Package: netify (via r-universe)

July 2, 2026

Title Tools for Network Data Workflows

Version 1.5.3

Description Builds, validates, analyzes, and visualizes network data from dyadic, event, matrix, 'igraph', and 'network' inputs. Supports cross-sectional, longitudinal, bipartite, and multi-layer networks, with conversion helpers for common modeling workflows and plotting utilities for exploratory analysis. Network methods are described in Wasserman and Faust (1994, ISBN:9780521387071), Cranmer et al. (2021) <doi:10.1017/9781316662915>, and Minhas et al. (2022) <doi:10.1017/psrm.2021.56>.

Depends R (>= 3.5.0)

Imports Rcpp, stats, methods, rlang, cli, checkmate, igraph, ggplot2 (>= 3.0.0), ggnewscale, ggrepel, withr

LinkingTo Rcpp, RcppEigen

Suggests knitr, rmarkdown, abind, tidyr, network, testthat (>= 3.0.0), dplyr, peacesciencer, amen, ergm, latentnet, RSiena, countrycode, scales, ggridges, ggbeeswarm, patchwork, RColorBrewer, tibble, broom, gganimate, generics, Matrix

VignetteBuilder knitr

License GPL-3

Encoding UTF-8

LazyData false

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

URL https://netify-dev.github.io/netify/

BugReports https://github.com/netify-dev/netify/issues

Config/pak/sysreqs libglpk-dev libxml2-dev

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

Date/Publication 2026-07-02 12:32:47 UTC

RemoteUrl https://github.com/netify-dev/netify

RemoteRef HEAD

RemoteSha d0dceb6a666e013651d017c5f6a55a51fee7bff2

Contents

add_dyad_vars *Add dyadic variables to a netify object*

Description

add_dyad_vars (also available as add_edge_attributes) merges additional dyadic (edge-level) variables from a data.frame into an existing netify object. this function allows you to incrementally build up the dyadic attributes of your network after initial creation, which is useful when variables come from different sources or need different preprocessing.

Usage

```
add_dyad_vars(
  netlet,
  dyad_data,
  actor1 = NULL,
  actor2 = NULL,
  time = NULL,
  dyad_vars = NULL,
  dyad_vars_symmetric = NULL,
  replace_existing = FALSE
)

add_edge_attributes(
  netlet,
  dyad_data,
  actor1 = NULL,
  actor2 = NULL,
  time = NULL,
  dyad_vars = NULL,
  dyad_vars_symmetric = NULL,
  replace_existing = FALSE
)
```

Arguments

netlet	a netify object (class "netify") to which dyadic variables will be added.
dyad_data	a data.frame object containing the dyadic variables to add. must include columns matching the actor1, actor2, and time specifications used in the original netify object. will be coerced to data.frame if a tibble or data.table is provided.
actor1	character string specifying the column name in dyad_data for the first actor in each dyad. should match the actor1 specification used when creating the netify object.
actor2	character string specifying the column name in dyad_data for the second actor in each dyad. should match the actor2 specification used when creating the netify object.

`time` character string specifying the column name in dyad_data for time periods. required for longitudinal netify objects. should match the time specification used when creating the netify object. set to NULL for cross-sectional networks.

`dyad_vars` character vector of column names from dyad_data to add as dyadic variables. if NULL (default), all columns except actor1, actor2, and time will be added.

`dyad_vars_symmetric` logical vector indicating whether each dyadic variable represents symmetric relationships. must have the same length as dyad_vars. if NULL, defaults to the symmetry setting of the netify object, but a warning will be issued recommending explicit specification.

`replace_existing` logical scalar. if TRUE, existing dyadic variables with the same names will be replaced. if FALSE (default), attempting to add variables that already exist will result in an error.

Details

dyadic variables are stored as matrix objects where rows represent the first actor (sender in directed networks) and columns represent the second actor (receiver in directed networks). for symmetric variables in undirected networks, the function ensures that `matrix[i,j]` equals `matrix[j,i]`.

the function optimizes storage by automatically detecting the data type of each variable and using the appropriate matrix storage mode:

- logical vectors -> logical matrices
- integer vectors -> integer matrices
- numeric vectors with only integer values -> integer matrices
- numeric vectors with decimals -> double matrices
- character vectors -> character matrices

for longitudinal networks, the function handles time-varying actor sets appropriately, creating matrices that include only actors present at each time point.

missing dyadic observations (na values) in the input data.frame will be set to missing in the resulting matrices as well.

Value

a netify object (class "netify") with the additional dyadic variables stored in the 'dyad_data' attribute. the structure is a nested list where:

- first level: named list with time periods as names (or "1" for cross-sectional data)
- second level: named list with variable names as names
- values: matrix objects with actors as rows/columns and numeric, integer, logical, or character values

Note

the input dyad_data must be a data.frame or an object that can be coerced into a data.frame (e.g., a tibble or data.table). inputs such as matrices or arrays are not supported.

when adding dyadic variables to bipartite networks, all variables are automatically treated as asymmetric regardless of the dyad_vars_symmetric specification.

for large networks, consider the memory implications of adding many dyadic variables, as each variable requires a full adjacency matrix for storage.

Author(s)

cassy dorff, colin henry, shahryar minhas

cassy dorff, shahryar minhas

Examples

```
# load example data
data(icews)

# cross-sectional example
icews_10 <- icews[icews$year == 2010, ]
actors <- sort(unique(c(icews_10$i, icews_10$j)))[1:35]
icews_10 <- icews_10[icews_10$i %in% actors & icews_10$j %in% actors, ]

# create initial netify object with just the main weight
verbCoop_net <- netify(
    icews_10, # data.frame input
    actor1 = "i", actor2 = "j",
    symmetric = FALSE,
    weight = "verbCoop"
)

# check class
class(verbCoop_net) # "netify"

# add additional dyadic variables
verbCoop_net <- add_dyad_vars(
    netlet = verbCoop_net, # netify object
    dyad_data = icews_10, # data.frame with variables to add
    actor1 = "i", actor2 = "j",
    dyad_vars = c("matlCoop", "verbConf", "matlConf"),
    dyad_vars_symmetric = rep(FALSE, 3)
)

# access the dyadic data structure (returns list)
dyad_data_structure <- attr(verbCoop_net, "dyad_data")
class(dyad_data_structure) # "list"
names(dyad_data_structure) # time periods
names(dyad_data_structure[["1"]]) # variables at time 1

# access specific variable matrix
```

```
matlCoop_matrix <- dyad_data_structure[["1"]][["matlCoop"]]
class(matlCoop_matrix) # "matrix" "array"
dim(matlCoop_matrix)
matlCoop_matrix[1:5, 1:5] # view subset

# longitudinal example
icews_panel <- icews[
    icews$year %in% 2002:2004 &
        icews$i %in% actors &
        icews$j %in% actors,
]

verbCoop_longit_net <- netify(
    icews_panel, # data.frame input
    actor1 = "i", actor2 = "j", time = "year",
    symmetric = FALSE,
    weight = "verbCoop"
)

# add dyadic variables across all time periods
verbCoop_longit_net <- add_dyad_vars(
    netlet = verbCoop_longit_net, # netify object
    dyad_data = icews_panel, # data.frame with longitudinal data
    actor1 = "i", actor2 = "j", time = "year",
    dyad_vars = c("matlCoop", "verbConf", "matlConf"),
    dyad_vars_symmetric = rep(FALSE, 3)
)

# access data for specific year (returns list)
year_2002_data <- attr(verbCoop_longit_net, "dyad_data")[["2002"]]
class(year_2002_data) # "list"
names(year_2002_data) # available variables

# each variable is stored as a matrix
matlCoop_2002 <- year_2002_data[["matlCoop"]]
class(matlCoop_2002) # "matrix" "array"

# example: add variables from a different source

# create a new data.frame with trade information
trade_data <- data.frame(
    i = icews_10$i,
    j = icews_10$j,
    trade_volume = runif(nrow(icews_10), 0, 1000),
    trade_balance = rnorm(nrow(icews_10))
)
class(trade_data) # "data.frame"

verbCoop_net <- add_dyad_vars(
    netlet = verbCoop_net,
    dyad_data = trade_data,
    actor1 = "i", actor2 = "j",
    dyad_vars = c("trade_volume", "trade_balance"),
```

```
    dyad_vars_symmetric = c(FALSE, FALSE)
)
```

---

add_node_vars *Add nodal variables to a netify object*

Description

add_node_vars (also available as add_vertex_attributes) merges nodal (vertex-level) variables from a data.frame into an existing netify object. this function allows you to incrementally build up the nodal attributes of your network after initial creation, which is useful when actor-level variables come from different sources or need different preprocessing.

Usage

```
add_node_vars(
  netlet,
  node_data,
  actor = NULL,
  time = NULL,
  node_vars = NULL,
  replace_existing = FALSE
)

add_vertex_attributes(
  netlet,
  node_data,
  actor = NULL,
  time = NULL,
  node_vars = NULL,
  replace_existing = FALSE
)
```

Arguments

netlet	a netify object (class "netify") to which nodal variables will be added.
node_data	a data.frame object containing the nodal variables to add. for cross-sectional networks, must have one row per unique actor. for longitudinal networks, must have one row per actor-time combination. will be coerced to data.frame if a tibble or data.table is provided.
actor	character string specifying the column name in node_data that uniquely identifies each actor. this should contain the same actor identifiers used in the original netify object.

`time` character string specifying the column name in node_data for time periods. required for longitudinal netify objects. should match the time specification used when creating the netify object. set to NULL for cross-sectional networks.

`node_vars` character vector of column names from node_data to add as nodal variables. if NULL (default), all columns except actor and time will be added.

`replace_existing` logical scalar. if TRUE, existing nodal variables with the same names will be replaced. if FALSE (default), attempting to add variables that already exist will result in an error.

Details

nodal variables are stored as a data.frame where each row represents an actor (cross-sectional) or an actor-time combination (longitudinal). this format allows for efficient storage and easy manipulation of actor-level attributes.

the function automatically handles merging based on actor identifiers, ensuring that nodal attributes are properly aligned with the network structure. for longitudinal networks, the function matches both actor and time dimensions.

missing actors in the node_data will result in na values for those actors' attributes in the netify object. similarly, if node_data contains actors not present in the network, those rows will be ignored.

Value

a netify object (class "netify") with the additional nodal variables stored in the 'nodal_data' attribute as a data.frame. the structure includes:

- **actor**: column with actor identifiers
- **time**: column with time periods (longitudinal only)
- **nodal variables**: columns for each variable specified in node_vars

Note

the input `node_data` must be a `data.frame` or an object that can be coerced into a `data.frame` (e.g., a `tibble` or `data.table`). inputs such as matrices or arrays are not supported.

for longitudinal networks, ensure that node_data contains entries for all actor-time combinations you wish to have attributes for. missing combinations will result in na values for those actors at those time points.

when working with large networks, the nodal data storage is more memory-efficient than dyadic data, as it scales linearly with the number of actors rather than quadratically.

Author(s)

colin henry, shahryar minhas

cassy dorff, shahryar minhas

Examples

```
# load example data
data(icews)

# cross-sectional example
icews_10 <- icews[icews$year == 2010, ]

# create initial netify object
verbCoop_net <- netify(
    icews_10, # data.frame input
    actor1 = "i", actor2 = "j",
    symmetric = FALSE,
    weight = "verbCoop"
)

# prepare nodal data - one row per unique actor
nvars <- c("i_polity2", "i_gdp", "i_log_gdp", "i_pop", "i_log_pop")
nodedata <- unique(icews_10[, c("i", nvars)])
class(nodedata) # "data.frame"
nrow(nodedata) # number of unique actors
head(nodedata)

# add nodal variables
verbCoop_net <- add_node_vars(
    netlet = verbCoop_net, # netify object
    node_data = nodedata, # data.frame with actor attributes
    actor = "i", # column identifying actors
    node_vars = nvars # variables to add
)

# access nodal data (returns data.frame)
node_data_stored <- attr(verbCoop_net, "nodal_data")
class(node_data_stored) # "data.frame"
head(node_data_stored)
names(node_data_stored) # "actor" plus variable names


# longitudinal example
verbCoop_longit_net <- netify(
    icews, # data.frame input
    actor1 = "i", actor2 = "j", time = "year",
    symmetric = FALSE,
    weight = "verbCoop"
)

# prepare longitudinal nodal data - one row per actor-time combination
nodedata_longit <- unique(icews[, c("i", "year", nvars)])

# add nodal variables with time dimension
verbCoop_longit_net <- add_node_vars(
    netlet = verbCoop_longit_net, # netify object
    node_data = nodedata_longit, # data.frame with longitudinal data
```

```
    actor = "i", # column identifying actors
    time = "year", # column identifying time
    node_vars = nvars # variables to add
)



# add variables from external source
# suppose you have additional actor data
external_data <- data.frame(
    i = unique(icews_10$i),
    democracy_score = runif(length(unique(icews_10$i)), 0, 10),
    trade_openness = runif(length(unique(icews_10$i)), 0, 100)
)

verbCoop_net <- add_node_vars(
    netlet = verbCoop_net,
    node_data = external_data,
    actor = "i",
    node_vars = c("democracy_score", "trade_openness")
)
```

`aggregate_dyad` *Aggregate dyadic event data by actor pairs*

Description

`aggregate_dyad` is designed for use with dyadic event datasets such as those from acled or icews. these datasets often contain multiple interactions between the same pair of actors–e.g., protest events, material cooperation, or verbal conflict–recorded at high frequency. this function aggregates such repeated observations into a single summary value per dyad, optionally within specified time periods. it is particularly useful for preparing network inputs by collapsing daily or monthly event-level data into actor-to-actor matrices.

