FALSE,
  poisson_edges = TRUE,
  allow_self_loops = TRUE
)
```

Arguments

- `n` (degree heterogeneity) The number of nodes in the blockmodel. Use when you don't want to specify the degree-heterogeneity parameters `theta` by hand. When `n` is specified, `theta` is randomly generated from a `LogNormal(2, 1)` distribution. This is subject to change, and may not be reproducible. `n` defaults to `NULL`. You must specify either `n` or `theta`, but not both.
- `theta` (degree heterogeneity) A numeric vector explicitly specifying the degree heterogeneity parameters. This implicitly determines the number of nodes in the resulting graph, i.e. it will have `length(theta)` nodes. Must be positive. Setting to a vector of ones recovers a stochastic blockmodel without degree correction. Defaults to `NULL`. You must specify either `n` or `theta`, but not both.
- `k` (mixing matrix) The number of blocks in the blockmodel. Use when you don't want to specify the mixing-matrix by hand. When `k` is specified, the elements of `B` are drawn randomly from a `Uniform(0, 1)` distribution. This is subject to change, and may not be reproducible. `k` defaults to `NULL`. You must specify either `k` or `B`, but not both.
- `B` (mixing matrix) A `k` by `k` matrix of block connection probabilities. The probability that a node in block `i` connects to a node in community `j` is `Poisson(B[i, j])`. Must be a square matrix. `matrix` and `Matrix` objects are both acceptable. If `B` is not symmetric, it will be symmetrized via the update `B := B + t(B)`. Defaults to `NULL`. You must specify either `k` or `B`, but not both.
- `...` Arguments passed on to `undirected_factor_model`
  - `expected_degree` If specified, the desired expected degree of the graph. Specifying `expected_degree` simply rescales `S` to achieve this. Defaults to `NULL`. Do not specify both `expected_degree` and `expected_density` at the same time.
  - `expected_density` If specified, the desired expected density of the graph. Specifying `expected_density` simply rescales `S` to achieve this. Defaults to `NULL`. Do not specify both `expected_degree` and `expected_density` at the same time.
- `pi` (relative block probabilities) Relative block probabilities. Must be positive, but do not need to sum to one, as they will be normalized internally. Must match the dimensions of `B` or `k`. Defaults to `rep(1 / k, k)`, or a balanced blocks.
- `sort_nodes` Logical indicating whether or not to sort the nodes so that they are grouped by block and by `theta`. Useful for plotting. Defaults to `TRUE`.
- `force_identifiability`
  Logical indicating whether or not to normalize `theta` such that it sums to one within each block. Defaults to `FALSE`, since this behavior can be surprise when `theta` is set to a vector of all ones to recover the DC-SBM case.
- `poisson_edges` Logical indicating whether or not multiple edges are allowed to form between a pair of nodes. Defaults to `TRUE`. When `FALSE`, sampling proceeds as usual, and duplicate edges are removed afterwards. Further, when `FALSE`, we assume that `S` specifies a desired between-factor connection probability, and back-transform this `S` to the appropriate Poisson intensity parameter to approximate Bernoulli factor connection probabilities. See Section 2.3 of Rohe et al. (2017) for some additional details.

`allow_self_loops`
: Logical indicating whether or not nodes should be allowed to form edges with themselves. Defaults to `TRUE`. When `FALSE`, sampling proceeds allowing self-loops, and these are then removed after the fact.

Value

An `undirected_dcsbm` S3 object, a subclass of the `undirected_factor_model()` with the following additional fields:

- `theta`: A numeric vector of degree-heterogeneity parameters.
- `z`: The community memberships of each node, as a `factor()`. The factor will have `k` levels, where `k` is the number of communities in the stochastic blockmodel. There will not always necessarily be observed nodes in each community.
- `pi`: Sampling probabilities for each block.
- `sorted`: Logical indicating where nodes are arranged by block (and additionally by degree heterogeneity parameter) within each block.

Generative Model

There are two levels of randomness in a degree-corrected stochastic blockmodel. First, we randomly chose a block membership for each node in the blockmodel. This is handled by `dcsbm()`. Then, given these block memberships, we randomly sample edges between nodes. This second operation is handled by `sample_edgelist()`, `sample_sparse()`, `sample_igraph()` and `sample_tidygraph()`, depending depending on your desired graph representation.

Block memberships:

Let z_i represent the block membership of node i. To generate z_i we sample from a categorical distribution (note that this is a special case of a multinomial) with parameter π, such that π_i represents the probability of ending up in the ith block. Block memberships for each node are independent.

Degree heterogeneity:

In addition to block membership, the DCSBM also allows nodes to have different propensities for edge formation. We represent this propensity for node i by a positive number θ_i. Typically the θ_i are constrained to sum to one for identifiability purposes, but this doesn't really matter during sampling (i.e. without the sum constraint scaling B and θ has the same effect on edge probabilities, but whether B or θ is responsible for this change is uncertain).

Edge formulation:

Once we know the block memberships z and the degree heterogeneity parameters $theta$, we need one more ingredient, which is the baseline intensity of connections between nodes in block `i` and block `j`. Then each edge $A_{i,j}$ is Poisson distributed with parameter

$$\lambda[i, j] = \theta_i \cdot B_{z_i, z_j} \cdot \theta_j .$$

See Also

Other stochastic block models: directed_dcsbm(), mmsbm(), overlapping_sbm(), planted_partition(), sbm()

Other undirected graphs: chung_lu(), erdos_renyi(), mmsbm(), overlapping_sbm(), planted_partition(), sbm()

Examples

```
set.seed(27)

lazy_dcsbm <- dcsbm(n = 1000, k = 5, expected_density = 0.01)
lazy_dcsbm

# sometimes you gotta let the world burn and
# sample a wildly dense graph

dense_lazy_dcsbm <- dcsbm(n = 500, k = 3, expected_density = 0.8)
dense_lazy_dcsbm

# explicitly setting the degree heterogeneity parameter,
# mixing matrix, and relative community sizes rather
# than using randomly generated defaults

k <- 5
n <- 1000
B <- matrix(stats::runif(k * k), nrow = k, ncol = k)

theta <- round(stats::rlnorm(n, 2))

pi <- c(1, 2, 4, 1, 1)

custom_dcsbm <- dcsbm(
  theta = theta,
  B = B,
  pi = pi,
  expected_degree = 50
)

custom_dcsbm

edgelist <- sample_edgelist(custom_dcsbm)
edgelist

# efficient eigendecompostion that leverages low-rank structure in
# E(A) so that you don't have to form E(A) to find eigenvectors,
# as E(A) is typically dense. computation is
# handled via RSpectra

population_eigs <- eigs_sym(custom_dcsbm)
```

directed_dcsbm *Create a directed degree corrected stochastic blockmodel object*

Description

To specify a degree-corrected stochastic blockmodel, you must specify the degree-heterogeneity parameters (via `n_in` or `theta_in`, and `n_out` or `theta_out`), the mixing matrix (via `k_in` and `k_out`, or `B`), and the relative block probabilities (optional, via `p_in` and `pi_out`). We provide defaults for most of these options to enable rapid exploration, or you can invest the effort for more control over the model parameters. We **strongly recommend** setting the `expected_in_degree`, `expected_out_degree`, or `expected_density` argument to avoid large memory allocations associated with sampling large, dense graphs.

Usage

