Package 'fastRG'

February 26, 2021

Title Sample Generalized Random Dot Product Graphs in Linear Time

Version 0.3.0

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. 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.

License MIT + file LICENSE

URL https://github.com/RoheLab/fastRG

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

Depends Matrix

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

Suggests covr, dplyr, ggplot2, knitr, rmarkdown, testthat (>= 2.1.0)

Config/testthat/edition 3

Config/testthat/parallel true

Encoding UTF-8

LazyData true

RoxygenNote 7.1.1

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 2021-02-26 09:40:03 UTC

R topics documented:

dcsbm . . . 2
directed_erdos_renyi . . . 5
directed_factor_model . . . 6
erdos_renyi . . . 8
expected_edges . . . 9
planted_partition . . . 10
sample_edgelist.directed_erdos_renyi . . . 13
sample_edgelist.matrix . . . 16
sample_igraph.directed_erdos_renyi . . . 19
sample_sparse.directed_erdos_renyi . . . 22
sample_tidygraph.directed_erdos_renyi . . . 26
sbm . . . 30
undirected_factor_model . . . 33

Index **35**

`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 probabilites (optional, via `pi`). We provide sane 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
)
```

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 square 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. Useful for plotting. Defaults to `TRUE`.

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 chosen 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 your desirable 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: `planted_partition()`, `sbm()`

Other undirected graphs: `erdos_renyi()`, `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_erdos_renyi` *Create an directed erdos renyi object*

Description

Create an directed erdos renyi object

Usage

```
directed_erdos_renyi(n, ..., p = NULL)
```

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`.

Value

Never returns Poisson edges.

See Also

Other bernoulli graphs: `erdos_renyi()`

Other erdos renyi: `erdos_renyi()`

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
)
```

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.

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).

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

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

`erdos_renyi` *Create an undirected erdos renyi object*

Description

Create an undirected erdos renyi object

Usage

```
erdos_renyi(n, ..., p = NULL)
```

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`.

Value

Never returns Poisson edges.

See Also

Other bernoulli graphs: `directed_erdos_renyi()`

Other erdos renyi: `directed_erdos_renyi()`

Other undirected graphs: `dcsbm()`, `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

```
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)
```

`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 sane 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),
  edge_distribution = c("poisson", "bernoulli"),
  sort_nodes = 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.

edge_distribution — Either "poisson" or "bernoulli". The default is "poisson", in which case the SBM can be a multigraph, i.e. multiple edges between the same two nodes are allowed. If edge_distribution == "bernoulli" only a single edge is allowed between any pair of nodes. See Section 2.3 of Rohe et al (2017) for details.

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

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(), sbm()

Other undirected graphs: dcsbm(), erdos_renyi(), 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.directed_erdos_renyi` *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

```
## S3 method for class 'directed_erdos_renyi'
sample_edgelist(factor_model, ..., allow_self_loops = TRUE)

sample_edgelist(
  factor_model,
  ...,
  poisson_edges = TRUE,
  allow_self_loops = TRUE
)

## S3 method for class 'undirected_factor_model'
sample_edgelist(
  factor_model,
  ...,
  poisson_edges = TRUE,
  allow_self_loops = TRUE
)

## S3 method for class 'directed_factor_model'
sample_edgelist(
  factor_model,
  ...,
  poisson_edges = TRUE,
  allow_self_loops = TRUE
)

## S3 method for class 'undirected_erdos_renyi'
sample_edgelist(
  factor_model,
  ...,
  poisson_edges = FALSE,
  allow_self_loops = TRUE
)

## S3 method for class 'undirected_sbm'
```

```
sample_edgelist(factor_model, ..., allow_self_loops = TRUE)

## S3 method for class 'undirected_sbm'
sample_edgelist(factor_model, ..., allow_self_loops = TRUE)
```

Arguments

factor_model	A directed_factor_model() or undirected_factor_model().
...	Ignored. Do not use.
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.
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. See Section 2.3 of Rohe et al (2017) for additional details.