Usage

```
aggregate_dyad(
  dyad_data,
  actor1,
  actor2,
  time = NULL,
  weight,
  symmetric,
  ignore_missing = TRUE
)
```

Arguments

dyad_data	a data.frame containing dyadic observations. must include columns for two actors and a weight variable. will be coerced to data.frame if a tibble or data.table is provided.
actor1	character string specifying the column name for the first actor in each dyad.
actor2	character string specifying the column name for the second actor in each dyad.
time	character string specifying the column name for time periods. if NULL (default), aggregation is performed across all time periods.
weight	character string specifying the column name containing values to be aggregated (summed) for each unique actor pair.
symmetric	logical. if TRUE, treats dyads as undirected (i.e., the dyad a-b is treated as identical to b-a). if FALSE, treats dyads as directed (i.e., a-b is distinct from b-a).
ignore_missing	logical. if TRUE (default), missing values in the weight variable are ignored during aggregation. if FALSE, any dyad containing a missing value will result in na for that aggregated dyad.

Details

the function handles both directed and undirected dyadic aggregation:

for symmetric (undirected) networks:

the function uses an efficient aggregation method that:

1. creates symmetric identifiers for each dyad using gen_symm_id (where a-b = b-a)
2. aggregates values by these symmetric identifiers
3. expands the results back to directed format for consistency with other netify functions

this ensures that interactions between actors a and b are combined regardless of direction, useful for undirected relationships like friendships or alliances.

for asymmetric (directed) networks:

standard aggregation is performed treating each directed dyad separately. this maintains the distinction between a->b and b->a relationships, which is important for directed interactions like exports/imports or sender/receiver communications.

missing value handling:

the ignore_missing parameter controls how na values are treated:

- when TRUE: missing values are excluded from the sum (e.g., sum(10, na, 20) = 30)
- when FALSE: any missing value results in na for that dyad (e.g., sum(10, na, 20) = na)

Value

a data.frame with unique actor pairs (and time periods if specified) and their aggregated weight values. the output contains columns:

- **actor1**: first actor in each dyad (using original column name)

- **actor2**: second actor in each dyad (using original column name)
- **time**: time period if time parameter was specified (using original column name)
- **weight**: aggregated (summed) values for each unique dyad (using original column name)

Note

the function preserves the original column names from the input data.frame in the output, making it easy to chain operations or merge results.

when symmetric = TRUE, the output still maintains a directed format (separate rows for a-b and b-a) with identical values for both directions. this ensures compatibility with other netify functions that expect directed dyadic data.

Author(s)

shahryar minhas

Examples

```
# load example data
data(icews)

# example 1: aggregate multiple events between countries
# the icews data contains multiple events per country pair
icews_2010 <- icews[icews$year == 2010, ]

# aggregate directed cooperation events
agg_coop <- aggregate_dyad(
    dyad_data = icews_2010,
    actor1 = "i",
    actor2 = "j",
    weight = "verbCoop",
    symmetric = FALSE
)

# check reduction in observations
nrow(icews_2010) # original observations
nrow(agg_coop) # unique directed dyads

# example 2: create symmetric trade volumes
trade_data <- data.frame(
    exporter = c("usa", "usa", "china", "china", "usa", "china"),
    importer = c("china", "china", "usa", "usa", "uk", "uk"),
    year = c(2020, 2020, 2020, 2021, 2021, 2021),
    trade_value = c(100, 50, 75, 80, 120, 90)
)

# aggregate as total trade between countries (undirected)
total_trade <- aggregate_dyad(
    dyad_data = trade_data,
    actor1 = "exporter",
    actor2 = "importer",
```

```
    time = "year",
    weight = "trade_value",
    symmetric = TRUE
)

# usa-china trade in 2020: 100+50+75 = 225 (appears in both directions)
total_trade[total_trade$year == 2020, ]

# example 3: aggregate across all time periods
all_time_trade <- aggregate_dyad(
    dyad_data = trade_data,
    actor1 = "exporter",
    actor2 = "importer",
    time = NULL, # aggregate across all years
    weight = "trade_value",
    symmetric = FALSE
)

# usa total exports to china: 100+50 = 150
all_time_trade

# example 4: handle missing values
trade_data_na <- trade_data
trade_data_na$trade_value[2] <- NA

# ignore missing values (default)
agg_ignore_na <- aggregate_dyad(
    dyad_data = trade_data_na,
    actor1 = "exporter",
    actor2 = "importer",
    time = "year",
    weight = "trade_value",
    symmetric = FALSE,
    ignore_missing = TRUE
)

# include missing values
agg_with_na <- aggregate_dyad(
    dyad_data = trade_data_na,
    actor1 = "exporter",
    actor2 = "importer",
    time = "year",
    weight = "trade_value",
    symmetric = FALSE,
    ignore_missing = FALSE
)

# compare results for usa->china in 2020
agg_ignore_na[agg_ignore_na$exporter == "usa" &
    agg_ignore_na$importer == "china" &
    agg_ignore_na$year == 2020, ] # 100 (ignored NA)

agg_with_na[agg_with_na$exporter == "usa" &
```

```
agg_with_na$importer == "china" &
agg_with_na$year == 2020, ] # NA
```

`animate_netify` *Animate a longitudinal netify object with gganimate*

Description

returns a ggplot built from the per-period `plot.netify()` output, with `gganimate::transition_manual()` keyed on the time variable so the animation steps through each period. requires the `gganimate` package.

Usage

```
animate_netify(netlet, ..., static_actor_positions = TRUE)
```

Arguments

`netlet`	a longitudinal netify object (`longit_array` / `longit_list`).
`...`	additional arguments passed to `plot.netify()` (`node_color_by`, `node_size_by`, `style`, etc.).
`static_actor_positions`	logical. if `TRUE` (default for animation since positions jumping around between periods is visually confusing), pin node positions across time.

Details

for static facet plots, just call `plot(net)` on a longit netlet – that defaults to faceting by time. use `animate_netify()` for single-panel transitions instead of grid faceting (better for presentations / videos where the eye can focus on one period at a time).

Value

a `gganim` object. render with `gganimate::animate(.)` or `gganimate::anim_save("file.gif", .)`.

Author(s)

cassy dorff, shahryar minhas

Examples

```
data(classroom_edges)
classroom_panel <- rbind(
    transform(classroom_edges[1:12, ], wave = 1),
    transform(classroom_edges[13:24, ], wave = 2)
)

longit_net <- netify(
    classroom_panel,
    actor1 = "from", actor2 = "to", time = "wave",
    symmetric = TRUE,
    missing_to_zero = TRUE
)

anim <- animate_netify(longit_net)
```

---

as.igraph.netify *as.igraph method for netify objects*

Description

s3 method that lets igraph's `as.igraph` generic dispatch on netify objects. equivalent to `netify_to_igraph(x, ...)`. registered against the **igraph** namespace in `.onload`, so the dispatch works regardless of whether **igraph** is attached before or after **netify**.

Usage

```
as.igraph.netify(x, ...)
```

Arguments

`x`	a netify object.
`...`	extra arguments forwarded to `netify_to_igraph` (e.g. `add_nodal_attribs`, `add_dyad_attribs`).

Value

an igraph object, or a list of igraph objects (longitudinal / multilayer), as produced by `netify_to_igraph`.

Author(s)

shahryar minhas

`as.matrix.netify` *coerce a netify object to a plain matrix*

Description

strips the netify class and netify-specific attributes so the result is a clean numeric matrix carrying only `dim` and `dimnames`. for longitudinal netify objects, `time` selects which slice to return; it defaults to the first time period and emits a hint. round-trips (`net |> as.matrix() |> netify()`) recover a fresh cross-sectional netify object, but structural attributes (symmetric / diag_to_NA / weight) are re-detected from the matrix on the way back in rather than copied across, so a directed matrix or one with a non-na diagonal will be flagged accordingly.

Usage

```
## S3 method for class 'netify'
as.matrix(x, time = NULL, layer = NULL, ...)
```

Arguments

`x`	a netify object.
`time`	for longitudinal netify objects, either the integer index or character label of the time slice to extract. defaults to the first slice and emits a hint when used implicitly.
`layer`	for multilayer netify objects, either the integer index or character label of the layer to extract. defaults to the first layer and emits a hint when used implicitly.
`...`	additional args (ignored).

Value

a plain numeric matrix with `dim` and `dimnames` only (no `netify` class, no netify metadata attributes).

Author(s)

cassy dorff, shahryar minhas

See Also

`get_adjacency` for the data.frame-input counterpart that also accepts a netify object; `netify` for rebuilding a netify object from a plain matrix.

Examples

```
data(icews)
icews_2010 <- icews[icews$year == 2010, ]
net <- netify(icews_2010, actor1 = "i", actor2 = "j",
    symmetric = FALSE, weight = "verbCoop")
m <- as.matrix(net)
```

```
dim(m)
class(m)
```

`as.network.netify` *as.network method for netify objects*

Description

s3 method that lets statnet's `as.network` generic dispatch on netify objects. equivalent to `netify_to_statnet(x, ...)`. registered against the **network** namespace in `.onload`, so the dispatch works regardless of whether **network** is attached before or after **netify**.

Usage

```
as.network.netify(x, ...)
```

Arguments

`x`	a netify object.
`...`	extra arguments forwarded to `netify_to_statnet` (e.g. `add_nodal_attribs`, `add_dyad_attribs`).

Value

a statnet `network` object or list of network objects, as produced by `netify_to_statnet`.

Author(s)

shahryar minhas

`as_tibble.netify` *convert a netify object to a tibble (long edge frame)*

Description

s3 method for `tibble::as_tibble()`. returns the same long-format frame as `unnetify()` / `tidy.netify()`, wrapped in a tibble for tidyverse-pipe friendliness.

Usage

```
as_tibble.netify(x, ...)
```

Arguments

`x`	a netify object.
`...`	passed to `unnetify()` (e.g., `remove_zeros = TRUE`).

Details

registered against the `tibble::as_tibble` generic via `.onload`, so `tibble` is not a hard dependency. when `tibble` isn't installed, a plain data.frame is returned.

Value

a tibble (or data.frame if tibble isn't installed) with one row per dyad. includes `from`, `to`, optional `time`/`layer`, the edge weight, dyadic covariates, and nodal covariates merged in with `_from` / `_to` suffixes.

Author(s)

cassy dorff, shahryar minhas

See Also

`tidy.netify()` for the broom-style sibling and `unnetify()` for the underlying converter.

Examples

```
data(icews)
icews_10 <- icews[icews$year == 2010, ]
net <- netify(icews_10, actor1 = "i", actor2 = "j",
    symmetric = FALSE, weight = "verbCoop")
tibble::as_tibble(net)
```

---

as_tibble.netify_comparison

Convert a netify_comparison to a tibble

Description

s3 method for `tibble::as_tibble()` that returns the `$comparisons` data frame directly. the raw `netify_comparison` object is a list with mixed scalar / nested fields, which `tibble::as_tibble()` cannot coerce cleanly. the per-pair comparison table is almost always what tidyverse users want downstream (filter / arrange / pivot / join with metadata).

Usage

```
as_tibble.netify_comparison(x, ...)
```

Arguments

`x`	a `netify_comparison` object from `compare_networks()`.
`...`	currently unused.

Details

registered against the `tibble::as_tibble` generic via `.onload`, so `tibble` is not a hard dependency.

Value

a tibble of pairwise comparisons (one row per (`net_i`, `net_j`, `metric`) triple). if the comparison object has no `$comparisons` slot (e.g., a single-network input), an empty tibble is returned with a one-shot inform.

Author(s)

cassy dorff, shahryar minhas

See Also

`compare_networks()`, `as_tibble.netify()`.

`assemble_netify_plot` *assemble netify plot from components*

Description

assembles a complete network plot from netify plot components. this function automatically adds all available layers (edges, nodes, text, labels, and their repel versions) in the correct order with appropriate scale resets between layers.

Usage

```
assemble_netify_plot(comp)
```

Arguments

`comp`	a netify_plot_components object returned from `plot(..., return_components = TRUE)`

Details

this function provides a convenient way to reassemble a plot from its components after extracting them with `return_components = TRUE`. it automatically:

- adds layers in the correct order (edges, nodes, text/text_repel, labels/label_repel)
- inserts scale resets between layers when necessary
- handles both standard and repel versions of text and label layers
- includes facets and themes if present

Value

a complete ggplot object ready for display or further customization

Author(s)

cassy dorff, shahryar minhas

See Also

plot.netify, netify_edge, netify_node

Examples

```
# create a netify object
mat <- matrix(c(NA, 1, 0, 0, NA, 1, 1, 0, NA), 3, 3,
    dimnames = list(c("alice", "bob", "carol"),
                    c("alice", "bob", "carol")))
net <- new_netify(mat, symmetric = FALSE)

# get plot components
comp <- plot(net, return_components = TRUE)

# reassemble the plot
p <- assemble_netify_plot(comp)
print(p)
```

---

attribute_report *Summary of network-attribute relationships*

Description

summarizes how nodal and dyadic attributes relate to network structure. combines homophily analysis, mixing patterns, dyadic correlations, and network position-based attribute summaries.

Usage

```
attribute_report(
  netlet,
  node_vars = NULL,
  dyad_vars = NULL,
  include_centrality = TRUE,
  include_homophily = TRUE,
  include_mixing = TRUE,
  include_dyadic_correlations = TRUE,
  centrality_measures = c("degree", "betweenness"),
  categorical_threshold = 10,
```

```
  significance_test = TRUE,
  other_stats = NULL,
  ...
)
```

Arguments

`netlet`	a netify object containing network data.
`node_vars`	character vector of nodal attributes to analyze. if NULL, analyzes all available nodal variables except actor and time.
`dyad_vars`	character vector of dyadic attributes to analyze. if NULL, analyzes all available dyadic variables.
`include_centrality`	logical. whether to calculate attribute-centrality relationships. default TRUE.
`include_homophily`	logical. whether to perform homophily analysis. default TRUE.
`include_mixing`	logical. whether to create mixing matrices for categorical attributes. default TRUE.
`include_dyadic_correlations`	logical. whether to calculate dyadic correlations. default TRUE.
`centrality_measures`	character vector of centrality measures to calculate. options: "degree", "betweenness", "closeness", "eigenvector". default c("degree", "betweenness").
`categorical_threshold`	maximum number of unique values for categorical treatment. default 10.
`significance_test`	logical. whether to perform significance tests. default TRUE.
`other_stats`	named list of custom functions for additional statistics.
`...`	additional arguments passed to component functions.