```
directed_dcsbm(
  n = NULL,
  theta_in = NULL,
  theta_out = NULL,
  k_in = NULL,
  k_out = NULL,
  B = NULL,
  ...,
  pi_in = rep(1/k_in, k_in),
  pi_out = rep(1/k_out, k_out),
  sort_nodes = TRUE,
  force_identifiability = TRUE,
  poisson_edges = TRUE,
  allow_self_loops = TRUE
)
```

Arguments

`n` (degree heterogeneity) The number of nodes in the blockmodel. Use when you don't want to specify the degree-heterogeneity parameters `theta_in` and `theta_out` by hand. When `n` is specified, `theta_in` and `theta_out` are randomly generated from a `LogNormal(2, 1)` distribution. This is subject to change, and may not be reproducible. `n` defaults to `NULL`. You must specify either `n` or `theta_in` and `theta_out` together, but not both.

`theta_in` (degree heterogeneity) A numeric vector explicitly specifying the degree heterogeneity parameters. This implicitly determines the number of nodes in the resulting graph, i.e. it will have `length(theta_in)` nodes. Must be positive. Setting to a vector of ones recovers a stochastic blockmodel without degree correction. Defaults to `NULL`. You must specify either `n` or `theta_in` and `theta_out` together, but not both.

theta_out
: (degree heterogeneity) A numeric vector explicitly specifying the degree heterogeneity parameters. This implicitly determines the number of nodes in the resulting graph, i.e. it will have `length(theta)` nodes. Must be positive. Setting to a vector of ones recovers a stochastic blockmodel without degree correction. Defaults to `NULL`. You must specify either `n` or `theta_in` and `theta_out` together, but not both.

k_in
: (mixing matrix) The number of blocks in the blockmodel. Use when you don't want to specify the mixing-matrix by hand. When `k_in` is specified, the elements of `B` are drawn randomly from a `Uniform(0, 1)` distribution. This is subject to change, and may not be reproducible. `k_in` defaults to `NULL`. You must specify either `k_in` and `k_out` together, or `B`. You may specify all three at once, in which case `k_in` is only used to set `pi_in` (when `pi_in` is left at its default argument value).

k_out
: (mixing matrix) The number of blocks in the blockmodel. Use when you don't want to specify the mixing-matrix by hand. When `k_out` is specified, the elements of `B` are drawn randomly from a `Uniform(0, 1)` distribution. This is subject to change, and may not be reproducible. `k_out` defaults to `NULL`. You may specify all three at once, in which case `k_out` is only used to set `pi_out` (when `pi_out` is left at its default argument value).

B
: (mixing matrix) A `k_in` by `k_out` matrix of block connection probabilities. The probability that a node in block `i` connects to a node in community `j` is `Poisson(B[i, j])`. `matrix` and `Matrix` objects are both acceptable. Defaults to `NULL`. You must specify either `k_in` and `k_out` together, or `B`, but not both.

...
: Arguments passed on to `directed_factor_model`

  `expected_in_degree` If specified, the desired expected in degree of the graph. Specifying `expected_in_degree` simply rescales `S` to achieve this. Defaults to `NULL`. Specify only one of `expected_in_degree`, `expected_out_degree`, and `expected_density`.

  `expected_out_degree` If specified, the desired expected out degree of the graph. Specifying `expected_out_degree` simply rescales `S` to achieve this. Defaults to `NULL`. Specify only one of `expected_in_degree`, `expected_out_degree`, and `expected_density`.

  `expected_density` If specified, the desired expected density of the graph. Specifying `expected_density` simply rescales `S` to achieve this. Defaults to `NULL`. Specify only one of `expected_in_degree`, `expected_out_degree`, and `expected_density`.

pi_in
: (relative block probabilities) Relative block probabilities. Must be positive, but do not need to sum to one, as they will be normalized internally. Must match the rows of `B`, or `k_in`. Defaults to `rep(1 / k_in, k_in)`, or a balanced incoming blocks.

pi_out
: (relative block probabilities) Relative block probabilities. Must be positive, but do not need to sum to one, as they will be normalized internally. Must match the columns of `B`, or `k_out`. Defaults to `rep(1 / k_out, k_out)`, or a balanced outgoing blocks.

sort_nodes
: Logical indicating whether or not to sort the nodes so that they are grouped by block. Useful for plotting. Defaults to `TRUE`.

force_identifiability
: Logical indicating whether or not to normalize theta_in such that it sums to one within each incoming block and theta_out such that it sums to one within each outgoing block. Defaults to TRUE.

poisson_edges
: Logical indicating whether or not multiple edges are allowed to form between a pair of nodes. Defaults to TRUE. When FALSE, sampling proceeds as usual, and duplicate edges are removed afterwards. Further, when FALSE, we assume that S specifies a desired between-factor connection probability, and back-transform this S to the appropriate Poisson intensity parameter to approximate Bernoulli factor connection probabilities. See Section 2.3 of Rohe et al. (2017) for some additional details.

allow_self_loops
: Logical indicating whether or not nodes should be allowed to form edges with themselves. Defaults to TRUE. When FALSE, sampling proceeds allowing self-loops, and these are then removed after the fact.

Value

A directed_dcsbm S3 object, a subclass of the directed_factor_model() with the following additional fields:

- theta_in: A numeric vector of incoming community degree-heterogeneity parameters.
- theta_out: A numeric vector of outgoing community degree-heterogeneity parameters.
- z_in: The incoming community memberships of each node, as a factor(). The factor will have k_in levels, where k_in is the number of incoming communities in the stochastic blockmodel. There will not always necessarily be observed nodes in each community.
- z_out: The outgoing community memberships of each node, as a factor(). The factor will have k_out levels, where k_out is the number of outgoing communities in the stochastic blockmodel. There will not always necessarily be observed nodes in each community.
- pi_in: Sampling probabilities for each incoming community.
- pi_out: Sampling probabilities for each outgoing community.
- sorted: Logical indicating where nodes are arranged by block (and additionally by degree heterogeneity parameter) within each block.

Generative Model

There are two levels of randomness in a directed degree-corrected stochastic blockmodel. First, we randomly chose a incoming block membership and an outgoing block membership for each node in the blockmodel. This is handled by directed_dcsbm(). Then, given these block memberships, we randomly sample edges between nodes. This second operation is handled by sample_edgelist(), sample_sparse(), sample_igraph() and sample_tidygraph(), depending on your desired graph representation.

Block memberships:

Let x represent the incoming block membership of a node and y represent the outgoing block membership of a node. To generate x we sample from a categorical distribution with parameter $\pi_i n$. To generate y we sample from a categorical distribution with parameter $\pi_o ut$. Block memberships are independent across nodes. Incoming and outgoing block memberships of the same node are also independent.

Degree heterogeneity:

In addition to block membership, the DCSBM also nodes to have different propensities for incoming and outgoing edge formation. We represent the propensity to form incoming edges for a given node by a positive number θ_in. We represent the propensity to form outgoing edges for a given node by a positive number θ_out. Typically the θ_in (and $theta_out$) across all nodes are constrained to sum to one for identifiability purposes, but this doesn't really matter during sampling.

Edge formulation:

Once we know the block memberships x and y and the degree heterogeneity parameters θ_{in} and θ_{out}, we need one more ingredient, which is the baseline intensity of connections between nodes in block `i` and block `j`. Then each edge forms independently according to a Poisson distribution with parameters

$$\lambda = \theta_{in} * B_{x,y} * \theta_{out}.$$

See Also

Other stochastic block models: `dcsbm()`, `mmsbm()`, `overlapping_sbm()`, `planted_partition()`, `sbm()`

Other directed graphs: `directed_erdos_renyi()`

Examples

```
set.seed(27)

B <- matrix(0.2, nrow = 5, ncol = 8)
diag(B) <- 0.9

ddcsbm <- directed_dcsbm(
  n = 1000,
  B = B,
  k_in = 5,
  k_out = 8,
  expected_density = 0.01
)

ddcsbm

population_svd <- svds(ddcsbm)
```

`directed_erdos_renyi` *Create an directed erdos renyi object*

Description

Create an directed erdos renyi object

Usage

```
directed_erdos_renyi(
  n,
  ...,
  p = NULL,
  poisson_edges = TRUE,
  allow_self_loops = TRUE
)
```

Arguments

n — Number of nodes in graph.

... — Arguments passed on to directed_factor_model

- expected_in_degree If specified, the desired expected in degree of the graph. Specifying expected_in_degree simply rescales S to achieve this. Defaults to NULL. Specify only one of expected_in_degree, expected_out_degree, and expected_density.
- expected_out_degree If specified, the desired expected out degree of the graph. Specifying expected_out_degree simply rescales S to achieve this. Defaults to NULL. Specify only one of expected_in_degree, expected_out_degree, and expected_density.

p — Probability of an edge between any two nodes. You must specify either p, expected_in_degree, or expected_out_degree.

poisson_edges — Logical indicating whether or not multiple edges are allowed to form between a pair of nodes. Defaults to TRUE. When FALSE, sampling proceeds as usual, and duplicate edges are removed afterwards. Further, when FALSE, we assume that S specifies a desired between-factor connection probability, and back-transform this S to the appropriate Poisson intensity parameter to approximate Bernoulli factor connection probabilities. See Section 2.3 of Rohe et al. (2017) for some additional details.

allow_self_loops — Logical indicating whether or not nodes should be allowed to form edges with themselves. Defaults to TRUE. When FALSE, sampling proceeds allowing self-loops, and these are then removed after the fact.

Value

A directed_factor_model S3 class based on a list with the following elements:

- X: The incoming latent positions as a Matrix() object.
- S: The mixing matrix as a Matrix() object.
- Y: The outgoing latent positions as a Matrix() object.
- n: The number of nodes with incoming edges in the network.
- k1: The dimension of the latent node position vectors encoding incoming latent communities (i.e. in X).

- d: The number of nodes with outgoing edges in the network. Does not need to match n – rectangular adjacency matrices are supported.
- k2: The dimension of the latent node position vectors encoding outgoing latent communities (i.e. in Y).
- poisson_edges: Whether or not the graph is taken to be have Poisson or Bernoulli edges, as indicated by a logical vector of length 1.
- allow_self_loops: Whether or not self loops are allowed.

See Also

Other erdos renyi: erdos_renyi()

Other directed graphs: directed_dcsbm()

Examples

```
set.seed(87)

er <- directed_erdos_renyi(n = 10, p = 0.1)
er

big_er <- directed_erdos_renyi(n = 10^6, expected_in_degree = 5)
big_er

A <- sample_sparse(er)
A
```

---

directed_factor_model *Create a directed factor model graph*

Description

A directed factor model graph is a directed generalized Poisson random dot product graph. The edges in this graph are assumpted to be independent and Poisson distributed. The graph is parameterized by its expected adjacency matrix, with is E[A] = X S Y'. We do not recommend that causal users use this function, see instead directed_dcsbm() and related functions, which will formulate common variants of the stochastic blockmodels as undirected factor models *with lots of helpful input validation*.

Usage