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.directed_erdos_renyi(), sample_sparse.directed_e
sample_tidygraph.directed_erdos_renyi()

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, poisson_edges = FALSE)

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

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

sample_igraph(ufm)

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

sample_tidygraph(ufm)

sample_tidygraph(ufm, poisson_edges = FALSE)

##### 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, poisson_edges = FALSE)

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, poisson_edges = FALSE)
```

---

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 = TRUE,
  allow_self_loops = TRUE
)

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

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.
`...`	Ignored. Do not use.
`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. See Section 2.3 of Rohe et al (2017) for 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

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.directed_erdos_renyi()`, `sample_igraph.directed_erdos_renyi()`, `sample_sparse.directed_erdos_renyi()`, `sample_tidygraph.directed_erdos_renyi()`

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)
```

sample_igraph.directed_erdos_renyi
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

```
## S3 method for class 'directed_erdos_renyi'
sample_igraph(factor_model, ..., allow_self_loops = TRUE)

sample_igraph(factor_model, ..., poisson_edges = TRUE, allow_self_loops = TRUE)

## S3 method for class 'undirected_factor_model'
sample_igraph(factor_model, ..., poisson_edges = TRUE, allow_self_loops = TRUE)

## S3 method for class 'directed_factor_model'
sample_igraph(factor_model, ..., poisson_edges = TRUE, allow_self_loops = TRUE)

## S3 method for class 'undirected_erdos_renyi'
sample_igraph(
  factor_model,
  ...,
  poisson_edges = FALSE,
  allow_self_loops = TRUE
)

## S3 method for class 'undirected_erdos_renyi'
sample_igraph(
  factor_model,
  ...,
  poisson_edges = FALSE,
  allow_self_loops = TRUE
)
```

Arguments

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

`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. See Section 2.3 of Rohe et al (2017) for additional details.

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.directed_erdos_renyi()`, `sample_edgelist.matrix()`, `sample_sparse.directed_`
`sample_tidygraph.directed_erdos_renyi()`

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, poisson_edges = FALSE)

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

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

sample_igraph(ufm)

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

sample_tidygraph(ufm)

sample_tidygraph(ufm, poisson_edges = FALSE)

##### 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, poisson_edges = FALSE)

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, poisson_edges = FALSE)
```

`sample_sparse.directed_erdos_renyi`

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

```
## S3 method for class 'directed_erdos_renyi'
sample_sparse(factor_model, ..., allow_self_loops = TRUE)

sample_sparse(factor_model, ..., poisson_edges = TRUE, allow_self_loops = TRUE)

## S3 method for class 'undirected_factor_model'
sample_sparse(factor_model, ..., poisson_edges = TRUE, allow_self_loops = TRUE)

## S3 method for class 'directed_factor_model'
sample_sparse(factor_model, ..., poisson_edges = TRUE, allow_self_loops = TRUE)

## S3 method for class 'undirected_erdos_renyi'
sample_sparse(
  factor_model,
  ...,
  poisson_edges = FALSE,
  allow_self_loops = TRUE
)

## S3 method for class 'undirected_erdos_renyi'
sample_sparse(
  factor_model,
  ...,
  poisson_edges = FALSE,
  allow_self_loops = TRUE
)
```

Arguments

factor_model	A directed_factor_model() or undirected_factor_model().
...	Ignored. Do not use.
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.
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. See Section 2.3 of Rohe et al (2017) for additional details.

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.directed_erdos_renyi()`, `sample_edgelist.matrix()`, `sample_igraph.directed_`
`sample_tidygraph.directed_erdos_renyi()`

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, poisson_edges = FALSE)

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

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

sample_igraph(ufm)

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

sample_tidygraph(ufm)

sample_tidygraph(ufm, poisson_edges = FALSE)

##### 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, poisson_edges = FALSE)

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, poisson_edges = FALSE)
```

`sample_tidygraph.directed_erdos_renyi` *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

```
## S3 method for class 'directed_erdos_renyi'
sample_tidygraph(factor_model, ..., allow_self_loops = TRUE)

sample_tidygraph(
  factor_model,
  ...,
  poisson_edges = TRUE,
```

```
  allow_self_loops = TRUE
)

## S3 method for class 'undirected_factor_model'
sample_tidygraph(
  factor_model,
  ...,
  poisson_edges = TRUE,
  allow_self_loops = TRUE
)

## S3 method for class 'directed_factor_model'
sample_tidygraph(
  factor_model,
  ...,
  poisson_edges = TRUE,
  allow_self_loops = TRUE
)

## S3 method for class 'undirected_erdos_renyi'
sample_tidygraph(
  factor_model,
  ...,
  poisson_edges = FALSE,
  allow_self_loops = TRUE
)

## S3 method for class 'undirected_erdos_renyi'
sample_tidygraph(
  factor_model,
  ...,
  poisson_edges = FALSE,
  allow_self_loops = TRUE
)
```

Arguments

factor_model	A directed_factor_model() or undirected_factor_model().
...	Ignored. Do not use.
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.
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. See Section 2.3 of Rohe et al (2017) for additional details.