Details

wraps the exploratory analysis functions and chooses methods based on attribute types. for large networks or many attributes, consider setting some components to FALSE for faster computation. centrality measures use igraph functions.

Value

list containing:

`homophily_analysis` results from homophily analysis for nodal attributes

`mixing_analysis` results from mixing matrix analysis for categorical attributes

`dyadic_correlations` results from dyadic correlation analysis

`centrality_correlations` correlations between nodal attributes and centrality

`attribute_summaries` descriptive statistics for attributes

`overall_summary` brief summary of key findings

Author(s)

cassy dorff, shahryar minhas

`binarize` *Binarize a netify object at a threshold*

Description

thin wrapper around `mutate_weights()` for the very common case of "dichotomize the weighted network at a cut-point." returns a new netify with edge values coerced to 0/1 based on the supplied threshold (or threshold function).

Usage

```
binarize(netlet, threshold = 0, strict = FALSE, abs = FALSE, new_name = NULL)
```

Arguments

`netlet`	a weighted netify object.
`threshold`	numeric scalar (default `0` – any nonzero edge becomes 1) or a function `f(x)` that takes the vector of edge weights and returns a single numeric threshold (e.g., `function(x) median(x, na.rm = TRUE)` or `function(x) quantile(x, 0.75, na.rm = TRUE)`).
`strict`	logical. if `TRUE`, edges with weight strictly greater than the threshold become 1; if `FALSE` (default), the threshold itself is included (>=). for `threshold = 0`, the default gives the "any nonzero edge counts" semantics that matches the rest of the package (signed-weight density, homophily-default-threshold, etc.).
`abs`	logical. if `TRUE`, compare `|x|` to the threshold so that negative-magnitude ties also count toward the binarization. defaults to `FALSE`. when the network contains both positive and negative weights and `abs = FALSE`, `binarize()` informs once that negative ties will be dropped.
`new_name`	optional character. new name for the binarized weight column (default keeps the original name).

Details

na cells (e.g., the diagonal under `diag_to_NA = TRUE`) propagate as na in the output rather than becoming 0. use `na.rm = TRUE` when summing edges if you want them treated as 0. structural zeros stay zero in every branch – a negative `threshold` will not promote empty cells to 1, regardless of `strict` or `abs`.

Value

a binarized netify object with `is_binary = TRUE`.

Author(s)

cassy dorff, shahryar minhas

See Also

`mutate_weights()` for arbitrary transformations.

Examples

```
data(icews)
net <- netify(icews[icews$year == 2010, ],
actor1 = "i", actor2 = "j", symmetric = FALSE, weight = "verbCoop")
# any-nonzero-edge dichotomization
bin0 <- binarize(net)
# 75th-percentile of nonzero weights
bin75 <- binarize(net, threshold = function(x) {
nz <- x[x > 0]
quantile(nz, 0.75, na.rm = TRUE)
})
```

---

bind_netifies *Combine multiple netify objects*

Description

`bind_netifies()` concatenates two or more netify objects along the time axis (for combining cross-sec -> longit, or stacking two longit panels into a longer one). all inputs must share the same mode (`unipartite` / `bipartite`), symmetry, layers, and (for cross-sec inputs) the same actor set. actor sets across periods may differ – the result is a `longit_list`.

Usage

```
bind_netifies(
  ...,
  names = NULL,
  align_actors = c("none", "union", "intersection")
)
```

Arguments

`...`	two or more netify objects, or a single list of netify objects.
`names`	optional character vector to name the resulting periods. if `NULL`, periods are auto-named from the inputs' existing period labels (with deduplication if collisions).
`align_actors`	one of `"none"` (default), `"union"`, or `"intersection"`. controls how per-period actor sets are reconciled when inputs differ:

- `"none"`: keep each period's actor set as supplied; resulting `longit_list` periods may have different dimensions (matches prior behavior).
- `"union"`: take the union of actor sets across all inputs and pad each period with na rows/columns for actors not originally present.
- `"intersection"`: take the intersection of actor sets across all inputs and subset each period to only those actors.

Details

for combining different *layers* of the same time slice, use `layer_netify()`.

this is **not** the same as `layer_netify()`:

- `bind_netifies()` joins along time.
- `layer_netify()` joins along relation (layer).

nodal and dyadic attributes are concatenated per-period; if two inputs supply conflicting values for the same (actor, time), the later input wins (with a one-shot inform).

when downstream models (e.g., `tergm` cmle) require uniform actor composition across periods, use `align_actors = "union"` or `"intersection"`.

multilayer inputs are supported when each input has the same layer labels. the result is a longitudinal list whose per-period elements keep the layer dimension, so `as.matrix(x, time = ..., layer = ...)`, `unnetify()`, and `decompose_netify()` can still address layers.

Value

a `longit_list` netify object.

Author(s)

cassy dorff, shahryar minhas

Examples

```
data(icews)
n1 <- netify(icews[icews$year == 2010, ],
actor1 = "i", actor2 = "j", symmetric = FALSE,
weight = "verbCoop")
n2 <- netify(icews[icews$year == 2011, ],
actor1 = "i", actor2 = "j", symmetric = FALSE,
weight = "verbCoop")
combined <- bind_netifies(n1, n2, names = c("2010", "2011"))
summary(combined)
```

`bootstrap_netlet` *Bootstrap any user-supplied function of a netify object*

Description

resamples actors with replacement (snijders & borgatti 1999 vertex bootstrap) per panel, rebuilds the netlet on each draw, applies the user-supplied `fn`, and returns the per-draw values plus percentile confidence intervals.

Usage

```
bootstrap_netlet(
  netlet,
  fn,
  n_boot = 200L,
  alpha = 0.05,
  seed = NULL,
  verbose = TRUE
)
```

Arguments

`netlet`	a `netify` object.
`fn`	function. takes a netlet, returns a single numeric value or a named numeric vector. called once per bootstrap draw.
`n_boot`	integer. number of bootstrap replicates (default `200`).
`alpha`	numeric in (0, 1). two-sided percentile-ci alpha (default `0.05` -> 95% intervals).
`seed`	optional integer. if supplied, sets a local rng seed and restores the user's global stream afterward. if `NULL`, bootstrap draws use and advance the current rng stream normally.
`verbose`	logical. if `TRUE` (default), print a progress ticker every 50 draws.

Details

this is the general bootstrap interface for any scalar or named numeric vector summary of a netify. it is useful for homophily, mixing-matrix, centrality, and downstream fit summaries (e.g., the coefficient of a `to_igraph` -> `igraph::cluster_*` -> modularity pipeline).

resampling is **vertex-level**, not edge-level: actors are sampled with replacement and the netlet's adjacency is sliced accordingly. for longitudinal netlets, the same resampled index set is applied to every period (preserves within-actor dependence). multilayer netlets are unsupported – bootstrap each layer independently via `subset_netify(layers = ...)` first. the resampled netlets contain the resampled adjacency only. statistics that require attached nodal or dyadic attributes should recreate those attributes inside `fn`, or use a NULL-model workflow such as `compare_to_null(..., model = "dyad_permutation")` when the goal is an attribute-aware randomization.

Value

a data.frame with one row per element of fn(netlet) and columns:

metric name of the output element (or "value" for scalar fn).

point point estimate from fn(netlet) on the original (un-resampled) netlet.

n_valid number of non-na bootstrap draws used for that metric.

mean bootstrap mean.

sd bootstrap standard deviation.

lower, upper lower / upper percentile ci bounds.

the full per-draw matrix is stashed as attr(out, "bootstrap_draws") for callers who want the empirical distribution (e.g., for kernel-density plots).

parallel execution

bootstrap_netlet runs serially. when n_boot > 50 and the netlet is large enough that each draw is non-trivial (rule of thumb: n > ~1000 with the default centrality-heavy fn), parallelism helps. there is no built-in parallel = argument; instead drive the loop yourself with future.apply::future_lapply(), which respects whatever future::plan() the caller set:

```
library(future); library(future.apply)
plan(multisession)        # or multicore on linux/macos
draws <- future_lapply(1:n_boot, function(b) {
  idx <- sample(actors, length(actors), replace = TRUE)
  fn(subset_netify(net, actors = idx))
}, future.seed = TRUE)
```

then summarize draws exactly as bootstrap_netlet does internally. wrapping the inner body of bootstrap_netlet in a future-aware loop is a small refactor and is the recommended path for ~15k-node weekly snapshots where the serial pass would tie up a single core for hours.

Author(s)

cassy dorff, shahryar minhas

Examples

```
data(icews)
net <- netify(icews[icews$year == 2010, ],
actor1 = "i", actor2 = "j", symmetric = FALSE, weight = "verbCoop")
# bootstrap a custom scalar: mean(closeness)
my_stat <- function(net) {
sa <- summary_actor(net, stats = "all")
c(mean_closeness = mean(sa$closeness_all, na.rm = TRUE))
}
boot_out <- bootstrap_netlet(net, fn = my_stat, n_boot = 100, seed = 1)
boot_out
```

`classroom_edges` *synthetic high-school friendship edgelist*

Description

a small synthetic edgelist of reported friendships among 30 students (see `classroom_nodes`). ties are **undirected** – each row records that two students named each other as friends.

Usage

```
data(classroom_edges)
```

Format

a data frame with about 50 rows and 2 columns:

`from` student identifier of one friend, character.

`to` student identifier of the other friend, character.

Details

the edgelist is synthetic and contains one row per friendship (not two). when you build a netify object with `symmetric = TRUE` (the default for undirected ties), the constructor automatically fills in both directions.

Author(s)

cassy dorff, shahryar minhas

See Also

`classroom_nodes`, `netify`, `netify_workflows`.

Examples

```
data(classroom_edges)
data(classroom_nodes)
head(classroom_edges)

# build a friendship network with student attributes attached.
net <- netify(
    classroom_edges,
    actor1 = "from", actor2 = "to",
    symmetric = TRUE,
    nodal_data = classroom_nodes
)
summary(net)
```

`classroom_nodes` *Synthetic high-school friendship roster (nodes)*

Description

a small synthetic roster of 30 students intended for examples and teaching. designed to support a typical survey-style workflow: one row per student, one column per attribute, plus a separate edgelist of reported friendship ties (`classroom_edges`).

Usage

```
data(classroom_nodes)
```

Format

a data frame with 30 rows and 4 columns:

`student` student identifier, character (e.g. `"s01"` .. `"s30"`). use this as the actor column when attaching attributes via `add_node_vars`.

`gender` reported gender, character, `"f"` or `"m"`.

`grade` grade level, integer 9-12.

`gpa` grade point average on the 0-4 scale, numeric.

Details

this dataset is **synthetic** – generated to illustrate how netify handles standard student/peer survey data. it is not drawn from any real classroom. ties tend to form within the same grade and (more weakly) within the same gender, so attribute-based analyses such as `homophily` and `mixing_matrix` produce meaningful (non-NULL) patterns.

pair with `classroom_edges` (an undirected friendship edgelist on the same 30 students).

Author(s)

cassy dorff, shahryar minhas

See Also

`classroom_edges`, `netify`, `netify_workflows`.

Examples

```
data(classroom_nodes)
head(classroom_nodes)
table(classroom_nodes$gender, classroom_nodes$grade)
```

compare_networks *Compare networks across time, layers, or attributes*

Description

compares networks to identify similarities and differences. supports comparison of edge patterns, structural properties, and node compositions across multiple networks or within subgroups.

Usage

```
compare_networks(
  nets,
  method = "correlation",
  by = NULL,
  what = "edges",
  test = TRUE,
  n_permutations = 5000,
  include_diagonal = FALSE,
  return_details = FALSE,
  edge_threshold = 0,
  permutation_type = c("classic", "degree_preserving"),
  correlation_type = c("pearson", "spearman"),
  binary_metric = c("phi", "simple_matching", "mean_centered"),
  seed = NULL,
  p_adjust = c("none", "holm", "BH", "BY"),
  adaptive_stop = FALSE,
  alpha = 0.05,
  max_permutations = 20000,
  spectral_rank = 0,
  attr_metric = c("ecdf_cor", "wasserstein"),
  other_stats = NULL
)
```

Arguments

nets	either a list of netify objects to compare, or a single netify object (for longitudinal, multilayer, or by-group comparisons).
method	character string specifying comparison method:
	"correlation" pearson correlation of edge weights (default)
	"jaccard" jaccard similarity for binary networks
	"hamming" hamming distance (proportion of differing edges)
	"qap" quadratic assignment procedure with permutation test
	"spectral" spectral distance based on eigenvalue spectra. measures global structural differences by comparing the sorted eigenvalues of network laplacian matrices. useful for detecting fundamental structural changes.
	"all" applies all applicable methods

`by` character vector of nodal attributes. if specified, compares networks within and between attribute groups rather than across input networks. (note: currently not fully implemented)

`what` character string specifying what to compare:

"edges" edge-level comparison (default)

"structure" structural properties (density, clustering, etc.)

"nodes" node composition and attributes

"attributes" compare networks based on node attribute distributions

`test` logical. whether to perform qap significance testing for edge comparisons when `method = "qap"` or `method = "all"`. other comparison metrics are returned as descriptive summaries. default TRUE.

`n_permutations` integer. number of permutations for qap test. default 5000.

`include_diagonal`
logical. whether to include diagonal in comparison. default FALSE.