```
directed_factor_model(
  X,
  S,
  Y,
  ...,
```

```
  expected_in_degree = NULL,
  expected_out_degree = NULL,
  expected_density = NULL,
  poisson_edges = TRUE,
  allow_self_loops = TRUE
)
```

Arguments

X
: A `matrix()` or Matrix() representing real-valued latent node positions encoding community structure of incoming edges. Entries must be positive.

S
: A `matrix()` or Matrix() mixing matrix. Entries must be positive.

Y
: A `matrix()` or Matrix() representing real-valued latent node positions encoding community structure of outgoing edges. Entries must be positive.

...
: Ignored. For internal developer use only.

expected_in_degree
: If specified, the desired expected in degree of the graph. Specifying `expected_in_degree` simply rescales `S` to achieve this. Defaults to `NULL`. Specify only one of `expected_in_degree`, `expected_out_degree`, and `expected_density`.

expected_out_degree
: If specified, the desired expected out degree of the graph. Specifying `expected_out_degree` simply rescales `S` to achieve this. Defaults to `NULL`. Specify only one of `expected_in_degree`, `expected_out_degree`, and `expected_density`.

expected_density
: If specified, the desired expected density of the graph. Specifying `expected_density` simply rescales `S` to achieve this. Defaults to `NULL`. Specify only one of `expected_in_degree`, `expected_out_degree`, and `expected_density`.

poisson_edges
: Logical indicating whether or not multiple edges are allowed to form between a pair of nodes. Defaults to `TRUE`. When `FALSE`, sampling proceeds as usual, and duplicate edges are removed afterwards. Further, when `FALSE`, we assume that `S` specifies a desired between-factor connection probability, and back-transform this `S` to the appropriate Poisson intensity parameter to approximate Bernoulli factor connection probabilities. See Section 2.3 of Rohe et al. (2017) for some additional details.

allow_self_loops
: Logical indicating whether or not nodes should be allowed to form edges with themselves. Defaults to `TRUE`. When `FALSE`, sampling proceeds allowing self-loops, and these are then removed after the fact.

Value

A `directed_factor_model` S3 class based on a list with the following elements:

- `X`: The incoming latent positions as a `Matrix()` object.
- `S`: The mixing matrix as a `Matrix()` object.
- `Y`: The outgoing latent positions as a `Matrix()` object.
- `n`: The number of nodes with incoming edges in the network.

- k1: The dimension of the latent node position vectors encoding incoming latent communities (i.e. in X).
- d: The number of nodes with outgoing edges in the network. Does not need to match n – rectangular adjacency matrices are supported.
- k2: The dimension of the latent node position vectors encoding outgoing latent communities (i.e. in Y).
- poisson_edges: Whether or not the graph is taken to be have Poisson or Bernoulli edges, as indicated by a logical vector of length 1.
- allow_self_loops: Whether or not self loops are allowed.

Examples

```
n <- 10000

k1 <- 5
k2 <- 3

d <- 5000

X <- matrix(rpois(n = n * k1, 1), nrow = n)
S <- matrix(runif(n = k1 * k2, 0, .1), nrow = k1, ncol = k2)
Y <- matrix(rexp(n = k2 * d, 1), nrow = d)

fm <- directed_factor_model(X, S, Y)
fm

fm2 <- directed_factor_model(X, S, Y, expected_in_degree = 50)
fm2
```

eigs_sym.undirected_factor_model

Compute the eigendecomposition of the expected adjacency matrix of an undirected factor model

Description

Compute the eigendecomposition of the expected adjacency matrix of an undirected factor model

Usage

```
## S3 method for class 'undirected_factor_model'
eigs_sym(A, k = A$k, which = "LM", sigma = NULL, opts = list(), ...)
```

Arguments

`A`	An `undirected_factor_model()`.
`k`	Desired rank of decomposition.
`which`	Selection criterion. See **Details** below.
`sigma`	Shift parameter. See section **Shift-And-Invert Mode**.
`opts`	Control parameters related to the computing algorithm. See **Details** below.
`...`	Unused, included only for consistency with generic signature.

Details

The `which` argument is a character string that specifies the type of eigenvalues to be computed. Possible values are:

"LM"	The k eigenvalues with largest magnitude. Here the magnitude means the Euclidean norm of complex numbers.
"SM"	The k eigenvalues with smallest magnitude.
"LR"	The k eigenvalues with largest real part.
"SR"	The k eigenvalues with smallest real part.
"LI"	The k eigenvalues with largest imaginary part.
"SI"	The k eigenvalues with smallest imaginary part.
"LA"	The k largest (algebraic) eigenvalues, considering any negative sign.
"SA"	The k smallest (algebraic) eigenvalues, considering any negative sign.
"BE"	Compute k eigenvalues, half from each end of the spectrum. When k is odd, compute more from the high and then fro

`eigs()` with matrix types "matrix", "dgeMatrix", "dgCMatrix" and "dgRMatrix" can use "LM", "SM", "LR", "SR", "LI" and "SI".

`eigs_sym()` with all supported matrix types, and `eigs()` with symmetric matrix types ("dsyMatrix", "dsCMatrix", and "dsRMatrix") can use "LM", "SM", "LA", "SA" and "BE".

The `opts` argument is a list that can supply any of the following parameters:

`ncv` Number of Lanzcos basis vectors to use. More vectors will result in faster convergence, but with greater memory use. For general matrix, `ncv` must satisfy $k + 2 \leq ncv \leq n$, and for symmetric matrix, the constraint is $k < ncv \leq n$. Default is `min(n, max(2*k+1, 20))`.

`tol` Precision parameter. Default is 1e-10.

`maxitr` Maximum number of iterations. Default is 1000.

`retvec` Whether to compute eigenvectors. If FALSE, only calculate and return eigenvalues.

`initvec` Initial vector of length n supplied to the Arnoldi/Lanczos iteration. It may speed up the convergence if `initvec` is close to an eigenvector of A.

`erdos_renyi` *Create an undirected erdos renyi object*

Description

Create an undirected erdos renyi object

Usage

```
erdos_renyi(n, ..., p = NULL, poisson_edges = TRUE, allow_self_loops = TRUE)
```

Arguments

n Number of nodes in graph.

... Arguments passed on to undirected_factor_model

expected_degree If specified, the desired expected degree of the graph. Specifying expected_degree simply rescales S to achieve this. Defaults to NULL. Do not specify both expected_degree and expected_density at the same time.

p Probability of an edge between any two nodes. You must specify either p or expected_degree.

poisson_edges Logical indicating whether or not multiple edges are allowed to form between a pair of nodes. Defaults to TRUE. When FALSE, sampling proceeds as usual, and duplicate edges are removed afterwards. Further, when FALSE, we assume that S specifies a desired between-factor connection probability, and back-transform this S to the appropriate Poisson intensity parameter to approximate Bernoulli factor connection probabilities. See Section 2.3 of Rohe et al. (2017) for some additional details.

allow_self_loops
Logical indicating whether or not nodes should be allowed to form edges with themselves. Defaults to TRUE. When FALSE, sampling proceeds allowing self-loops, and these are then removed after the fact.

Value

An undirected_factor_model S3 class based on a list with the following elements:

- X: The latent positions as a Matrix() object.
- S: The mixing matrix as a Matrix() object.
- n: The number of nodes in the network.
- k: The rank of expectation matrix. Equivalently, the dimension of the latent node position vectors.

See Also

Other erdos renyi: directed_erdos_renyi()

Other undirected graphs: chung_lu(), dcsbm(), mmsbm(), overlapping_sbm(), planted_partition(), sbm()

Examples

```
set.seed(87)

er <- erdos_renyi(n = 10, p = 0.1)
```

```
er


er <- erdos_renyi(n = 10, expected_density = 0.1)
er

big_er <- erdos_renyi(n = 10^6, expected_degree = 5)
big_er

A <- sample_sparse(er)
A
```

`expected_edges` *Calculate the expected edges in Poisson RDPG graph*

Description

These calculations are conditional on the latent factors `X` and `Y`.

Usage

```
expected_edges(factor_model, ...)

expected_degree(factor_model, ...)

expected_in_degree(factor_model, ...)

expected_out_degree(factor_model, ...)

expected_density(factor_model, ...)

expected_degrees(factor_model, ...)
```

Arguments

`factor_model`	A `directed_factor_model()` or `undirected_factor_model()`.
`...`	Ignored. Do not use.

Details

Note that the runtime of the `fastRG` algorithm is proportional to the expected number of edges in the graph. Expected edge count will be an underestimate of expected number of edges for Bernoulli graphs. See the Rohe et al for details.

Value

Expected edge counts, or graph densities.

References

Rohe, Karl, Jun Tao, Xintian Han, and Norbert Binkiewicz. 2017. "A Note on Quickly Sampling a Sparse Matrix with Low Rank Expectation." Journal of Machine Learning Research; 19(77):1-13, 2018. https://www.jmlr.org/papers/v19/17-128.html

Examples

```
##### an undirected blockmodel example

n <- 1000
pop <- n / 2
a <- .1
b <- .05

B <- matrix(c(a,b,b,a), nrow = 2)

b_model <- fastRG::sbm(n = n, k = 2, B = B, poisson_edges = FALSE)

b_model

A <- sample_sparse(b_model)

# compare
mean(rowSums(triu(A)))

pop * a + pop * b  # analytical average degree

##### more generic examples

n <- 10000
k <- 5

X <- matrix(rpois(n = n * k, 1), nrow = n)
S <- matrix(runif(n = k * k, 0, .1), nrow = k)

ufm <- undirected_factor_model(X, S)

expected_edges(ufm)
expected_degree(ufm)
eigs_sym(ufm)

n <- 10000
d <- 1000

k1 <- 5
k2 <- 3

X <- matrix(rpois(n = n * k1, 1), nrow = n)
Y <- matrix(rpois(n = d * k2, 1), nrow = d)
S <- matrix(runif(n = k1 * k2, 0, .1), nrow = k1)
```