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.directed_erdos_renyi()`, `sample_edgelist.matrix()`, `sample_igraph.directed`
`sample_sparse.directed_erdos_renyi()`

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, poisson_edges = FALSE)

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

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

sample_igraph(ufm)

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

sample_tidygraph(ufm)

sample_tidygraph(ufm, poisson_edges = FALSE)

##### 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, poisson_edges = FALSE)

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, poisson_edges = FALSE)
```

---

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 sane 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),
  edge_distribution = c("poisson", "bernoulli"),
  sort_nodes = 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 square 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.
edge_distribution	Either "poisson" or "bernoulli". The default is "poisson", in which case the SBM can be a multigraph, i.e. multiple edges between the same two nodes are allowed. If edge_distribution == "bernoulli" only a single edge is allowed between any pair of nodes. See Section 2.3 of Rohe et al (2017) for details.
sort_nodes	Logical indicating whether or not to sort the nodes so that they are grouped by block. Useful for plotting. Defaults to TRUE.

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, with one additional field.

- `edge_distribution`: Either "poisson" or "bernoulli".

See Also

Other stochastic block models: `dcsbm()`, `planted_partition()`

Other undirected graphs: `dcsbm()`, `erdos_renyi()`, `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,
  edge_distribution = "bernoulli",
  expected_degree = 80
)

bernoulli_sbm

edgelist <- sample_edgelist(bernoulli_sbm)
edgelist

A <- sample_sparse(bernoulli_sbm)

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

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
)
```

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.

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

sane_ufm <- undirected_factor_model(X, S, expected_degree = 50)
sane_ufm
```

Index

∗ **bernoulli graphs**
  directed_erdos_renyi, 5
  erdos_renyi, 8
∗ **directed graphs**
  directed_erdos_renyi, 5
∗ **erdos renyi**
  directed_erdos_renyi, 5
  erdos_renyi, 8
∗ **samplers**
  sample_edgelist.directed_erdos_renyi, 13
  sample_edgelist.matrix, 16
  sample_igraph.directed_erdos_renyi, 19
  sample_sparse.directed_erdos_renyi, 22
  sample_tidygraph.directed_erdos_renyi, 26
∗ **stochastic block models**
  dcsbm, 2
  planted_partition, 10
  sbm, 30
∗ **undirected graphs**
  dcsbm, 2
  erdos_renyi, 8
  planted_partition, 10
  sbm, 30

dcsbm, 2, *8*, *12*, *32*
dcsbm(), *13*, *19*, *23*, *26*, *32*, *33*
directed_erdos_renyi, 5, *8*
directed_factor_model, *5*, 6
directed_factor_model(), *9*, *14*, *19*, *23*, *27*

erdos_renyi, *4*, *6*, 8, *12*, *32*
expected_degree *(*expected_edges*)*, 9
expected_degrees *(*expected_edges*)*, 9
expected_density *(*expected_edges*)*, 9
expected_edges, 9
expected_in_degree *(*expected_edges*)*, 9
expected_out_degree *(*expected_edges*)*, 9

factor(), *3*

igraph::igraph(), *20*

Matrix(), *7*, *33*
matrix(), *7*, *17*, *33*
Matrix::Matrix(), *17*, *24*

planted_partition, *4*, *8*, 10, *32*

sample_edgelist
  *(*sample_edgelist.directed_erdos_renyi*)*, 13
sample_edgelist(), *4*, *14*, *18*
sample_edgelist.directed_erdos_renyi, 13, *18*, *20*, *24*, *28*
sample_edgelist.Matrix
  *(*sample_edgelist.matrix*)*, 16
sample_edgelist.matrix, *14*, 16, *20*, *24*, *28*
sample_igraph
  *(*sample_igraph.directed_erdos_renyi*)*, 19
sample_igraph(), *4*, *14*, *18*
sample_igraph.directed_erdos_renyi, *14*, *18*, 19, *24*, *28*
sample_sparse
  *(*sample_sparse.directed_erdos_renyi*)*, 22
sample_sparse(), *4*, *14*, *18*
sample_sparse.directed_erdos_renyi, *14*, *18*, *20*, 22, *28*
sample_tidygraph
  *(*sample_tidygraph.directed_erdos_renyi*)*, 26
sample_tidygraph(), *4*, *14*, *18*
sample_tidygraph.directed_erdos_renyi, *14*, *18*, *20*, *24*, 26
sbm, *4*, *8*, *12*, 30
sbm(), *12*, *13*, *19*, *23*, *26*

tibble::tibble(), *14*, *18*
tidygraph::tbl_graph(), *28*

undirected_factor_model, *3*, *8*, *11*, *31*, 33
undirected_factor_model(), *3*, *9*, *14*, *19*, *23*, *27*