`return_details` logical. whether to return detailed comparison matrices. default FALSE.

`edge_threshold` numeric or function. for weighted networks, threshold to determine edge presence in jaccard, hamming, and edge-change summaries. default is 0 (positive weights are treated as present ties).

`permutation_type`
character string specifying permutation scheme:

"classic" standard label permutation qap (default)

"degree_preserving" preserves the first network's degree sequence for binary directed or symmetric networks

`correlation_type`
character string. "pearson" (default) or "spearman" for rank-based edge correlation. qap significance testing currently uses pearson.

`binary_metric` character string for binary network correlation:

"phi" standard phi coefficient (default)

"simple_matching" proportion of matching edges

"mean_centered" mean-centered phi coefficient

`seed` integer. random seed for reproducible permutations. default NULL.

`p_adjust` character string for multiple testing correction: "none" (default), "holm", "bh" (benjamini-hochberg), or "by".

`adaptive_stop` logical. whether to use adaptive stopping for permutations. default FALSE. (currently not implemented)

`alpha` numeric. significance level for adaptive stopping. default 0.05.

`max_permutations`
integer. maximum permutations for adaptive stopping. default 20000.

`spectral_rank` integer. number of eigenvalues to use for spectral distance. default 0 (use all). set to a smaller value (e.g., 50-100) for large networks to improve performance while maintaining accuracy.

`attr_metric` character string for continuous attribute comparison:

"ecdf_cor" correlation of empirical cdfs (default)

"wasserstein" wasserstein-1 (earth mover's) distance

`other_stats` named list of custom functions to calculate additional comparison statistics. each function should accept a netify object (or matrix for edge comparisons) and return a named vector of scalar values. the specific input depends on the `what` parameter:

for `what = "edges"` functions receive adjacency matrices

for `what = "structure"` functions receive netify objects

for `what = "nodes"` functions receive netify objects

for `what = "attributes"` functions receive netify objects

example: `list(connectivity = function(net) { g <- to_igraph(net); c(vertex_conn = igraph::vertex_connectivity(g)) })`

Details

the function supports four types of comparisons:

edge comparison (what = "edges"): compares edge patterns between networks using correlation, jaccard similarity, hamming distance, spectral distance, or qap permutation tests. returns detailed edge changes showing which edges are added, removed, or maintained between networks.

structure comparison (what = "structure"): compares network-level properties like density, reciprocity, transitivity, and mean degree. for two networks, also provides percent change calculations.

node comparison (what = "nodes"): analyzes changes in actor composition between networks, tracking which actors enter or exit the network.

attribute comparison (what = "attributes"): compares distributions of node attributes across networks using appropriate statistical tests (ks test for continuous, total variation distance for categorical).

automatic handling of longitudinal data: when passed a single longitudinal netify object, the function automatically extracts time periods and performs pairwise comparisons between them.

automatic handling of multilayer networks: when passed a single multilayer netify object (created with `layer_netify()`), the function automatically extracts layers and performs pairwise comparisons between them. this works for cross-sectional multilayer (3d arrays), longitudinal multilayer (4d arrays), and longitudinal list multilayer formats.

summary statistics interpretation:

- correlation: ranges from -1 to 1, measuring linear relationship between edge weights
- jaccard: ranges from 0 to 1, proportion of edges present in both networks
- hamming: ranges from 0 to 1, proportion of differing edges
- qap p-value: observed pearson edge correlation compared to the selected permutation reference distribution

permutation methods: when `permutation_type = "degree_preserving"`, the first network must be binary (0/1). directed inputs use directed edge swaps that preserve in- and out-degree. symmetric inputs use undirected edge swaps that preserve the undirected degree sequence.

Value

a list of class "netify_comparison" containing:

comparison_type character string: "cross_network", "temporal", "multilayer", or "by_group"

method comparison method(s) used

n_networks number of networks compared

summary data frame with comparison statistics

edge_changes list detailing added, removed, and maintained edges (for edge comparisons)

node_changes list detailing added, removed, and maintained nodes (for node comparisons)

significance_tests qap test results when edge qap testing was run

details detailed comparison matrices if return_details = TRUE

comparisons long-format data frame for `what = "edges"` with one row per (network pair, metric) triple. columns: `net_i`, `net_j` (the two network names), `metric` (e.g. `"correlation"`, `"jaccard"`, `"hamming"`, `"qap_correlation"`, `"spectral"`, `"weight_correlation"`), `value` (scalar metric), `p_value` (only populated for `qap_correlation`; `na` otherwise). coerce to a tibble with `tibble::as_tibble(comp)` (the `as_tibble` s3 method returns this frame directly).

Author(s)

cassy dorff, shahryar minhas

Examples

```
data(classroom_edges)
data(classroom_nodes)

first_half <- classroom_edges[1:35, ]
second_half <- classroom_edges[17:51, ]

net_a <- netify(
    first_half,
    actor1 = "from", actor2 = "to",
    symmetric = TRUE,
    nodal_data = classroom_nodes,
    missing_to_zero = TRUE
)

net_b <- netify(
    second_half,
    actor1 = "from", actor2 = "to",
    symmetric = TRUE,
    nodal_data = classroom_nodes,
    missing_to_zero = TRUE
)

comp <- compare_networks(list("first" = net_a, "second" = net_b))
print(comp)
```

```
struct_comp <- compare_networks(
    list("first" = net_a, "second" = net_b),
    what = "structure"
)

classroom_panel <- rbind(
    transform(first_half, wave = 1),
    transform(second_half, wave = 2)
)

longit_net <- netify(
    classroom_panel,
    actor1 = "from", actor2 = "to",
    time = "wave",
    symmetric = TRUE,
    output_format = "longit_list",
    missing_to_zero = TRUE
)

temporal_comp <- compare_networks(longit_net, method = "hamming")

detailed_comp <- compare_networks(
    list("first" = net_a, "second" = net_b),
    return_details = TRUE
)

names(detailed_comp$details) # shows available matrices

# compare with custom statistics

library(igraph)

# define custom connectivity function
connectivity_stats <- function(net) {
    g <- to_igraph(net)
    c(vertex_connectivity = vertex_connectivity(g),
      edge_connectivity = edge_connectivity(g),
      diameter = diameter(g, directed = FALSE))
}

# apply to structural comparison
struct_comp_custom <- compare_networks(
    list("first" = net_a, "second" = net_b),
    what = "structure",
    other_stats = list(connectivity = connectivity_stats)
)

# custom stats will appear in the summary
print(struct_comp_custom$summary)
```

`compare_to_null` *Test an observed network statistic against a NULL distribution*

Description

simulates `n_sim` networks from a NULL model (default erdos-renyi at the observed density), applies the user-supplied statistic to each, and reports how the observed statistic compares – point estimate, percentile of the NULL distribution, and two-sided monte carlo p-value against the selected NULL model.

Usage

```
compare_to_null(
  netlet,
  fn,
  n_sim = 200L,
  model = c("erdos_renyi", "configuration", "dyad_permutation"),
  alpha = 0.05,
  seed = NULL,
  verbose = TRUE
)
```

Arguments

`netlet`	the observed netify object.
`fn`	function. takes a **netify object** (not an igraph or matrix) and returns a single numeric value or a named numeric vector. called once on the observed netlet and `n_sim` times on the simulated draws. wrap igraph helpers with `to_igraph()` inside `fn`, e.g. `fn = function(net) igraph::transitivity(to_igraph(net))`.
`n_sim`	integer. number of NULL-model draws (default `200`).
`model`	character. NULL-model family; passed to `simulate.netify()`. one of `"erdos_renyi"` (default), `"configuration"`, `"dyad_permutation"`.

picking a NULL model. for ergm-trained users:

`erdos_renyi` random graph at observed density. use when the NULL hypothesis is "no structure beyond density."

`configuration` random graph preserving the observed degree sequence. use when you want to ask "is my observed structure surprising *given* the degree distribution?" – already conditions on degree.

`dyad_permutation` vertex-label permutation. preserves the full degree distribution and weight values; use when comparing attribute-aware structure (homophily, mixing) against actor relabelings.

for weighted netlets, the simulator draws weights from the empirical non-zero weight distribution so observed-vs-NULL stats are computed on comparable scales (`dyad_permutation` is exact – it preserves weights via relabel).

alpha	numeric in (0, 1). two-sided alpha for the percentile ci of the NULL distribution (default 0.05 -> 95%).
seed	optional integer. if supplied, sets a local rng seed and restores the user's global stream afterward. if NULL, NULL draws use and advance the current rng stream normally.
verbose	logical. progress ticker every 50 draws.

Details

combines simulate.netify() + a vectorized fn loop into the single-call entry users actually want: "is my observed transitivity surprising vs. a random graph?"

Value

a data.frame (also class "netify_null_test") with one row per stat and columns:

metric name of the statistic.

observed observed value on netlet.

n_valid number of non-na simulated draws used for that metric.

null_mean, null_sd moments of the NULL distribution.

null_lower, null_upper percentile ci bounds.

p_value two-sided monte carlo p-value using the simulated NULL draws.

extreme logical: is observed outside [null_lower, null_upper]?

parallel execution

compare_to_null runs serially. at n_sim > 50 with large n, the per-draw fn() cost (igraph community detection, full summary_actor) dominates wall-clock. there is no built-in parallel = argument; drive the loop manually with future.apply::future_lapply():

```
library(future); library(future.apply)
plan(multisession)
sims <- simulate(net, nsim = n_sim, model = "erdos_renyi", seed = 1)
null_draws <- future_lapply(sims, fn, future.seed = TRUE)
```

summarize null_draws as compare_to_null does internally (rowmeans, quantiles, two-sided p). for 15k-node weekly snapshots with n_sim = 500, a 4-core multisession plan cuts wall-clock by roughly 3.5x.

Author(s)

cassy dorff, shahryar minhas

See Also

simulate.netify() for the NULL-model engine, bootstrap_netlet() for sampling-variation cis around the observed stat itself.

Examples

```
data(icews)
net <- netify(icews[icews$year == 2010, ],
actor1 = "i", actor2 = "j", symmetric = FALSE, weight = "verbCoop")
my_stat <- function(net) {
s = suppressMessages(summary(net))
c(transitivity = s$transitivity, reciprocity = s$reciprocity)
}
compare_to_null(net, my_stat, n_sim = 20, seed = 1)
```

---

decompose_igraph *decompose an igraph object into base r components*

Description

decompose_igraph extracts the adjacency matrix and any vertex/edge attributes from an igraph object, returning them in a standardized list format.

Usage

```
decompose_igraph(grph, weight = NULL)
```

Arguments

grph	an igraph object to be decomposed.
weight	character string specifying the edge attribute to use as weights in the adjacency matrix. if NULL (default), the unweighted adjacency matrix is returned with 1s for edges and 0s for non-edges.

Details

the function handles both unipartite and bipartite graphs appropriately:

graph type detection:

- bipartite graphs must have a logical 'type' vertex attribute
- if the 'type' attribute exists but is not logical, the graph is treated as unipartite with a warning

vertex naming:

if the graph lacks vertex names, default names are assigned:

- unipartite graphs: "a1", "a2", ..., "an"
- bipartite graphs: "r1", "r2", ... for type 1; "c1", "c2", ... for type 2

existing vertex names are always preserved and used in the output.

matrix extraction:

- unipartite: uses `as_adjacency_matrix()` to get $n \times n$ matrix
- bipartite: uses `as_biadjacency_matrix()` to get $n_1 \times n_2$ matrix where rows correspond to type=FALSE vertices and columns to type=TRUE vertices

attribute handling:

all vertex and edge attributes are preserved in the output data frames. system attributes (like 'name' and 'type') are included alongside user-defined attributes.

Value

a list containing four elements:

- **adj_mat**: the adjacency matrix extracted from the igraph object
  - for unipartite graphs: square matrix of dimension $n \times n$
  - for bipartite graphs: rectangular matrix of dimension $n_1 \times n_2$
  - values are edge weights if specified, otherwise 0/1
- **ndata**: a data frame of vertex attributes, or NULL if none exist
  - always includes an 'actor' column with vertex names
  - additional columns for each vertex attribute
- **ddata**: a data frame of edge attributes, or NULL if none exist
  - columns 'from' and 'to' specify edge endpoints
  - additional columns for each edge attribute
- **weight**: the edge attribute name used for weights, if provided

Note

for longitudinal networks with changing actor composition, explicitly set vertex names before decomposition to ensure consistent actor identification across time periods.

the adjacency matrix is always returned as a standard r matrix (not sparse), which may have memory implications for very large graphs.

when edge attributes are used as weights, ensure they contain numeric values. non-numeric edge attributes will cause an error.

Author(s)

cassy dorff, shahryar minhas

`decompose_netify` *Decompose a netify object into edge and node data frames*

Description

`decompose_netify` separates a netify object into its constituent parts: a data frame of edges with attributes and a data frame of nodal attributes.

Usage

```
decompose_netify(netlet, remove_zeros = TRUE)
```

Arguments

`netlet`	a netify object (class "netify") to be decomposed.
`remove_zeros`	logical. if TRUE (default), edges with zero weight values are removed from the edge data frame. if FALSE, zero-weight edges are retained.

Details

the function helpful for:

edge data processing:

- extracts the adjacency matrix (or array for longitudinal networks) from the netify object
- optionally removes zero-weight edges based on the remove_zeros parameter
- merges any dyadic variables stored in the netify object
- renames columns to standardized names (from, to, time)

node data processing:

- extracts nodal attributes if present, or constructs from actor_pds information
- ensures consistent time variable across node and edge data
- renames columns to standardized names (name, time)

time handling:

- for cross-sectional networks: sets time to "1" in both data frames
- for longitudinal networks: preserves original time periods as character values
- for ego networks: extracts time from ego-time concatenated identifiers

variable preservation:

all dyadic and nodal variables stored in the netify object are preserved in the output data frames. dyadic variables are merged with the edge data, while nodal variables remain in the nodal data frame.