```
dfm <- directed_factor_model(X = X, S = S, Y = Y)

expected_edges(dfm)
expected_in_degree(dfm)
expected_out_degree(dfm)

svds(dfm)
```

---

mmsbm *Create an undirected degree-corrected mixed membership stochastic blockmodel object*

Description

To specify a degree-corrected mixed membership stochastic blockmodel, you must specify the degree-heterogeneity parameters (via `n` or `theta`), the mixing matrix (via `k` or `B`), and the relative block propensities (optional, via `alpha`). We provide defaults for most of these options to enable rapid exploration, or you can invest the effort for more control over the model parameters. We **strongly recommend** setting the `expected_degree` or `expected_density` argument to avoid large memory allocations associated with sampling large, dense graphs.

Usage

```
mmsbm(
  n = NULL,
  theta = NULL,
  k = NULL,
  B = NULL,
  ...,
  alpha = rep(1, k),
  sort_nodes = TRUE,
  force_pure = TRUE,
  poisson_edges = TRUE,
  allow_self_loops = TRUE
)
```

Arguments

`n`	(degree heterogeneity) The number of nodes in the blockmodel. Use when you don't want to specify the degree-heterogeneity parameters `theta` by hand. When `n` is specified, `theta` is randomly generated from a `LogNormal(2, 1)` distribution. This is subject to change, and may not be reproducible. `n` defaults to `NULL`. You must specify either `n` or `theta`, but not both.
`theta`	(degree heterogeneity) A numeric vector explicitly specifying the degree heterogeneity parameters. This implicitly determines the number of nodes in the resulting graph, i.e. it will have `length(theta)` nodes. Must be positive. Setting to a vector of ones recovers a stochastic blockmodel without degree correction. Defaults to `NULL`. You must specify either `n` or `theta`, but not both.

k (mixing matrix) The number of blocks in the blockmodel. Use when you don't want to specify the mixing-matrix by hand. When k is specified, the elements of B are drawn randomly from a Uniform(0, 1) distribution. This is subject to change, and may not be reproducible. k defaults to NULL. You must specify either k or B, but not both.

B (mixing matrix) A k by k matrix of block connection probabilities. The probability that a node in block i connects to a node in community j is Poisson(B[i, j]). Must be a square matrix. matrix and Matrix objects are both acceptable. If B is not symmetric, it will be symmetrized via the update B := B + t(B). Defaults to NULL. You must specify either k or B, but not both.

... Arguments passed on to undirected_factor_model

expected_degree If specified, the desired expected degree of the graph. Specifying expected_degree simply rescales S to achieve this. Defaults to NULL. Do not specify both expected_degree and expected_density at the same time.

expected_density If specified, the desired expected density of the graph. Specifying expected_density simply rescales S to achieve this. Defaults to NULL. Do not specify both expected_degree and expected_density at the same time.

alpha (relative block propensities) Relative block propensities, which are parameters of a Dirichlet distribution. All elments of alpha must thus be positive. Must match the dimensions of B or k. Defaults to rep(1, k), or balanced membership across blocks.

sort_nodes Logical indicating whether or not to sort the nodes so that they are grouped by block and by theta. Useful for plotting. Defaults to TRUE.

force_pure Logical indicating whether or not to force presence of "pure nodes" (nodes that belong only to a single community) for the sake of identifiability. To include pure nodes, block membership sampling first proceeds as per usual. Then, after it is complete, k nodes are chosen randomly as pure nodes, one for each block. Defaults to TRUE.

poisson_edges Logical indicating whether or not multiple edges are allowed to form between a pair of nodes. Defaults to TRUE. When FALSE, sampling proceeds as usual, and duplicate edges are removed afterwards. Further, when FALSE, we assume that S specifies a desired between-factor connection probability, and back-transform this S to the appropriate Poisson intensity parameter to approximate Bernoulli factor connection probabilities. See Section 2.3 of Rohe et al. (2017) for some additional details.

allow_self_loops
Logical indicating whether or not nodes should be allowed to form edges with themselves. Defaults to TRUE. When FALSE, sampling proceeds allowing self-loops, and these are then removed after the fact.

Value

An undirected_mmsbm S3 object, a subclass of the undirected_factor_model() with the following additional fields:

- `theta`: A numeric vector of degree-heterogeneity parameters.
- `Z`: The community memberships of each node, a `matrix()` with `k` columns, whose row sums all equal one.
- `alpha`: Community membership proportion propensities.
- `sorted`: Logical indicating where nodes are arranged by block (and additionally by degree heterogeneity parameter) within each block.

Generative Model

There are two levels of randomness in a degree-corrected stochastic blockmodel. First, we randomly choose how much each node belongs to each block in the blockmodel. Each node is one unit of block membership to distribute. This is handled by `mmsbm()`. Then, given these block memberships, we randomly sample edges between nodes. This second operation is handled by `sample_edgelist()`, `sample_sparse()`, `sample_igraph()` and `sample_tidygraph()`, depending depending on your desired graph representation.

Block memberships:

Let Z_i by a vector on the `k` dimensional simplex representing the block memberships of node i. To generate z_i we sample from a Dirichlet distribution with parameter vector α. Block memberships for each node are independent.

Degree heterogeneity:

In addition to block membership, the MMSBM also allows nodes to have different propensities for edge formation. We represent this propensity for node i by a positive number θ_i.

Edge formulation:

Once we know the block membership vector z_i, z_j and the degree heterogeneity parameters θ, we need one more ingredient, which is the baseline intensity of connections between nodes in block `i` and block `j`. This is given by a $k \times k$ matrix B. Then each edge $A_{i,j}$ is Poisson distributed with parameter

$$\lambda_{i,j} = \theta_i \cdot z_i^T B z_j \cdot \theta_j.$$

See Also

Other stochastic block models: `dcsbm()`, `directed_dcsbm()`, `overlapping_sbm()`, `planted_partition()`, `sbm()`

Other undirected graphs: `chung_lu()`, `dcsbm()`, `erdos_renyi()`, `overlapping_sbm()`, `planted_partition()`, `sbm()`

Examples

```
set.seed(27)

lazy_mmsbm <- mmsbm(n = 1000, k = 5, expected_density = 0.01)
lazy_mmsbm

# sometimes you gotta let the world burn and
```

```
# sample a wildly dense graph

dense_lazy_mmsbm <- mmsbm(n = 500, k = 3, expected_density = 0.8)
dense_lazy_mmsbm

# explicitly setting the degree heterogeneity parameter,
# mixing matrix, and relative community sizes rather
# than using randomly generated defaults

k <- 5
n <- 1000
B <- matrix(stats::runif(k * k), nrow = k, ncol = k)

theta <- round(stats::rlnorm(n, 2))

alpha <- c(1, 2, 4, 1, 1)

custom_mmsbm <- mmsbm(
  theta = theta,
  B = B,
  alpha = alpha,
  expected_degree = 50
)

custom_mmsbm

edgelist <- sample_edgelist(custom_mmsbm)
edgelist

# efficient eigendecompostion that leverages low-rank structure in
# E(A) so that you don't have to form E(A) to find eigenvectors,
# as E(A) is typically dense. computation is
# handled via RSpectra

population_eigs <- eigs_sym(custom_mmsbm)
svds(custom_mmsbm)$d
```

`overlapping_sbm` *Create an undirected overlapping degree corrected stochastic blockmodel object*

Description

To specify a overlapping stochastic blockmodel, you must specify the number of nodes (via `n`), the mixing matrix (via `k` or `B`), and the block probabilities (optional, via `pi`). We provide defaults for most of these options to enable rapid exploration, or you can invest the effort for more control over the model parameters. We **strongly recommend** setting the `expected_degree` or `expected_density` argument to avoid large memory allocations associated with sampling large, dense graphs.

Usage