Value

a list containing two data frames:

- **edge_data**: a data frame where each row represents an edge with columns:
  - `from`: source node identifier
  - `to`: target node identifier
  - `time`: time period (character; "1" for cross-sectional networks)
  - `weight`: edge weight values (using original weight variable name if specified)
  - additional columns for any dyadic variables stored in the netify object
- **nodal_data**: a data frame where each row represents a node-time combination with columns:
  - `name`: node identifier
  - `time`: time period (character; "1" for cross-sectional networks)
  - additional columns for any nodal variables stored in the netify object

Author(s)

cassy dorff, shahryar minhas

Examples

```
# load example data
data(icews)

# example 1: cross-sectional network
icews_10 <- icews[icews$year == 2010, ]

# create netify object
net_cs <- netify(
    icews_10,
    actor1 = "i", actor2 = "j",
    symmetric = FALSE,
    weight = "verbCoop"
)

# decompose to data frames
decomposed_cs <- decompose_netify(net_cs)

# examine structure
str(decomposed_cs)
head(decomposed_cs$edge_data)
head(decomposed_cs$nodal_data)

# example 2: keep zero-weight edges
decomposed_with_zeros <- decompose_netify(net_cs, remove_zeros = FALSE)

# compare edge counts
nrow(decomposed_cs$edge_data) # without zeros
nrow(decomposed_with_zeros$edge_data) # with zeros
```

```
# example 4: use for visualization prep

# decompose for use with ggplot2
plot_data <- decompose_netify(net_cs)

# can now use edge_data and nodal_data separately
# for network visualization
```

`decompose_statnet` *decompose a network object into base r components*

Description

`decompose_statnet` (also available as `decompose_network`) extracts the adjacency matrix and any vertex/edge attributes from a network object (from the statnet suite), returning them in a standardized list format.

Usage

```
decompose_statnet(ntwk, weight = NULL)

decompose_network(ntwk, weight = NULL)
```

Arguments

`ntwk`	a network object (class "network") to be decomposed.
`weight`	character string specifying the edge attribute to use as weights in the adjacency matrix. if NULL (default), the unweighted adjacency matrix is returned with 1s for edges and 0s for non-edges.

Details

the function handles both unipartite and bipartite networks appropriately:

network type detection:

- bipartite networks are identified using `is.bipartite()`
- the bipartite partition size is retrieved from the 'bipartite' network attribute

vertex naming:

the function checks for existing vertex names in the 'vertex.names' attribute. if names are just the default numeric sequence (1, 2, 3, ...), they are treated as missing. default names are assigned when needed:

- unipartite networks: "a1", "a2", ..., "an"
- bipartite networks: "r1", "r2", ... for first partition; "c1", "c2", ... for second partition

matrix extraction:

unlike igraph's bipartite handling, the network package returns the full adjacency matrix even for bipartite networks. the function:

- extracts the full matrix using `as.matrix.network.adjacency()`
- for bipartite networks, the matrix has dimension $(n_1 + n_2) \times (n_1 + n_2)$ with the first n_1 rows/columns for the first partition

attribute handling:

system attributes ('vertex.names' and 'na') are excluded from the vertex attribute data frame. all user-defined vertex and edge attributes are preserved.

Value

a list containing four elements:

- **adj_mat**: the adjacency matrix extracted from the network object
  - for unipartite networks: square matrix of dimension $n \times n$
  - for bipartite networks: full square matrix of dimension $(n_1 + n_2) \times (n_1 + n_2)$
  - values are edge weights if specified, otherwise 0/1
- **ndata**: a data frame of vertex attributes, or NULL if none exist
  - always includes an 'actor' column with vertex names
  - additional columns for each vertex attribute (excluding system attributes)
- **ddata**: a data frame of edge attributes, or NULL if none exist
  - columns 'from' and 'to' specify edge endpoints using vertex names
  - additional columns for each edge attribute
- **weight**: the edge attribute name used for weights, if provided

Note

for longitudinal networks with changing actor composition, explicitly set vertex names before decomposition to ensure consistent actor identification across time periods.

the adjacency matrix format differs between this function and `decompose_igraph` for bipartite networks: this function returns the full square matrix while `decompose_igraph` returns only the rectangular bipartite portion.

edge directions are preserved in the adjacency matrix according to the network's directed/undirected property.

Author(s)

cassy dorff, shahryar minhas

drop_na_actors *Drop actors with NA covariates from a netify object*

Description

removes actors whose nodal_data carries na in one or more covariate columns. ergm terms like nodecov() and nodematch() reject na-bearing vertex attributes, so this helper is handy upstream of netify_to_statnet. works for cross-sectional, longitudinal, and bipartite netlets.

Usage

```
drop_na_actors(netlet, cols = NULL)
```

Arguments

netlet	a netify object with a nodal_data attribute.
cols	character vector of column names in nodal_data to check for na. NULL (the default) checks every non-bookkeeping column (everything except actor, time, and layer).

Details

for longitudinal netlets an actor is dropped from every period if any of its rows in nodal_data carry na in the inspected columns. for bipartite netlets only actors in the mode that the nodal table covers are filtered; the other mode passes through untouched.

corner cases:

- if cols references a name that is not in nodal_data, the call aborts with a clear message listing the missing columns.
- if no actor carries na in the inspected columns, the input netlet is returned unchanged (no inform).
- if *every* actor carries na (so the cleaned netlet would have zero actors), the call aborts rather than silently returning an empty netify, which would break downstream to_statnet() / ergm() pipelines.
- if the netlet has no nodal_data attribute attached, the input is returned unchanged.

Value

a netify object equivalent to subset_netify(netlet, actors = clean_actors) after dropping any actor whose nodal rows contain na in the inspected columns. if no nas are found the input is returned unchanged.

Author(s)

shahryar minhas

Examples

```
data(icews)
net <- netify(
    icews,
    actor1 = "i", actor2 = "j", time = "year",
    symmetric = FALSE, weight = "verbCoop",
    nodal_vars = c("i_polity2", "i_log_gdp", "i_log_pop")
)
clean <- drop_na_actors(net, cols = c("i_polity2", "i_log_gdp"))
```

---

dyad_correlation *Analyze correlations between dyadic attributes and network ties*

Description

examines relationships between dyadic (pairwise) attributes and network connections. calculates correlations between dyadic variables and edge weights/presence, with support for multiple correlation methods and significance testing.

Usage

```
dyad_correlation(
  netlet,
  dyad_vars = NULL,
  edge_vars = NULL,
  method = "pearson",
  binary_network = FALSE,
  remove_diagonal = TRUE,
  significance_test = TRUE,
  alpha = 0.05,
  partial_correlations = FALSE,
  other_stats = NULL,
  ...
)
```

Arguments

netlet	a netify object containing network data.
dyad_vars	character vector of dyadic attribute names to analyze. if NULL, analyzes all available dyadic variables.
edge_vars	character vector of edge variables to correlate with. if NULL, uses the main network matrix.
method	character string specifying correlation method:
	"pearson" pearson product-moment correlation (default)

	"spearman" spearman rank correlation
	"kendall" kendall's tau correlation
`binary_network`	logical. whether to convert ties to binary before correlation. default FALSE.
`remove_diagonal`	logical. whether to exclude diagonal elements. default TRUE.
`significance_test`	logical. whether to calculate ordinary correlation p-values and confidence intervals on the dyad vectors. default TRUE.
`alpha`	significance level for confidence intervals. default 0.05.
`partial_correlations`	logical. whether to calculate partial correlations controlling for other dyadic variables. default FALSE.
`other_stats`	named list of custom functions for additional statistics.
`...`	additional arguments passed to custom functions.

Details

extracts dyadic variables from dyad_data attribute and correlates them with network ties. for longitudinal networks, correlations are calculated separately for each time period. dyadic variables should be stored as matrices with rows and columns corresponding to network actors. missing values are handled using pairwise complete observations.

the reported p-values and confidence intervals are the standard tests from `stats::cor.test()` applied to the dyad vectors. they are useful as descriptive screens, but they do not model network dependence among dyads.

Value

data frame with one row per dyadic variable per network/time period:

`net` network/time identifier

`layer` layer name

`dyad_var` name of dyadic variable

`edge_var` name of edge variable

`correlation` correlation coefficient

`p_value` p-value for correlation significance

`ci_lower`, `ci_upper` confidence interval bounds

`n_pairs` number of dyad pairs included

`method` correlation method used

`mean_dyad_var` mean value of dyadic variable

`sd_dyad_var` standard deviation of dyadic variable

`mean_edge_var` mean value of edge variable

`sd_edge_var` standard deviation of edge variable

Author(s)

cassy dorff, shahryar minhas

ego_layouts *Create ego-centric layouts for ego networks*

Description

these functions create specialized layouts for ego networks that place the ego at the center and arrange alters in meaningful ways around it.

Usage

```
create_hierarchical_ego_layout(
  netlet,
  min_radius = 2,
  max_radius = 5,
  seed = 123
)

create_radial_ego_layout(
  netlet,
  ego_name = NULL,
  n_rings = 4,
  min_radius = 1.5,
  max_radius = 5,
  seed = 123
)

create_ego_centric_layout(
  netlet,
  buffer_radius = 1.5,
  transition_zone = 0.5,
  seed = 123
)
```

Arguments

netlet	a netify object created with ego_netify()
min_radius	minimum distance from ego to any alter
max_radius	maximum distance from ego to any alter
seed	random seed for reproducible layouts
ego_name	name of the ego node (extracted from netlet attributes if not provided)
n_rings	number of concentric rings for radial layout
buffer_radius	minimum distance from ego for ego_centric layout
transition_zone	smooth transition zone width for ego_centric layout

Details

three layout algorithms are provided:

hierarchical layout: places ego at center and arranges alters in concentric circles based on their network centrality. more central alters are placed closer to the ego.

radial layout: places ego at center and arranges alters in concentric rings based on connection strength quartiles. stronger connections are placed in inner rings.

ego centric layout: uses force-directed layout but ensures ego remains at center with a buffer zone. preserves natural clustering while maintaining ego visibility.

Value

a list of data frames with x,y coordinates for each time period

Author(s)

cassy dorff, shahryar minhas

Examples

```
# create ego network
mat <- matrix(c(NA, 1, 1, 1, NA, 0, 1, 0, NA), 3, 3,
    dimnames = list(c("alice", "bob", "carol"),
                    c("alice", "bob", "carol")))
net <- new_netify(mat, symmetric = FALSE)
ego_net <- ego_netify(net, ego = "alice")

# hierarchical layout
layout <- create_hierarchical_ego_layout(ego_net)
plot(ego_net, point_layout = layout)

# radial layout with custom rings
layout <- create_radial_ego_layout(ego_net, n_rings = 5)
plot(ego_net, point_layout = layout)
```

---

ego_netify *Create ego network from a netify object*

Description

`ego_netify` extracts an ego network from a netify object. an ego network consists of a focal node (ego) and its immediate neighbors (alters). for weighted networks, users can define neighborhoods using edge weight thresholds. the function returns a netify object representing the ego network.

Usage

```
ego_netify(
  netlet,
  ego,
  threshold = NULL,
  ngbd_direction = "any",
  include_ego = TRUE
)
```

Arguments

netlet	a netify object (class "netify") from which to extract the ego network.
ego	character string specifying the name of the ego for whom to create the ego network. must match an actor name in the netify object.
threshold	numeric value or vector specifying the threshold for including alters in the ego network based on edge weights. for longitudinal networks, can be a vector with length equal to the number of time periods to apply different thresholds over time. if NULL (default), uses 0 for unweighted networks and the mean edge weight for weighted networks.
ngbd_direction	character string specifying which neighbors to include for directed networks. options are: • "out": include alters that ego has outgoing ties to • "in": include alters that ego has incoming ties from • "any": include alters with any tie to/from ego (default)
include_ego	logical. if TRUE (default), the ego node is included in the ego network. if FALSE, only alters are included.

Details

the function extracts an ego network by identifying all nodes connected to the specified ego based on the given criteria:

neighborhood definition:

- for unweighted networks: all nodes with edges to/from ego (threshold = 0)
- for weighted networks: all nodes with edge weights exceeding the threshold
- direction matters only for directed networks (controlled by ngbd_direction)

threshold behavior:

- if not specified, defaults to 0 for unweighted networks
- if not specified for weighted networks, uses the mean edge weight
- for longitudinal networks, can vary by time period if a vector is provided
- edges with weights > threshold are included (not >=)

output structure:

the function preserves all attributes from the original netify object, including nodal and dyadic variables, but subsets them to include only ego and its neighbors. for longitudinal networks, ego networks may vary in composition across time periods as relationships change.

limitations:

- currently does not support multilayer networks
- currently does not support bipartite networks

Value

a netify object representing the ego network. for longitudinal networks, returns a list of netify objects with one ego network per time period.

each returned netify object includes additional attributes:

- `ego_netify`: TRUE (indicator that this is an ego network)
- `ego_id`: identifier of the ego
- `threshold`: threshold value(s) used
- `ngbd_direction`: direction specification used
- `include_ego`: whether ego was included

Note

to create ego networks for multiple egos, use `lapply` or a loop to call this function for each ego separately.

Author(s)

cassy dorff, shahryar minhas

Examples

```
# cross-sectional ego network from the bundled classroom data
data(classroom_edges)
data(classroom_nodes)
net <- netify(
    classroom_edges,
    actor1 = "from", actor2 = "to",
    symmetric = TRUE,
    nodal_data = classroom_nodes
)
s07_ego <- ego_netify(net, ego = "s07")
print(s07_ego)


# longitudinal ego network with a weighted, directed netlet
data(icews)
netlet <- netify(
    icews,
```

```
    actor1 = "i", actor2 = "j", time = "year",
    weight = "verbCoop"
)
pakistan_ego <- ego_netify(netlet, ego = "Pakistan")
summary(pakistan_ego)
```

`from_lame_fit` *Convert a fitted* `lame`/`amen` *AME object back into a netify*

Description

takes the posterior-mean (or point-estimate) prediction matrix from a `lame::ame()` / `lame::ame_als()` / `amen::ame()` fit and wraps it as a netify so the predictions can be summarized, plotted, or compared to the observed netlet via `compare_networks()`.

Usage

```
from_lame_fit(
  fit,
  value = c("fitted", "residual", "prob", "prob_lower", "prob_upper", "fitted_lower",
    "fitted_upper"),
  symmetric = NULL,
  alpha = 0.05
)
```

Arguments

`fit`	a fitted object from `lame::ame()`, `lame::ame_als()`, `lame::lame()`, or `amen::ame()`. must expose a fitted-value matrix (e.g., `fit$ez`, `fit$zpostmean`, or `fitted(fit)`). for `value` in `"prob_lower"` / `"prob_upper"` / `"fitted_lower"` / `"fitted_upper"` the fit must additionally expose a per-draw array of fitted values. the slots searched, in order, are: `fit$boot$ez` and `fit$boot$y_hat` (lame als parametric / block bootstrap), then `fit$ez_draws`, `fit$ezps`, and `fit$z_draws` (gibbs posterior draws), with shape `[n, n, b]` or `[n, n, t, b]`.
`value`	one of `"fitted"` (default – posterior-mean linear predictor ez/zpostmean), `"residual"` (observed - fitted), `"prob"` (logistic / probit -> probability scale, when the family supports it), or `"prob_lower"` / `"prob_upper"` / `"fitted_lower"` / `"fitted_upper"` (per-cell `alpha`/`1-alpha` quantiles across bootstrap or posterior draws). when `lame` exposes a `fitted()` method for the object, that's preferred. for `value = "prob"` with a binary family, link detection follows this priority: (1) any explicit `link` slot on the fit (or `fit$control$link`); (2) als class (any token containing "als") or `fit$fit_method = "als"` -> probit, matching `lame::ame_als()`; (3) `lame` class -> logit, matching `lame::lame()` gibbs default; (4) `ame` / `amen` class (and no `fit_method = "logit"` override) -> probit, matching `amen::ame()` gibbs convention; (5) otherwise logit fallback.

symmetric	logical. override the inferred symmetry; default reads from the fit's stored mode/y.
alpha	numeric in (0, 0.5). tail probability for the *_lower / *_upper quantiles. default 0.05 -> 90% interval (lower = 0.025, upper = 0.975 when interpreted as a two-sided ci; here we use lower = alpha/2, upper = 1 - alpha/2).

Details

useful for posterior-predictive checks: build the observed netlet, run an ame fit, then from_lame_fit(fit) |> plot(style = "heatmap") to visualize the fitted intensity matrix.

Value

a cross-sectional netify object whose underlying matrix is the fitted-value (or residual) matrix at the same actor ordering.

Author(s)

cassy dorff, shahryar minhas

Examples

```
fitted_values <- matrix(
    c(-1.2, 0.4, 0.8, -0.5),
    nrow = 2,
    dimnames = list(c("alice", "bob"), c("club_a", "club_b"))
)

fit <- list(
    EZ = fitted_values,
    family = "binary",
    link = "logit",
    mode = "bipartite"
)

pred_net <- from_lame_fit(fit, value = "prob")
plot(pred_net, style = "heatmap")
```

---

gen_symm_id *generate symmetric identifiers for dyadic data*

Description

gen_symm_id creates symmetric identifiers for dyadic data, ensuring that each unique pair of actors receives the same id regardless of order. this is particularly useful for undirected network data where the relationship between actors a and b is identical to the relationship between b and a.

Usage

```
gen_symm_id(dyad_data, actor1, actor2, time = NULL)
```

Arguments

dyad_data	a data.frame containing dyadic data with at least two columns representing actors in each dyad. will be coerced to data.frame if a tibble or data.table is provided.
actor1	character string specifying the column name for the first actor in each dyad.
actor2	character string specifying the column name for the second actor in each dyad.
time	character string specifying the column name for time periods. if provided, the time value will be appended to the symmetric id to create unique identifiers for each time period. set to NULL (default) for cross-sectional data.

Details

the function ensures symmetry by alphabetically sorting actor names before creating the identifier. this guarantees that:

- the dyad "usa-china" receives the same id as "china-usa"
- the dyad "brazil-argentina" receives the same id as "argentina-brazil"
- actor pairs are consistently ordered regardless of input order

when a time column is specified, it's appended to the symmetric id to maintain unique identifiers across different time periods. this allows for proper aggregation of longitudinal dyadic data while preserving temporal variation.

Value

a character vector of symmetric identifiers with the same length as the number of rows in dyad_data. each id is formatted as:

- **without time**: a length-prefixed key for the alphabetically sorted actor pair
- **with time**: the same key with the time value appended

Note

all actor values are converted to character strings before creating ids to ensure consistent sorting behavior across different data types.

ids are intended as opaque keys. do not parse them to recover actor names; keep the original actor columns when those values are needed downstream.

this function is primarily used internally by `aggregate_dyad` for efficient symmetric aggregation, but can be used independently for creating symmetric dyad identifiers.

Author(s)

shahryar minhas

Examples

```
# create example dyadic data
trade_df <- data.frame(
    from = c("usa", "china", "russia", "usa", "brazil", "argentina"),
    to = c("china", "usa", "usa", "russia", "argentina", "brazil"),
    trade_value = c(100, 100, 50, 75, 30, 25),
    year = c(2020, 2020, 2021, 2021, 2021, 2021)
)

# generate symmetric ids without time
trade_df$symm_id <- gen_symm_id(trade_df, "from", "to")
print(trade_df[, c("from", "to", "symm_id")])
# note: usa-china and china-usa both get "china_usa"

# generate symmetric ids with time
trade_df$symm_id_time <- gen_symm_id(trade_df, "from", "to", "year")
print(trade_df[, c("from", "to", "year", "symm_id_time")])
# note: usa-china in 2020 gets "china_usa_2020"

# use for aggregation of undirected relationships
trade_df$total_trade <- ave(
    trade_df$trade_value,
    trade_df$symm_id_time,
    FUN = sum
)
print(unique(trade_df[, c("symm_id_time", "total_trade")]))

# example with longitudinal data
library(netify)
data(icews)
icews_sample <- icews[1:100, ]

# create symmetric ids for conflict events
icews_sample$symm_dyad <- gen_symm_id(
    icews_sample,
    actor1 = "i",
    actor2 = "j",
    time = "year"
)

# check that symmetric pairs get same id
icews_sample[icews_sample$i == "united states" & icews_sample$j == "israel", "symm_dyad"]
icews_sample[icews_sample$i == "israel" & icews_sample$j == "united states", "symm_dyad"]
```

---

get_actor_time_info *Extract actor time range information*

Description

`get_actor_time_info` returns a per-actor data.frame of entry and exit times. it dispatches on the first argument:

Usage

```
get_actor_time_info(x, ...)

## S3 method for class 'netify'
get_actor_time_info(x, ...)

## S3 method for class 'data.frame'
get_actor_time_info(x, actor1, actor2, time, ...)

## Default S3 method:
get_actor_time_info(x, actor1, actor2, time, ...)
```

Arguments

`x`	a netify object, or a data.frame of dyadic observations.
`...`	unused; reserved for future methods.
`actor1`	character string specifying the column name for the first actor in each dyad (data.frame method only).
`actor2`	character string specifying the column name for the second actor in each dyad (data.frame method only).
`time`	character string specifying the column name for time periods (data.frame method only).

Details

- if `x` is a **netify object**, it returns the stored `actor_pds` attribute directly (one row per actor with `min_time` / `max_time`). this is the open-cohort roster the netlet was built with – and the roster every per-period statistic (density, degree, homophily) is computed against.
- if `x` is a **data.frame** of dyadic observations, it computes the entry / exit times from the data. entry is defined as the first time period in which an actor appears in any interaction (as either sender or receiver), and exit as the last time period. use this form to prepare the `actor_pds` argument to `netify()`.

use cases:

- on a **dyad data.frame**: build the `actor_pds` argument to `netify(..., actor_time_uniform = FALSE, actor_pds = ...)` for open-cohort panels (panel surveys with attrition, contact-tracing chains, organizational membership over time, etc.).
- on a **netify object**: inspect the entry / exit roster the netlet is currently using – useful when debugging density denominators, writing custom exporters, or verifying that an open-cohort netlet has the actor windows you expect.

assumptions (data.frame method):

- an actor is considered "present" in any time period where they appear in the data, regardless of role (sender/receiver).
- missing values in time are ignored when calculating min/max.
- actors must appear in at least one non-missing time period.

Value

a data.frame with three columns:

- **actor**: character vector of unique actor identifiers.
- **min_time**: earliest time period the actor is in the network (entry point).
- **max_time**: latest time period the actor is in the network (exit point).

for the netify method, this is a verbatim copy of `attr(x, "actor_pds")`. for the data.frame method, actors are ordered as they appear in the aggregation, not alphabetically or by time.

Note

the data.frame method assumes that presence in the data indicates network participation. if actors can be temporarily absent from the network while still being considered members, this method will not capture such gaps – supply an explicit `actor_pds` roster to `netify()` instead.

Author(s)

shahryar minhas, ha eun choi

cassy dorff, shahryar minhas

Examples

```
# data.frame input: derive the roster
df <- data.frame(
    i = c("a", "a", "b", "c"),
    j = c("b", "c", "c", "a"),
    t = c(1, 2, 2, 3)
)
get_actor_time_info(df, "i", "j", "t")

# netify input: read back the stored roster

roster <- data.frame(actor = c("a", "b"), min_time = c(1, 1), max_time = c(3, 4))
net <- netify(df, actor1 = "i", actor2 = "j", time = "t",
              actor_time_uniform = FALSE, actor_pds = roster)
get_actor_time_info(net)
```

get_adjacency *Create a netify matrix from cross-sectional dyadic data*

Description

get_adjacency converts cross-sectional dyadic data into an adjacency matrix of class "netify". this function creates a single network matrix representing relationships at one point in time.

Usage

```
get_adjacency(
  dyad_data,
  actor1 = NULL,
  actor2 = NULL,
  symmetric = TRUE,
  mode = "unipartite",
  weight = NULL,
  sum_dyads = FALSE,
  diag_to_NA = TRUE,
  missing_to_zero = TRUE,
  nodelist = NULL
)
```

Arguments

dyad_data	a data.frame containing dyadic observations. will be coerced to data.frame if a tibble or data.table is provided. as a convenience, an existing netify object is also accepted: in that case `get_adjacency()` returns the underlying adjacency as a plain matrix (no class / attributes), and for longitudinal inputs returns the first time slice with a cli hint pointing to `as.matrix(net, time = ...)` for selecting a specific slice.
actor1	character string specifying the column name for the first actor in each dyad.
actor2	character string specifying the column name for the second actor in each dyad.
symmetric	logical. if TRUE (default), treats the network as undirected (i.e., edges have no direction). if FALSE, treats the network as directed.
mode	character string specifying network structure. options are: • `"unipartite"`: one set of actors (default) • `"bipartite"`: two distinct sets of actors
weight	character string specifying the column name containing edge weights. if NULL (default), edges are treated as unweighted (binary).
sum_dyads	logical. if TRUE, sums weight values when multiple edges exist between the same actor pair. if FALSE (default), uses the last observed value.
diag_to_NA	logical. if TRUE (default), sets diagonal values (self-loops) to na. automatically set to FALSE for bipartite networks.

`missing_to_zero`
: logical. if TRUE (default), treats missing edges as zeros. if FALSE, missing edges remain as na.

`nodelist`
: character vector of actor names to include in the network. if provided, ensures all listed actors appear in the network even if they have no edges (isolates). useful when working with edgelists that only contain active dyads.

Details

note on usage:

while this function is exported and available for direct use, the primary and recommended way to create netify objects from dyadic data is through the `netify()` function. the `netify()` function:

- provides a consistent interface for both cross-sectional and longitudinal data
- includes additional data validation and preprocessing options
- can incorporate nodal and dyadic attributes during creation
- checks parameters before constructing the matrix

use `get_adjacency()` directly only when you need a simple adjacency matrix creation without additional features.

Value

a matrix of class "netify" (a netify matrix) with:

- **dimensions**: `[n_actors x n_actors]` for unipartite networks or `[n_actors1 x n_actors2]` for bipartite networks
- **class**: "netify" - this is a full netify object compatible with all netify functions
- **attributes**: metadata including network properties and processing parameters

the returned object is a netify matrix that can be used with all netify functions such as `summary()`, `plot()`, `to_igraph()`, etc.

Author(s)

ha eun choi, cassy dorff, colin henry, shahryar minhas

Examples

```
# load example data
data(icews)

# subset to one year for cross-sectional analysis
icews_2010 <- icews[icews$year == 2010, ]

# create a directed network with verbal cooperation weights
verbCoop_net <- get_adjacency(
    dyad_data = icews_2010,
    actor1 = "i",
    actor2 = "j",
```

```
    symmetric = FALSE,
    weight = "verbCoop"
)

# create a directed network with material conflict weights
matlConf_net <- get_adjacency(
    dyad_data = icews_2010,
    actor1 = "i",
    actor2 = "j",
    symmetric = FALSE,
    weight = "matlConf"
)

# verify class
class(verbCoop_net) # "netify"

# check dimensions
dim(verbCoop_net)
```

`get_adjacency_array` *Create a netify array from longitudinal dyadic data*

Description

`get_adjacency_array` converts longitudinal dyadic data into a three-dimensional netify array where the first two dimensions represent actors and the third dimension represents time periods. this function creates an array of class "netify" and should only be used when actor composition remains constant across all time periods.