```
overlapping_sbm(
  n,
  k = NULL,
  B = NULL,
  ...,
  pi = rep(1/k, k),
  sort_nodes = TRUE,
  force_pure = TRUE,
  poisson_edges = TRUE,
  allow_self_loops = TRUE
)
```

Arguments

n	The number of nodes in the overlapping SBM.
k	(mixing matrix) The number of blocks in the blockmodel. Use when you don't want to specify the mixing-matrix by hand. When k is specified, B is set to a diagonal dominant matrix with value 0.8 along the diagonal and 0.1 / (k - 1) on the off-diagonal. k defaults to NULL. You must specify either k or B, but not both.
B	(mixing matrix) A k by k matrix of block connection probabilities. The probability that a node in block i connects to a node in community j is Poisson(B[i, j]). Must be an *invertible*, symmetric square matrix. matrix and Matrix objects are both acceptable. If B is not symmetric, it will be symmetrized via the update B := B + t(B). Defaults to NULL. You must specify either k or B, but not both.
...	Arguments passed on to undirected_factor_model expected_degree If specified, the desired expected degree of the graph. Specifying expected_degree simply rescales S to achieve this. Defaults to NULL. Do not specify both expected_degree and expected_density at the same time. expected_density If specified, the desired expected density of the graph. Specifying expected_density simply rescales S to achieve this. Defaults to NULL. Do not specify both expected_degree and expected_density at the same time.
pi	(block probabilities) Probability of membership in each block. Membership in each block is independent under the overlapping SBM. Defaults to rep(1 / k, k).
sort_nodes	Logical indicating whether or not to sort the nodes so that they are grouped by block. Useful for plotting. Defaults to TRUE.
force_pure	Logical indicating whether or not to force presence of "pure nodes" (nodes that belong only to a single community) for the sake of identifiability. To include pure nodes, block membership sampling first proceeds as per usual. Then, after it is complete, k nodes are chosen randomly as pure nodes, one for each block. Defaults to TRUE.

`poisson_edges` Logical indicating whether or not multiple edges are allowed to form between a pair of nodes. Defaults to `TRUE`. When `FALSE`, sampling proceeds as usual, and duplicate edges are removed afterwards. Further, when `FALSE`, we assume that `S` specifies a desired between-factor connection probability, and back-transform this `S` to the appropriate Poisson intensity parameter to approximate Bernoulli factor connection probabilities. See Section 2.3 of Rohe et al. (2017) for some additional details.

`allow_self_loops` Logical indicating whether or not nodes should be allowed to form edges with themselves. Defaults to `TRUE`. When `FALSE`, sampling proceeds allowing self-loops, and these are then removed after the fact.

Value

An `undirected_overlapping_sbm` S3 object, a subclass of the `undirected_factor_model()` with the following additional fields:

- `pi`: Sampling probabilities for each block.
- `sorted`: Logical indicating where nodes are arranged by block (and additionally by degree heterogeneity parameter) within each block.

Generative Model

There are two levels of randomness in a degree-corrected overlapping stochastic blockmodel. First, for each node, we independently determine if that node is a member of each block. This is handled by `overlapping_sbm()`. Then, given these block memberships, we randomly sample edges between nodes. This second operation is handled by `sample_edgelist()`, `sample_sparse()`, `sample_igraph()` and `sample_tidygraph()`, depending depending on your desired graph representation.

Identifiability:

In order to be identifiable, an overlapping SBM must satisfy two conditions:

1. `B` must be invertible, and
2. the must be at least one "pure node" in each block that belongs to no other blocks.

Block memberships:

Note that some nodes may not belong to any blocks.

TODO

Edge formulation:

Once we know the block memberships, we need one more ingredient, which is the baseline intensity of connections between nodes in block `i` and block `j`. Then each edge $A_{i,j}$ is Poisson distributed with parameter

TODO

References

Kaufmann, Emilie, Thomas Bonald, and Marc Lelarge. "A Spectral Algorithm with Additive Clustering for the Recovery of Overlapping Communities in Networks," Vol. 9925. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016. https://doi.org/10.1007/978-3-319-46379-7.

Latouche, Pierre, Etienne Birmelé, and Christophe Ambroise. "Overlapping Stochastic Block Models with Application to the French Political Blogosphere." The Annals of Applied Statistics 5, no. 1 (March 2011): 309–36. https://doi.org/10.1214/10-AOAS382.

Zhang, Yuan, Elizaveta Levina, and Ji Zhu. "Detecting Overlapping Communities in Networks Using Spectral Methods." ArXiv:1412.3432, December 10, 2014. http://arxiv.org/abs/1412.3432.

See Also

Other stochastic block models: `dcsbm()`, `directed_dcsbm()`, `mmsbm()`, `planted_partition()`, `sbm()`

Other undirected graphs: `chung_lu()`, `dcsbm()`, `erdos_renyi()`, `mmsbm()`, `planted_partition()`, `sbm()`

Examples

```
set.seed(27)

lazy_overlapping_sbm <- overlapping_sbm(n = 1000, k = 5, expected_density = 0.01)
lazy_overlapping_sbm

# sometimes you gotta let the world burn and
# sample a wildly dense graph

dense_lazy_overlapping_sbm <- overlapping_sbm(n = 500, k = 3, expected_density = 0.8)
dense_lazy_overlapping_sbm

k <- 5
n <- 1000
B <- matrix(stats::runif(k * k), nrow = k, ncol = k)

pi <- c(1, 2, 4, 1, 1) / 5

custom_overlapping_sbm <- overlapping_sbm(
  n = 200,
  B = B,
  pi = pi,
  expected_degree = 5
)

custom_overlapping_sbm

edgelist <- sample_edgelist(custom_overlapping_sbm)
edgelist
```

```
# efficient eigendecompostion that leverages low-rank structure in
# E(A) so that you don't have to form E(A) to find eigenvectors,
# as E(A) is typically dense. computation is
# handled via RSpectra

population_eigs <- eigs_sym(custom_overlapping_sbm)
```

---

planted_partition *Create an undirected planted partition object*

Description

To specify a planted partition model, you must specify the number of nodes (via `n`), the mixing matrix (optional, either via `within_block`/`between_block` or `a`/`b`), and the relative block probabilites (optional, via `pi`). We provide defaults for most of these options to enable rapid exploration, or you can invest the effort for more control over the model parameters. We **strongly recommend** setting the `expected_degree` or `expected_density` argument to avoid large memory allocations associated with sampling large, dense graphs.

Usage

```
planted_partition(
  n,
  k,
  ...,
  within_block = NULL,
  between_block = NULL,
  a = NULL,
  b = NULL,
  pi = rep(1/k, k),
  sort_nodes = TRUE,
  poisson_edges = TRUE,
  allow_self_loops = TRUE
)
```

Arguments

`n`	The number of nodes in the network. Must be a positive integer. This argument is required.
`k`	Number of planted partitions, as a positive integer. This argument is required.
`...`	Arguments passed on to `undirected_factor_model` `expected_degree` If specified, the desired expected degree of the graph. Specifying `expected_degree` simply rescales `S` to achieve this. Defaults to `NULL`. Do not specify both `expected_degree` and `expected_density` at the same time.

expected_density
: If specified, the desired expected density of the graph. Specifying expected_density simply rescales S to achieve this. Defaults to NULL. Do not specify both expected_degree and expected_density at the same time.

within_block
: Probability of within block edges. Must be strictly between zero and one. Must specify either within_block and between_block, or a and b to determine edge probabilities.

between_block
: Probability of between block edges. Must be strictly between zero and one. Must specify either within_block and between_block, or a and b to determine edge probabilities.

a
: Integer such that a/n is the probability of edges within a block. Useful for sparse graphs. Must specify either within_block and between_block, or a and b to determine edge probabilities.

b
: Integer such that b/n is the probability of edges between blocks. Useful for sparse graphs. Must specify either within_block and between_block, or a and b to determine edge probabilities.

pi
: (relative block probabilities) Relative block probabilities. Must be positive, but do not need to sum to one, as they will be normalized internally. Must match the dimensions of B or k. Defaults to rep(1 / k, k), or a balanced blocks.

sort_nodes
: Logical indicating whether or not to sort the nodes so that they are grouped by block and by theta. Useful for plotting. Defaults to TRUE.

poisson_edges
: Logical indicating whether or not multiple edges are allowed to form between a pair of nodes. Defaults to TRUE. When FALSE, sampling proceeds as usual, and duplicate edges are removed afterwards. Further, when FALSE, we assume that S specifies a desired between-factor connection probability, and back-transform this S to the appropriate Poisson intensity parameter to approximate Bernoulli factor connection probabilities. See Section 2.3 of Rohe et al. (2017) for some additional details.

allow_self_loops
: Logical indicating whether or not nodes should be allowed to form edges with themselves. Defaults to TRUE. When FALSE, sampling proceeds allowing self-loops, and these are then removed after the fact.

Details

A planted partition model is stochastic blockmodel in which the diagonal and the off-diagonal of the mixing matrix B are both constant. This means that edge probabilities depend only on whether two nodes belong to the same block, or to different blocks, but the particular blocks themselves don't have any impact apart from this.

Value

An undirected_planted_partition S3 object, which is a subclass of the sbm() object, with additional fields:

- within_block: The probability of edge formation within a block.
- between_block: The probability of edge formation between two distinct blocks.

See Also

Other stochastic block models: dcsbm(), directed_dcsbm(), mmsbm(), overlapping_sbm(), sbm()

Other undirected graphs: chung_lu(), dcsbm(), erdos_renyi(), mmsbm(), overlapping_sbm(), sbm()

Examples

```
set.seed(27)

lazy_pp <- planted_partition(
  n = 1000,
  k = 5,
  expected_density = 0.01,
  within_block = 0.1,
  between_block = 0.01
)

lazy_pp
```

---

sample_edgelist   *Sample a random edgelist from a random dot product graph*

Description

There are two steps to using the fastRG package. First, you must parameterize a random dot product graph by sampling the latent factors. Use functions such as dcsbm(), sbm(), etc, to perform this specification. Then, use sample_*() functions to generate a random graph in your preferred format.

Usage

```
sample_edgelist(factor_model, ...)