Usage

```
get_adjacency_array(
  dyad_data,
  actor1 = NULL,
  actor2 = NULL,
  time = NULL,
  symmetric = TRUE,
  mode = "unipartite",
  weight = NULL,
  sum_dyads = FALSE,
  diag_to_NA = TRUE,
  missing_to_zero = TRUE,
  nodelist = NULL
)
```

Arguments

dyad_data	a data.frame containing longitudinal dyadic observations. will be coerced to data.frame if a tibble or data.table is provided.
actor1	character string specifying the column name for the first actor in each dyad.
actor2	character string specifying the column name for the second actor in each dyad.
time	character string specifying the column name for time periods.
symmetric	logical. if TRUE (default), treats the network as undirected (i.e., edges have no direction). if FALSE, treats the network as directed.
mode	character string specifying network structure. options are: • "unipartite": one set of actors (default) • "bipartite": two distinct sets of actors
weight	character string specifying the column name containing edge weights. if NULL (default), edges are treated as unweighted (binary).
sum_dyads	logical. if TRUE, sums weight values when multiple edges exist between the same actor pair in the same time period. if FALSE (default), uses the last observed value.
diag_to_NA	logical. if TRUE (default), sets diagonal values (self-loops) to na. automatically set to FALSE for bipartite networks.
missing_to_zero	logical. if TRUE (default), treats missing edges as zeros. if FALSE, missing edges remain as na.
nodelist	character vector of actor names to include in the network. if provided, ensures all listed actors appear in the network even if they have no edges (isolates). useful when working with edgelists that only contain active dyads.

Details

note on usage:

while this function is exported and available for direct use, the primary and recommended way to create netify arrays from longitudinal dyadic data is through the `netify()` function. the `netify()` function:

- automatically determines whether to create an array or list structure
- handles time-varying actor composition
- validates inputs before constructing arrays
- offers a unified interface for all types of network data

use `get_adjacency_array()` directly only when you specifically need low-level control over array creation and are certain your actors remain constant across time.

Value

a three-dimensional array of class "netify" (a netify array) with:

- **dimensions**: `[n_actors x n_actors x n_time]` for unipartite networks or `[n_actors1 x n_actors2 x n_time]` for bipartite networks
- **class**: "netify" - this is a full netify object compatible with all netify functions
- **attributes**: extensive metadata including network properties, actor information, and processing parameters

the returned object is a netify array that can be used with all netify functions such as `summary()`, `plot()`, `to_igraph()`, etc.

Author(s)

cassy dorff, ha eun choi, shahryar minhas

Examples

```
# load example data
data(icews)

# create a netify array (longitudinal directed network)
# with material conflict as edge weights
icews_array <- get_adjacency_array(
    dyad_data = icews,
    actor1 = "i",
    actor2 = "j",
    time = "year",
    symmetric = FALSE,
    weight = "matlConf"
)

# verify it's a netify object
class(icews_array) # "netify"

# check dimensions
dim(icews_array) # [n_actors, n_actors, n_years]

# access specific time period
icews_2010 <- icews_array[, , "2010"]
```

---

get_adjacency_list *Create a netify list from longitudinal dyadic data*

Description

`get_adjacency_list` converts longitudinal dyadic data into a list of adjacency matrices of class "netify". this function creates a list structure where each element is a network matrix for a specific time period, allowing for time-varying actor composition.

Usage

```
get_adjacency_list(
  dyad_data,
  actor1 = NULL,
  actor2 = NULL,
  time = NULL,
  symmetric = TRUE,
  mode = "unipartite",
  weight = NULL,
  sum_dyads = FALSE,
  actor_time_uniform = FALSE,
  actor_pds = NULL,
  diag_to_NA = TRUE,
  missing_to_zero = TRUE,
  nodelist = NULL
)
```

Arguments

dyad_data	a data.frame containing longitudinal dyadic observations. will be coerced to data.frame if a tibble or data.table is provided.
actor1	character string specifying the column name for the first actor in each dyad.
actor2	character string specifying the column name for the second actor in each dyad.
time	character string specifying the column name for time periods.
symmetric	logical. if TRUE (default), treats the network as undirected (i.e., edges have no direction). if FALSE, treats the network as directed.
mode	character string specifying network structure. options are: • "unipartite": one set of actors (default) • "bipartite": two distinct sets of actors
weight	character string specifying the column name containing edge weights. if NULL (default), edges are treated as unweighted (binary).
sum_dyads	logical. if TRUE, sums weight values when multiple edges exist between the same actor pair in the same time period. if FALSE (default), uses the last observed value.
actor_time_uniform	logical indicating how to handle actor composition: • TRUE: assumes all actors exist across the entire time range • FALSE: determines actor existence from the data - actors exist from their first observed interaction to their last
actor_pds	optional data.frame specifying when actors enter and exit the network. must contain columns 'actor', 'min_time', and 'max_time'. can be created using get_actor_time_info(). if provided, overrides actor_time_uniform.
diag_to_NA	logical. if TRUE (default), sets diagonal values (self-loops) to na. automatically set to FALSE for bipartite networks.

`missing_to_zero`
: logical. if TRUE (default), treats missing edges as zeros. if FALSE, missing edges remain as na.

`nodelist`
: character vector of actor names to include in the network. if provided, ensures all listed actors appear in the network even if they have no edges (isolates). useful when working with edgelists that only contain active dyads.

Details

note on usage:

while this function is exported and available for direct use, the primary and recommended way to create netify objects from longitudinal dyadic data is through the `netify()` function. the `netify()` function:

- automatically chooses between array and list representations based on your data
- validates inputs before constructing matrices
- can incorporate nodal and dyadic attributes during creation
- offers a unified interface for all types of network data

use `get_adjacency_list()` directly only when you specifically need a list structure or require low-level control over the creation process.

actor composition handling:

this function is particularly useful when actors enter and exit the network over time. unlike `get_adjacency_array()`, which requires constant actor composition, this function can handle:

- new actors appearing in later time periods
- actors exiting and no longer appearing in the data
- different sets of actors active in each time period

Value

a list of class "netify" (a netify list) with:

- **elements**: named list where each element is a netify matrix for one time period
- **names**: character representation of time periods
- **class**: "netify" - this is a full netify object compatible with all netify functions
- **attributes**: extensive metadata including network properties, actor composition information, and processing parameters

each matrix in the list may have different dimensions if actor composition varies over time. the returned object can be used with all netify functions such as `summary()`, `plot()`, `to_igraph()`, etc.

Author(s)

cassy dorff, ha eun choi, shahryar minhas

Examples

```
# load example data
data(icews)

# create a netify list with constant actor composition
icews_list <- get_adjacency_list(
    dyad_data = icews,
    actor1 = "i",
    actor2 = "j",
    time = "year",
    actor_time_uniform = TRUE,
    symmetric = FALSE,
    weight = "verbConf"
)

# verify it's a netify object
class(icews_list) # "netify"

# check structure
length(icews_list) # number of time periods
names(icews_list) # time period labels

# access specific time period
icews_2010 <- icews_list[["2010"]]
dim(icews_2010)
```

---

get_edge_layout *Generate edge layout coordinates for netify visualization*

Description

get_edge_layout prepares edge data for network visualization by calculating start and end coordinates for line segments representing edges. this function maps edges from a netify object to their corresponding node positions as determined by a layout algorithm.

Usage

```
get_edge_layout(netlet, nodes_layout, ig_netlet = NULL)
```

Arguments

netlet	a netify object (class "netify") containing the network structure from which edges will be extracted.
nodes_layout	a data.frame or matrix containing node positions, or a list of such objects for longitudinal networks. each element must include columns: • **actor**: character string identifying each node • **x**: numeric x-coordinate of the node position

- **y**: numeric y-coordinate of the node position

for longitudinal networks, provide a named list where:

- names correspond to time periods in the netify object
- each element follows the structure described above
- time period names must match those in the netify object

`ig_netlet` an optional pre-converted igraph object. if provided, this function will use it directly instead of converting the netify object again.

Details

this function performs the following operations:

edge extraction:

- converts the netify object to igraph format internally
- extracts the edge list preserving edge directions
- handles both cross-sectional and longitudinal networks

coordinate mapping:

- matches each edge endpoint to its corresponding node position
- creates a complete set of coordinates for drawing edges
- preserves the temporal structure for longitudinal networks

use in visualization:

this function is typically used as part of a visualization pipeline:

1. create node layout using `get_node_layout()` or a custom layout algorithm
2. generate edge coordinates using this function
3. pass both to visualization functions for plotting

Value

depending on the input netify object:

- **cross-sectional**: a list containing one data.frame with columns:
  - `from`: source node name
  - `to`: target node name
  - `x1`, `y1`: coordinates of the source node
  - `x2`, `y2`: coordinates of the target node
- **longitudinal**: a named list of data.frames (one per time period) with the same structure as above

the output maintains the same temporal structure as the input netify object.

Note

the nodes_layout structure must exactly match the actors and time periods in the netify object. missing actors in the layout will result in na coordinates for their associated edges.

for longitudinal networks, ensure that the names of the nodes_layout list match the time period labels in the netify object (e.g., "2008", "2009").

this function always returns a list structure for consistency, even for cross-sectional networks where the list contains only one element.

Author(s)

cassy dorff, shahryar minhas

`get_ego_layout` *Calculate ego-centric layout positions for network visualization*

Description

`get_ego_layout` computes node positions for ego network visualization using ego-centric layout algorithms. these layouts place the ego at the center and arrange alters around it (radially, by concentric ring, etc.) so that the ego's relationships are organized around the focal node.

Usage

```
get_ego_layout(
  netlet,
  layout = "star",
  group_by = NULL,
  order_by = NULL,
  weight_to_distance = FALSE,
  ring_gap = 0.3,
  ego_size = 0.1,
  seed = 6886
)
```

Arguments

`netlet`	a netify object (class "netify") that is an ego network, created using `ego_netify`. the object must have ego_netify = TRUE attribute.
`layout`	character string specifying the ego-centric layout algorithm. options: • `"radial"`: places ego at center with alters arranged in a circle around it. distance from center can encode relationship strength. • `"concentric"`: places ego at center with alters in concentric circles based on a grouping variable or relationship strength. • `"star"`: simple star layout with ego at center and alters equally spaced around it (default).

`group_by` character string specifying a nodal attribute to use for grouping alters in the layout. for "radial" layout, groups are arranged in sectors. for "concentric" layout, groups determine which ring alters appear in. if NULL (default), no grouping is applied.

`order_by` character string specifying a nodal attribute to use for ordering alters within their groups or around the ego. common options include network statistics like "degree_total" or custom attributes. if NULL (default), alters are arranged alphabetically.

`weight_to_distance` logical. if TRUE and the network is weighted, use edge weights to determine distance from ego (higher weights = closer). for "radial" layout only. default is FALSE.

`ring_gap` numeric value between 0 and 1 specifying the gap between concentric rings as a proportion of the total radius. only used for "concentric" layout. default is 0.3.

`ego_size` numeric value specifying the relative size of the central area reserved for the ego. larger values create more space between ego and alters. default is 0.1.

`seed` integer for random number generation to ensure reproducible layouts when there are ties in ordering. default is 6886.

Details

this function provides specialized layouts for ego networks that emphasize the ego-alter structure:

layout algorithms:

- **radial**: places the ego at the origin and arranges alters in a circle around it. if grouping is specified, alters are arranged in sectors with related nodes near each other. if weight_to_distance is TRUE, alters with stronger ties to ego are placed closer to the center.
- **concentric**: places the ego at the origin and arranges alters in concentric circles. the ring assignment can be based on a grouping variable (categorical) or a continuous variable (discretized into rings).
- **star**: a simple star layout that places ego at center and distributes alters evenly around a single circle. this is equivalent to the radial layout without grouping or weighting.

visual encoding:

the layouts allow encoding of network properties through spatial arrangement:

- **distance from ego**: can represent tie strength, frequency of interaction, or other dyadic measures
- **angular position**: can group similar alters together (e.g., family, friends, colleagues)
- **ring assignment**: can represent categories, levels of importance, or discretized continuous variables

longitudinal networks:

for longitudinal ego networks, the function maintains consistent angular positions for alters across time periods when possible, making it easier to track changes in the ego's network over time.

Value

a list of data frames (one per time period) where each data frame contains:

- **index**: integer node index
- **actor**: character string with actor name
- **x**: numeric x-coordinate for node position
- **y**: numeric y-coordinate for node position

for cross-sectional networks, returns a list with one element. for longitudinal networks, returns a named list with time periods as names.

Note

this function is designed specifically for ego networks created with `ego_netify`. for general network layouts, use `get_node_layout`.

the function will issue a warning if used on non-ego networks but will attempt to proceed by treating the first node as the ego.

Author(s)

cassy dorff, shahryar minhas

Examples

```
# create an ego network
mat <- matrix(c(NA, 1, 1, 1, NA, 0, 1, 0, NA), 3, 3,
    dimnames = list(c("alice", "bob", "carol"),
                    c("alice", "bob", "carol")))
net <- new_netify(mat, symmetric = FALSE)
nodes <- data.frame(
    actor = c("alice", "bob", "carol"),
    department = c("lead", "field", "field"),
    degree_total = c(2, 1, 1)
)
net <- add_node_vars(net, nodes, actor = "actor")
ego_net <- ego_netify(net, ego = "alice")

# get radial layout with alters grouped by attribute
layout_radial <- get_ego_layout(ego_net,
                                layout = "radial",
                                group_by = "department")

# get concentric layout with rings based on degree
layout_circles <- get_ego_layout(ego_net,
                                 layout = "concentric",
                                 group_by = "degree_total")

# use with plot
plot(ego_net, point_layout = layout_radial)
```

get_node_layout *Calculate node layout positions for netify visualization*

Description

get_node_layout computes node positions for network visualization using various layout algorithms from igraph. this function converts a netify object to igraph format, applies the specified layout algorithm, and returns node coordinates suitable for plotting.