## S3 method for class 'undirected_factor_model'
sample_edgelist(factor_model, ...)

## S3 method for class 'directed_factor_model'
sample_edgelist(factor_model, ...)
```

Arguments

factor_model	A directed_factor_model() or undirected_factor_model().
...	Ignored. Do not use.

Details

This function implements the `fastRG` algorithm as described in Rohe et al (2017). Please see the paper (which is short and open access!!) for details.

Value

A single realization of a random Poisson (or Bernoulli) Dot Product Graph, represented as a `tibble::tibble()` with two integer columns, `from` and `to`.

In the undirected case, `from` and `to` do not encode information about edge direction, but we will always have `from` <= `to` for convenience of edge identification. To avoid handling such considerations yourself, we recommend using `sample_sparse()`, `sample_igraph()`, and `sample_tidygraph()` over `sample_edgelist()`.

References

Rohe, Karl, Jun Tao, Xintian Han, and Norbert Binkiewicz. 2017. "A Note on Quickly Sampling a Sparse Matrix with Low Rank Expectation." Journal of Machine Learning Research; 19(77):1-13, 2018. https://www.jmlr.org/papers/v19/17-128.html

See Also

Other samplers: `sample_edgelist.matrix()`, `sample_igraph()`, `sample_sparse()`, `sample_tidygraph()`

Examples

```
library(igraph)
library(tidygraph)

set.seed(27)

##### undirected examples ----------------------------

n <- 100
k <- 5

X <- matrix(rpois(n = n * k, 1), nrow = n)
S <- matrix(runif(n = k * k, 0, .1), nrow = k)

# S will be symmetrized internal here, or left unchanged if
# it is already symmetric

ufm <- undirected_factor_model(
  X, S,
  expected_density = 0.1
)

ufm

### sampling graphs as edgelists ----------------------
```

```
edgelist <- sample_edgelist(ufm)
edgelist

### sampling graphs as sparse matrices ----------------

A <- sample_sparse(ufm)

inherits(A, "dsCMatrix")
isSymmetric(A)
dim(A)

B <- sample_sparse(ufm)

inherits(B, "dsCMatrix")
isSymmetric(B)
dim(B)

### sampling graphs as igraph graphs ------------------

sample_igraph(ufm)

### sampling graphs as tidygraph graphs ---------------

sample_tidygraph(ufm)

##### directed examples ----------------------------

n2 <- 100

k1 <- 5
k2 <- 3

d <- 50

X <- matrix(rpois(n = n2 * k1, 1), nrow = n2)
S <- matrix(runif(n = k1 * k2, 0, .1), nrow = k1, ncol = k2)
Y <- matrix(rexp(n = k2 * d, 1), nrow = d)

fm <- directed_factor_model(X, S, Y, expected_in_degree = 2)
fm

### sampling graphs as edgelists ----------------------

edgelist2 <- sample_edgelist(fm)
edgelist2

### sampling graphs as sparse matrices ----------------

A2 <- sample_sparse(fm)

inherits(A2, "dgCMatrix")
isSymmetric(A2)
dim(A2)
```

```
B2 <- sample_sparse(fm)

inherits(B2, "dgCMatrix")
isSymmetric(B2)
dim(B2)

### sampling graphs as igraph graphs ------------------

# since the number of rows and the number of columns
# in `fm` differ, we will get a bipartite igraph here

# creating the bipartite igraph is slow relative to other
# sampling -- if this is a blocker for
# you please open an issue and we can investigate speedups

dig <- sample_igraph(fm)
is_bipartite(dig)

### sampling graphs as tidygraph graphs ---------------

sample_tidygraph(fm)
```

---

sample_edgelist.matrix *Low level interface to sample RPDG edgelists*

Description

This is a breaks-off, no safety checks interface. We strongly recommend that you do not call `sample_edgelist.matrix()` unless you know what you are doing, and even then, we still do not recommend it, as you will bypass all typical input validation. **extremely loud coughing** All those who bypass input validation suffer foolishly at their own hand. **extremely loud coughing**

Usage

```
## S3 method for class 'matrix'
sample_edgelist(
  factor_model,
  S,
  Y,
  directed,
  poisson_edges,
  allow_self_loops,
  ...
)

## S3 method for class 'Matrix'
```

```
sample_edgelist(
  factor_model,
  S,
  Y,
  directed,
  poisson_edges,
  allow_self_loops,
  ...
)
```

Arguments

`factor_model` An `n` by `k1` `matrix()` or `Matrix::Matrix()` of latent node positions encoding incoming edge community membership. The `X` matrix in Rohe et al (2017). Naming differs only for consistency with the S3 generic.

`S` A `k1` by `k2` mixing `matrix()` or `Matrix::Matrix()`. In the undirect case this is assumed to be symmetric but **we do not check that this is the case**.

`Y` A `d` by `k2` `matrix()` or `Matrix::Matrix()` of latent node positions encoding outgoing edge community membership.

`directed` Logical indicating whether or not the graph should be directed. When `directed = FALSE`, symmetrizes `S` internally. `Y = X` together with a symmetric `S` implies a symmetric expectation (although not necessarily an undirected graph). When `directed = FALSE`, samples a directed graph with symmetric expectation, and then adds edges until symmetry is achieved.

`poisson_edges` Whether or not to remove duplicate edges after sampling. See Section 2.3 of Rohe et al. (2017) for some additional details. Defaults to `TRUE`.

`allow_self_loops`
Logical indicating whether or not nodes should be allowed to form edges with themselves. Defaults to `TRUE`. When `FALSE`, sampling proceeds allowing self-loops, and these are then removed after the fact.

`...` Ignored, for generic consistency only.

Details

This function implements the `fastRG` algorithm as described in Rohe et al (2017). Please see the paper (which is short and open access!!) for details.

Value

A single realization of a random Poisson (or Bernoulli) Dot Product Graph, represented as a `tibble::tibble()` with two integer columns, `from` and `to`.

In the undirected case, `from` and `to` do not encode information about edge direction, but we will always have `from <= to` for convenience of edge identification. To avoid handling such considerations yourself, we recommend using `sample_sparse()`, `sample_igraph()`, and `sample_tidygraph()` over `sample_edgelist()`.

References

Rohe, Karl, Jun Tao, Xintian Han, and Norbert Binkiewicz. 2017. "A Note on Quickly Sampling a Sparse Matrix with Low Rank Expectation." Journal of Machine Learning Research; 19(77):1-13, 2018. https://www.jmlr.org/papers/v19/17-128.html

See Also

Other samplers: sample_edgelist(), sample_igraph(), sample_sparse(), sample_tidygraph()

Examples

```
set.seed(46)

n <- 10000
d <- 1000

k1 <- 5
k2 <- 3

X <- matrix(rpois(n = n * k1, 1), nrow = n)
S <- matrix(runif(n = k1 * k2, 0, .1), nrow = k1)
Y <- matrix(rpois(n = d * k2, 1), nrow = d)

sample_edgelist(X, S, Y, TRUE, TRUE, TRUE)
```

---

sample_igraph *Sample a random dot product graph as an igraph graph*

Description

There are two steps to using the fastRG package. First, you must parameterize a random dot product graph by sampling the latent factors. Use functions such as dcsbm(), sbm(), etc, to perform this specification. Then, use sample_*() functions to generate a random graph in your preferred format.

Usage

```
sample_igraph(factor_model, ...)

## S3 method for class 'undirected_factor_model'
sample_igraph(factor_model, ...)

## S3 method for class 'directed_factor_model'
sample_igraph(factor_model, ...)
```

Arguments

factor_model — A directed_factor_model() or undirected_factor_model().

... — Ignored. Do not use.

Details

This function implements the fastRG algorithm as described in Rohe et al (2017). Please see the paper (which is short and open access!!) for details.

Value

An igraph::igraph() object that is possibly a multigraph (that is, we take there to be multiple edges rather than weighted edges).

When factor_model is **undirected**:

```
- the graph is undirected and one-mode.
```

When factor_model is **directed** and **square**:

```
- the graph is directed and one-mode.
```

When factor_model is **directed** and **rectangular**:

```
- the graph is undirected and bipartite.
```

Note that working with bipartite graphs in igraph is more complex than working with one-mode graphs.

References

Rohe, Karl, Jun Tao, Xintian Han, and Norbert Binkiewicz. 2017. "A Note on Quickly Sampling a Sparse Matrix with Low Rank Expectation." Journal of Machine Learning Research; 19(77):1-13, 2018. https://www.jmlr.org/papers/v19/17-128.html

See Also

Other samplers: sample_edgelist.matrix(), sample_edgelist(), sample_sparse(), sample_tidygraph()

Examples

```
library(igraph)
library(tidygraph)

set.seed(27)

##### undirected examples ----------------------------

n <- 100
```

```
k <- 5

X <- matrix(rpois(n = n * k, 1), nrow = n)
S <- matrix(runif(n = k * k, 0, .1), nrow = k)

# S will be symmetrized internal here, or left unchanged if
# it is already symmetric

ufm <- undirected_factor_model(
  X, S,
  expected_density = 0.1
)

ufm

### sampling graphs as edgelists ----------------------

edgelist <- sample_edgelist(ufm)
edgelist

### sampling graphs as sparse matrices ----------------

A <- sample_sparse(ufm)

inherits(A, "dsCMatrix")
isSymmetric(A)
dim(A)

B <- sample_sparse(ufm)

inherits(B, "dsCMatrix")
isSymmetric(B)
dim(B)

### sampling graphs as igraph graphs ------------------

sample_igraph(ufm)

### sampling graphs as tidygraph graphs ---------------

sample_tidygraph(ufm)

##### directed examples ----------------------------

n2 <- 100

k1 <- 5
k2 <- 3

d <- 50

X <- matrix(rpois(n = n2 * k1, 1), nrow = n2)
S <- matrix(runif(n = k1 * k2, 0, .1), nrow = k1, ncol = k2)
```

```
Y <- matrix(rexp(n = k2 * d, 1), nrow = d)

fm <- directed_factor_model(X, S, Y, expected_in_degree = 2)
fm

### sampling graphs as edgelists ----------------------

edgelist2 <- sample_edgelist(fm)
edgelist2

### sampling graphs as sparse matrices ----------------

A2 <- sample_sparse(fm)

inherits(A2, "dgCMatrix")
isSymmetric(A2)
dim(A2)

B2 <- sample_sparse(fm)

inherits(B2, "dgCMatrix")
isSymmetric(B2)
dim(B2)

### sampling graphs as igraph graphs ------------------

# since the number of rows and the number of columns
# in `fm` differ, we will get a bipartite igraph here

# creating the bipartite igraph is slow relative to other
# sampling -- if this is a blocker for
# you please open an issue and we can investigate speedups

dig <- sample_igraph(fm)
is_bipartite(dig)

### sampling graphs as tidygraph graphs ---------------

sample_tidygraph(fm)
```

`sample_sparse` *Sample a random dot product graph as a sparse Matrix*

Description

There are two steps to using the `fastRG` package. First, you must parameterize a random dot product graph by sampling the latent factors. Use functions such as `dcsbm()`, `sbm()`, etc, to perform this specification. Then, use `sample_*()` functions to generate a random graph in your preferred format.

Usage

```
sample_sparse(factor_model, ...)

## S3 method for class 'undirected_factor_model'
sample_sparse(factor_model, ...)

## S3 method for class 'directed_factor_model'
sample_sparse(factor_model, ...)
```

Arguments

`factor_model`	A `directed_factor_model()` or `undirected_factor_model()`.
`...`	Ignored. Do not use.

Details

This function implements the `fastRG` algorithm as described in Rohe et al (2017). Please see the paper (which is short and open access!!) for details.

Value

For undirected factor models, a sparse `Matrix::Matrix()` of class `dsCMatrix`. In particular, this means the `Matrix` object (1) has double data type, (2) is symmetric, and (3) is in column compressed storage format.

For directed factor models, a sparse `Matrix::Matrix()` of class `dgCMatrix`. This means the `Matrix` object (1) has double data type, (2) in *not* symmetric, and (3) is in column compressed storage format.

To reiterate: for undirected graphs, you will get a symmetric matrix. For directed graphs, you will get a general sparse matrix.

References

Rohe, Karl, Jun Tao, Xintian Han, and Norbert Binkiewicz. 2017. "A Note on Quickly Sampling a Sparse Matrix with Low Rank Expectation." Journal of Machine Learning Research; 19(77):1-13, 2018. https://www.jmlr.org/papers/v19/17-128.html

See Also

Other samplers: `sample_edgelist.matrix()`, `sample_edgelist()`, `sample_igraph()`, `sample_tidygraph()`

Examples

```
library(igraph)
library(tidygraph)

set.seed(27)

##### undirected examples ----------------------------
```

```
n <- 100
k <- 5

X <- matrix(rpois(n = n * k, 1), nrow = n)
S <- matrix(runif(n = k * k, 0, .1), nrow = k)

# S will be symmetrized internal here, or left unchanged if
# it is already symmetric

ufm <- undirected_factor_model(
  X, S,
  expected_density = 0.1
)

ufm

### sampling graphs as edgelists ----------------------

edgelist <- sample_edgelist(ufm)
edgelist

### sampling graphs as sparse matrices ----------------

A <- sample_sparse(ufm)

inherits(A, "dsCMatrix")
isSymmetric(A)
dim(A)

B <- sample_sparse(ufm)

inherits(B, "dsCMatrix")
isSymmetric(B)
dim(B)

### sampling graphs as igraph graphs ------------------

sample_igraph(ufm)

### sampling graphs as tidygraph graphs ---------------

sample_tidygraph(ufm)

##### directed examples ----------------------------

n2 <- 100

k1 <- 5
k2 <- 3

d <- 50
```

```
X <- matrix(rpois(n = n2 * k1, 1), nrow = n2)
S <- matrix(runif(n = k1 * k2, 0, .1), nrow = k1, ncol = k2)
Y <- matrix(rexp(n = k2 * d, 1), nrow = d)

fm <- directed_factor_model(X, S, Y, expected_in_degree = 2)
fm

### sampling graphs as edgelists ----------------------

edgelist2 <- sample_edgelist(fm)
edgelist2

### sampling graphs as sparse matrices ----------------

A2 <- sample_sparse(fm)

inherits(A2, "dgCMatrix")
isSymmetric(A2)
dim(A2)

B2 <- sample_sparse(fm)

inherits(B2, "dgCMatrix")
isSymmetric(B2)
dim(B2)

### sampling graphs as igraph graphs ------------------

# since the number of rows and the number of columns
# in `fm` differ, we will get a bipartite igraph here

# creating the bipartite igraph is slow relative to other
# sampling -- if this is a blocker for
# you please open an issue and we can investigate speedups

dig <- sample_igraph(fm)
is_bipartite(dig)

### sampling graphs as tidygraph graphs ---------------

sample_tidygraph(fm)
```

`sample_tidygraph` *Sample a random dot product graph as a tidygraph graph*

Description

There are two steps to using the `fastRG` package. First, you must parameterize a random dot product graph by sampling the latent factors. Use functions such as `dcsbm()`, `sbm()`, etc, to perform this specification. Then, use `sample_*()` functions to generate a random graph in your preferred format.

Usage

```
sample_tidygraph(factor_model, ...)

## S3 method for class 'undirected_factor_model'
sample_tidygraph(factor_model, ...)

## S3 method for class 'directed_factor_model'
sample_tidygraph(factor_model, ...)
```

Arguments

factor_model	A directed_factor_model() or undirected_factor_model().
...	Ignored. Do not use.

Details

This function implements the `fastRG` algorithm as described in Rohe et al (2017). Please see the paper (which is short and open access!!) for details.

Value

A `tidygraph::tbl_graph()` object that is possibly a multigraph (that is, we take there to be multiple edges rather than weighted edges).

When `factor_model` is **undirected**:

```
- the graph is undirected and one-mode.
```

When `factor_model` is **directed** and **square**:

```
- the graph is directed and one-mode.
```

When `factor_model` is **directed** and **rectangular**:

```
- the graph is undirected and bipartite.
```

Note that working with bipartite graphs in `tidygraph` is more complex than working with one-mode graphs.

References

Rohe, Karl, Jun Tao, Xintian Han, and Norbert Binkiewicz. 2017. "A Note on Quickly Sampling a Sparse Matrix with Low Rank Expectation." Journal of Machine Learning Research; 19(77):1-13, 2018. https://www.jmlr.org/papers/v19/17-128.html

See Also

Other samplers: `sample_edgelist.matrix()`, `sample_edgelist()`, `sample_igraph()`, `sample_sparse()`

Examples

```
library(igraph)
library(tidygraph)

set.seed(27)

##### undirected examples ----------------------------

n <- 100
k <- 5

X <- matrix(rpois(n = n * k, 1), nrow = n)
S <- matrix(runif(n = k * k, 0, .1), nrow = k)

# S will be symmetrized internal here, or left unchanged if
# it is already symmetric

ufm <- undirected_factor_model(
  X, S,
  expected_density = 0.1
)

ufm

### sampling graphs as edgelists ----------------------

edgelist <- sample_edgelist(ufm)
edgelist

### sampling graphs as sparse matrices ----------------

A <- sample_sparse(ufm)

inherits(A, "dsCMatrix")
isSymmetric(A)
dim(A)

B <- sample_sparse(ufm)

inherits(B, "dsCMatrix")
isSymmetric(B)
dim(B)

### sampling graphs as igraph graphs ------------------

sample_igraph(ufm)

### sampling graphs as tidygraph graphs ---------------

sample_tidygraph(ufm)
```

```
##### directed examples ----------------------------

n2 <- 100

k1 <- 5
k2 <- 3

d <- 50

X <- matrix(rpois(n = n2 * k1, 1), nrow = n2)
S <- matrix(runif(n = k1 * k2, 0, .1), nrow = k1, ncol = k2)
Y <- matrix(rexp(n = k2 * d, 1), nrow = d)

fm <- directed_factor_model(X, S, Y, expected_in_degree = 2)
fm

### sampling graphs as edgelists ----------------------

edgelist2 <- sample_edgelist(fm)
edgelist2

### sampling graphs as sparse matrices ----------------

A2 <- sample_sparse(fm)

inherits(A2, "dgCMatrix")
isSymmetric(A2)
dim(A2)

B2 <- sample_sparse(fm)

inherits(B2, "dgCMatrix")
isSymmetric(B2)
dim(B2)

### sampling graphs as igraph graphs ------------------

# since the number of rows and the number of columns
# in `fm` differ, we will get a bipartite igraph here

# creating the bipartite igraph is slow relative to other
# sampling -- if this is a blocker for
# you please open an issue and we can investigate speedups

dig <- sample_igraph(fm)
is_bipartite(dig)

### sampling graphs as tidygraph graphs ---------------

sample_tidygraph(fm)
```

sbm *Create an undirected stochastic blockmodel object*

Description

To specify a stochastic blockmodel, you must specify the number of nodes (via n), the mixing matrix (via k or B), and the relative block probabilites (optional, via pi). We provide defaults for most of these options to enable rapid exploration, or you can invest the effort for more control over the model parameters. We **strongly recommend** setting the expected_degree or expected_density argument to avoid large memory allocations associated with sampling large, dense graphs.

Usage

```
sbm(
  n,
  k = NULL,
  B = NULL,
  ...,
  pi = rep(1/k, k),
  sort_nodes = TRUE,
  poisson_edges = TRUE,
  allow_self_loops = TRUE
)
```

Arguments

n — The number of nodes in the network. Must be a positive integer. This argument is required.

k — (mixing matrix) The number of blocks in the blockmodel. Use when you don't want to specify the mixing-matrix by hand. When k is specified, the elements of B are drawn randomly from a Uniform(0, 1) distribution. This is subject to change, and may not be reproducible. k defaults to NULL. You must specify either k or B, but not both.

B — (mixing matrix) A k by k matrix of block connection probabilities. The probability that a node in block i connects to a node in community j is Poisson(B[i, j]). Must be a square matrix. matrix and Matrix objects are both acceptable. If B is not symmetric, it will be symmetrized via the update B := B + t(B). Defaults to NULL. You must specify either k or B, but not both.

... — Arguments passed on to undirected_factor_model

expected_degree If specified, the desired expected degree of the graph. Specifying expected_degree simply rescales S to achieve this. Defaults to NULL. Do not specify both expected_degree and expected_density at the same time.

expected_density If specified, the desired expected density of the graph. Specifying expected_density simply rescales S to achieve this. Defaults to

NULL. Do not specify both expected_degree and expected_density at the same time.

pi
: (relative block probabilities) Relative block probabilities. Must be positive, but do not need to sum to one, as they will be normalized internally. Must match the dimensions of B or k. Defaults to rep(1 / k, k), or a balanced blocks.

sort_nodes
: Logical indicating whether or not to sort the nodes so that they are grouped by block and by theta. Useful for plotting. Defaults to TRUE.

poisson_edges
: Logical indicating whether or not multiple edges are allowed to form between a pair of nodes. Defaults to TRUE. When FALSE, sampling proceeds as usual, and duplicate edges are removed afterwards. Further, when FALSE, we assume that S specifies a desired between-factor connection probability, and back-transform this S to the appropriate Poisson intensity parameter to approximate Bernoulli factor connection probabilities. See Section 2.3 of Rohe et al. (2017) for some additional details.

allow_self_loops
: Logical indicating whether or not nodes should be allowed to form edges with themselves. Defaults to TRUE. When FALSE, sampling proceeds allowing self-loops, and these are then removed after the fact.

Details

A stochastic block is equivalent to a degree-corrected stochastic blockmodel where the degree heterogeneity parameters have all been set equal to 1.

Value

An undirected_sbm S3 object, which is a subclass of the dcsbm() object.

See Also

Other stochastic block models: dcsbm(), directed_dcsbm(), mmsbm(), overlapping_sbm(), planted_partition()

Other undirected graphs: chung_lu(), dcsbm(), erdos_renyi(), mmsbm(), overlapping_sbm(), planted_partition()

Examples

```
set.seed(27)

lazy_sbm <- sbm(n = 1000, k = 5, expected_density = 0.01)
lazy_sbm

# by default we get a multigraph (i.e. multiple edges are
# allowed between the same two nodes). using bernoulli edges
# will with an adjacency matrix with only zeroes and ones

bernoulli_sbm <- sbm(
  n = 5000,
```

```
  k = 300,
  poisson_edges = FALSE,
  expected_degree = 8
)

bernoulli_sbm

edgelist <- sample_edgelist(bernoulli_sbm)
edgelist

A <- sample_sparse(bernoulli_sbm)

# only zeroes and ones!
sign(A)
```

---

svds.directed_factor_model
Compute the singular value decomposition of the expected adjacency matrix of a directed factor model

---

Description

Compute the singular value decomposition of the expected adjacency matrix of a directed factor model

Usage

```
## S3 method for class 'directed_factor_model'
svds(A, k = min(A$k1, A$k2), nu = k, nv = k, opts = list(), ...)
```

Arguments

A	An `undirected_factor_model()`.
k	Desired rank of decomposition.
nu	Number of left singular vectors to be computed. This must be between 0 and k.
nv	Number of right singular vectors to be computed. This must be between 0 and k.
opts	Control parameters related to the computing algorithm. See **Details** below.
...	Unused, included only for consistency with generic signature.

Details

The `opts` argument is a list that can supply any of the following parameters:

`ncv` Number of Lanzcos basis vectors to use. More vectors will result in faster convergence, but with greater memory use. `ncv` must be satisfy $k < ncv \le p$ where `p = min(m, n)`. Default is `min(p, max(2*k+1, 20))`.

`tol` Precision parameter. Default is 1e-10.

`maxitr` Maximum number of iterations. Default is 1000.

`center` Either a logical value (`TRUE`/`FALSE`), or a numeric vector of length n. If a vector c is supplied, then SVD is computed on the matrix $A - 1c'$, in an implicit way without actually forming this matrix. `center = TRUE` has the same effect as `center = colMeans(A)`. Default is `FALSE`.

`scale` Either a logical value (`TRUE`/`FALSE`), or a numeric vector of length n. If a vector s is supplied, then SVD is computed on the matrix $(A - 1c')S$, where c is the centering vector and $S = diag(1/s)$. If `scale = TRUE`, then the vector s is computed as the column norm of $A - 1c'$. Default is `FALSE`.

`svds.undirected_factor_model` *Compute the singular value decomposition of the expected adjacency matrix of an undirected factor model*

Description

Compute the singular value decomposition of the expected adjacency matrix of an undirected factor model

Usage

```
## S3 method for class 'undirected_factor_model'
svds(A, k = A$k, nu = k, nv = k, opts = list(), ...)
```

Arguments

`A`	An `undirected_factor_model()`.
`k`	Desired rank of decomposition.
`nu`	Number of left singular vectors to be computed. This must be between 0 and `k`.
`nv`	Number of right singular vectors to be computed. This must be between 0 and `k`.
`opts`	Control parameters related to the computing algorithm. See **Details** below.
`...`	Unused, included only for consistency with generic signature.

Details

The `opts` argument is a list that can supply any of the following parameters:

`ncv` Number of Lanzcos basis vectors to use. More vectors will result in faster convergence, but with greater memory use. `ncv` must be satisfy $k < ncv \leq p$ where `p = min(m, n)`. Default is `min(p, max(2*k+1, 20))`.

`tol` Precision parameter. Default is 1e-10.

`maxitr` Maximum number of iterations. Default is 1000.

`center` Either a logical value (`TRUE`/`FALSE`), or a numeric vector of length n. If a vector c is supplied, then SVD is computed on the matrix $A - 1c'$, in an implicit way without actually forming this matrix. `center = TRUE` has the same effect as `center = colMeans(A)`. Default is `FALSE`.

`scale` Either a logical value (`TRUE`/`FALSE`), or a numeric vector of length n. If a vector s is supplied, then SVD is computed on the matrix $(A - 1c')S$, where c is the centering vector and $S = diag(1/s)$. If `scale = TRUE`, then the vector s is computed as the column norm of $A - 1c'$. Default is `FALSE`.

`undirected_factor_model` *Create an undirected factor model graph*

Description

An undirected factor model graph is an undirected generalized Poisson random dot product graph. The edges in this graph are assumed to be independent and Poisson distributed. The graph is parameterized by its expected adjacency matrix, which is `E[A|X] = X S X'`. We do not recommend that casual users use this function, see instead `dcsbm()` and related functions, which will formulate common variants of the stochastic blockmodels as undirected factor models *with lots of helpful input validation*.

Usage

```
undirected_factor_model(
  X,
  S,
  ...,
  expected_degree = NULL,
  expected_density = NULL,
  poisson_edges = TRUE,
  allow_self_loops = TRUE
)
```

Arguments

`X` A `matrix()` or Matrix() representing real-valued latent node positions. Entries must be positive.

`S` A `matrix()` or Matrix() mixing matrix. `S` is symmetrized if it is not already, as this is the undirected case. Entries must be positive.

`...` Ignored. Must be empty.

`expected_degree` If specified, the desired expected degree of the graph. Specifying `expected_degree` simply rescales `S` to achieve this. Defaults to `NULL`. Do not specify both `expected_degree` and `expected_density` at the same time.

expected_density
: If specified, the desired expected density of the graph. Specifying expected_density simply rescales S to achieve this. Defaults to NULL. Do not specify both expected_degree and expected_density at the same time.

poisson_edges
: Logical indicating whether or not multiple edges are allowed to form between a pair of nodes. Defaults to TRUE. When FALSE, sampling proceeds as usual, and duplicate edges are removed afterwards. Further, when FALSE, we assume that S specifies a desired between-factor connection probability, and back-transform this S to the appropriate Poisson intensity parameter to approximate Bernoulli factor connection probabilities. See Section 2.3 of Rohe et al. (2017) for some additional details.

allow_self_loops
: Logical indicating whether or not nodes should be allowed to form edges with themselves. Defaults to TRUE. When FALSE, sampling proceeds allowing self-loops, and these are then removed after the fact.

Value

An undirected_factor_model S3 class based on a list with the following elements:

- X: The latent positions as a Matrix() object.
- S: The mixing matrix as a Matrix() object.
- n: The number of nodes in the network.
- k: The rank of expectation matrix. Equivalently, the dimension of the latent node position vectors.

Examples

```
n <- 10000
k <- 5

X <- matrix(rpois(n = n * k, 1), nrow = n)
S <- matrix(runif(n = k * k, 0, .1), nrow = k)

ufm <- undirected_factor_model(X, S)
ufm

ufm2 <- undirected_factor_model(X, S, expected_degree = 50)
ufm2

svds(ufm2)
```

Index