Usage

```
get_node_layout(
  netlet,
  layout = NULL,
  static_actor_positions = FALSE,
  which_static = NULL,
  seed = 6886,
  ig_netlet = NULL,
  ...
)
```

Arguments

netlet	a netify object (class "netify") for which to compute layout positions.
layout	character string specifying the layout algorithm to use. options include: • "nicely": automatic selection of appropriate layout (default for unipartite) • "bipartite": two-column layout for bipartite networks (default for bipartite) • "fruchtermanreingold" or "fr": force-directed layout • "kamadakawai" or "kk": another force-directed layout • "circle": nodes arranged in a circle • "star": star-shaped layout • "grid": nodes on a grid • "tree": hierarchical tree layout • "random" or "randomly": random positions • additional options: "graphopt", "sugiyama", "drl", "lgl", "dh", "gem", "mds" if NULL, defaults to "nicely" for unipartite or "bipartite" for bipartite networks.
static_actor_positions	logical. if TRUE, maintains consistent node positions across all time periods in longitudinal networks. if FALSE (default), each time period gets its own optimized layout.

`which_static`	integer specifying which time period's layout to use as the static template when static_actor_positions is TRUE. if NULL (default), creates a static layout based on the union of all edges across time periods, giving more weight to persistent edges.
`seed`	integer for random number generation to ensure reproducible layouts. default is 6886.
`ig_netlet`	an optional pre-converted igraph object. if provided, this function will use it directly instead of converting the netify object again.
`...`	additional arguments passed to ego-specific layout functions when layout is "radial" or "concentric". see `get_ego_layout` for available options (ego_group_by, ego_order_by, ego_weight_to_distance, etc.).

Details

this function handles layout generation for both cross-sectional and longitudinal networks with several key features:

layout algorithms:

the function provides access to all major igraph layout algorithms. the default "nicely" option automatically selects an appropriate algorithm based on the network structure. for ego networks, specialized layouts ("radial" and "concentric") are available that emphasize the ego-alter structure.

longitudinal layouts:

for longitudinal networks, two approaches are available:

- **dynamic layouts**: each time period gets its own optimized layout, which may better reveal structural changes but makes visual comparison harder
- **static layouts**: all time periods use the same node positions, facilitating visual comparison of network evolution

when using static layouts with which_static = NULL, the function creates a composite layout based on the union of all edges across time periods. edges that appear more frequently are given higher weight, producing layouts that emphasize the stable core structure of the network.

bipartite networks:

for bipartite networks, the default layout arranges the two node sets in separate columns.

Value

a list of data frames (one per time period) where each data frame contains:

- **index**: integer node index
- **actor**: character string with actor name
- **x**: numeric x-coordinate for node position
- **y**: numeric y-coordinate for node position

for cross-sectional networks, returns a list with one element. for longitudinal networks, returns a named list with time periods as names.

Author(s)

cassy dorff, shahryar minhas

`get_raw` *get raw network data without netify attributes*

Description

`get_raw` extracts the underlying network data structure (matrix or list) from a netify object, removing all netify-specific attributes. this is useful when you need to work with the base r data structures or pass the network data to functions that don't recognize netify objects.

Usage

```
get_raw(netlet)
```

Arguments

`netlet`	a netify object

Value

a matrix or list object with netify attributes removed. the structure returned depends on the type of netify object:

- cross-sectional networks: returns a matrix
- longitudinal array networks: returns a matrix
- longitudinal list networks: returns a list

Author(s)

shahryar minhas

`ggplot_add.netify_edge` *add netify_edge to ggplot*

Description

s3 method to add netify_edge objects to ggplot objects. this method is called automatically when using the + operator with a netify_edge object.

Usage

```
## S3 method for class 'netify_edge'
ggplot_add(object, plot, ...)
```

Arguments

object	a netify_edge object created by netify_edge
plot	a ggplot object to which the edge layer will be added
...	additional arguments passed by ggplot2 (used internally)

Value

a ggplot object with the edge layer added

Author(s)

cassy dorff, shahryar minhas

---

ggplot_add.netify_label

add netify_label to ggplot

Description

s3 method to add netify_label objects to ggplot objects. this method is called automatically when using the + operator with a netify_label object.

Usage

```
## S3 method for class 'netify_label'
ggplot_add(object, plot, ...)
```

Arguments

object	a netify_label object created by netify_label
plot	a ggplot object to which the label layer will be added
...	additional arguments passed by ggplot2 (used internally)

Value

a ggplot object with the label layer added

Author(s)

cassy dorff, shahryar minhas

ggplot_add.netify_label_repel

add netify_label_repel to ggplot

Description

s3 method to add netify_label_repel objects to ggplot objects. this method is called automatically when using the + operator with a netify_label_repel object.

Usage

```
## S3 method for class 'netify_label_repel'
ggplot_add(object, plot, ...)
```

Arguments

object	a netify_label_repel object created by netify_label_repel
plot	a ggplot object to which the label_repel layer will be added
...	additional arguments passed by ggplot2 (used internally)

Value

a ggplot object with the label_repel layer added

Author(s)

cassy dorff, shahryar minhas

ggplot_add.netify_labels

add netify scale labels to ggplot

Description

s3 method to add netify_labels objects to ggplot objects. this method updates the labels of existing scales based on the specifications in the netify_labels object.

Usage

```
## S3 method for class 'netify_labels'
ggplot_add(object, plot, ...)
```

Arguments

object	a netify_labels object created by netify_scale_labels
plot	a ggplot object to which the labels will be added
...	additional arguments passed by ggplot2 (used internally)

Value

a ggplot object with updated scale labels

Author(s)

cassy dorff, shahryar minhas

---

ggplot_add.netify_node

add netify_node to ggplot

Description

s3 method to add netify_node objects to ggplot objects. this method is called automatically when using the + operator with a netify_node object.

Usage

```
## S3 method for class 'netify_node'
ggplot_add(object, plot, ...)
```

Arguments

object	a netify_node object created by netify_node
plot	a ggplot object to which the node layer will be added
...	additional arguments passed by ggplot2 (used internally)

Value

a ggplot object with the node layer added

Author(s)

cassy dorff, shahryar minhas

ggplot_add.netify_scale_reset

add scale resets to ggplot

Description

s3 method to add scale reset objects to ggplot objects. this method resets color, fill, alpha, and size scales using the ggnewscale package.

Usage

```
## S3 method for class 'netify_scale_reset'
ggplot_add(object, plot, ...)
```

Arguments

`object`	a netify_scale_reset object created by `reset_scales`
`plot`	a ggplot object to which scale resets will be added
`...`	additional arguments passed by ggplot2 (used internally)

Value

a ggplot object with scale resets applied

Author(s)

cassy dorff, shahryar minhas

ggplot_add.netify_text

add netify_text to ggplot

Description

s3 method to add netify_text objects to ggplot objects. this method is called automatically when using the + operator with a netify_text object.

Usage

```
## S3 method for class 'netify_text'
ggplot_add(object, plot, ...)
```

Arguments

`object`	a netify_text object created by `netify_text`
`plot`	a ggplot object to which the text layer will be added
`...`	additional arguments passed by ggplot2 (used internally)

Value

a ggplot object with the text layer added

Author(s)

cassy dorff, shahryar minhas

`ggplot_add.netify_text_repel` *add netify_text_repel to ggplot*

Description

s3 method to add netify_text_repel objects to ggplot objects. this method is called automatically when using the + operator with a netify_text_repel object.

Usage

```
## S3 method for class 'netify_text_repel'
ggplot_add(object, plot, ...)
```

Arguments

`object`	a netify_text_repel object created by `netify_text_repel`
`plot`	a ggplot object to which the text_repel layer will be added
`...`	additional arguments passed by ggplot2 (used internally)

Value

a ggplot object with the text_repel layer added

Author(s)

cassy dorff, shahryar minhas

`glance.netify` *one-row-per-network summary of a netify object (broom style)*

Description

`glance.netify` is an s3 method for the `glance()` generic from the broom package. it returns the graph-level statistics produced by `summary.netify()` – one row per network / time period / layer – so the netify object plays nicely with broom-style workflows. (broom is not a hard dependency; this method is registered as an s3 method on `glance` and only triggers when the generic is available.)

Usage

```
glance.netify(x, ...)
```

Arguments

`x`	a netify object.
`...`	additional arguments passed to `summary.netify()` (e.g., `other_stats = list(my_stat = my_fn)`).

Value

a tibble (or data.frame if `tibble` isn't installed): one row per (network, time, layer) combination with density, reciprocity, mutual, transitivity, edge counts, etc.

Author(s)

cassy dorff, shahryar minhas

See Also

`summary.netify()` for the underlying summary, and `tidy.netify` for one-row-per-edge output.

Examples

```
data(icews)
icews_10 <- icews[icews$year == 2010, ]
net <- netify(icews_10, actor1 = "i", actor2 = "j",
symmetric = FALSE, weight = "verbCoop")
glance.netify(net)
```

homophily *Analyze homophily in network data*

Description

tests whether connected actors have similar attributes (homophily). calculates the correlation between attribute similarity and tie presence, with support for multiple similarity metrics and significance testing.

Usage

```
homophily(
  netlet,
  attribute,
  method = "correlation",
  threshold = 0,
  signed_handling = c("abs", "drop_negative", "preserve_sign"),
  significance_test = TRUE,
  n_permutations = 1000,
  alpha = 0.05,
  other_stats = NULL,
  ...
)
```

Arguments

netlet	a netify object containing network data.
attribute	character string specifying the nodal attribute to analyze.
method	character string specifying the similarity metric: **"correlation"** negative absolute difference for continuous data (default) **"euclidean"** negative euclidean distance for continuous data **"manhattan"** negative manhattan/city-block distance for continuous data **"cosine"** cosine similarity for continuous data **"categorical"** binary similarity (0/1) for categorical data **"jaccard"** jaccard similarity for binary/presence-absence data **"hamming"** negative hamming distance for categorical data
threshold	numeric value or function to determine tie presence in weighted networks. if numeric, edges with weights > threshold are considered ties. if a function, it should take the network matrix and return a logical matrix. default is 0 (any positive weight is a tie). common values: 0 (default), mean(weights), median(weights), or quantile-based thresholds. for pre-binarized networks, consider using `mutate_weights()` first.
signed_handling	character. strategy for signed (negative-weight) edges:

"abs" (default) take absolute values before thresholding so any tie magnitude – positive or negative – can become a connection.

"drop_negative" set negative weights to zero before thresholding (positive ties only).

"preserve_sign" treat any non-zero entry (positive or negative) as a connection; threshold is ignored for sign decisions.

ignored when the network has no negative weights.

significance_test
: logical. whether to perform a dyad-level permutation test. default TRUE.

n_permutations
: number of permutations for significance testing. default 1000.

alpha
: significance level for confidence intervals. default 0.05.

other_stats
: named list of custom functions for additional statistics.

...
: additional arguments passed to custom functions.

Details

auto-promotion to categorical**:**

if you leave method at its default and pass a character, factor, or logical attribute, homophily() will switch to method = "categorical" automatically and inform you once per attribute. this avoids the c++-level error that would otherwise come from feeding non-numeric data to the correlation-based similarity routine. pass method = "categorical" (or any other explicit choice) to silence the message.

similarity metrics:

for continuous attributes:

- correlation: based on absolute difference, good general purpose metric
- euclidean: similar to correlation for single attributes
- manhattan: less sensitive to outliers than euclidean
- cosine: useful for normalized data or when sign matters

for categorical/binary attributes:

- categorical: simple matching (1 if same, 0 if different)
- jaccard: for binary data, emphasizes shared presence over shared absence
- hamming: counts positions where values differ (negated for similarity)

threshold parameter:

for weighted networks, the threshold parameter determines what edge weights constitute a "connection". you can specify:

- a numeric value: edges with weight > threshold are ties
- a function: should take a matrix and return a single numeric threshold
- common threshold functions:
  - function(x) mean(x, na.rm = TRUE) - mean weight

- `function(x) median(x, na.rm = TRUE)` - median weight
- `function(x) quantile(x, 0.75, na.rm = TRUE)` - 75th percentile

for more complex binarization needs (e.g., different thresholds by time period), consider using `mutate_weights()` to pre-process your network.

permutation test:

when `significance_test = TRUE`, `homophily()` holds the tie indicators fixed and permutes the dyad-level similarity values to form an exploratory reference distribution. the resulting p-value and confidence interval summarize how unusual the observed dyad-level association is under that exchangeability assumption. they are not a node-label permutation test and should not be read as a causal estimate of tie formation.

Value

data frame with homophily statistics per network/time period:

`net` network/time identifier

`layer` layer name

`attribute` analyzed attribute name

`method` similarity method used

`threshold_value` threshold used for determining ties (na for binary networks)

`homophily_correlation` correlation between similarity and tie presence (binary: tie/no tie)

`mean_similarity_connected` mean similarity among connected pairs (weight > threshold)

`mean_similarity_unconnected` mean similarity among unconnected pairs (weight <= threshold or missing)

`similarity_difference` difference between connected and unconnected mean similarities

`p_value` permutation test p-value

`ci_lower`, `ci_upper` confidence interval bounds

`n_connected_pairs` number of connected pairs

`n_unconnected_pairs` number of unconnected pairs

Author(s)

cassy dorff, shahryar minhas

Examples

```
# quick homophily check on the bundled classroom friendship data
data(classroom_edges)
data(classroom_nodes)
net <- netify(
    classroom_edges,
    actor1 = "from", actor2 = "to",
    symmetric = TRUE,
    nodal_data = classroom_nodes
)
```

```
# do students cluster by gender?
homophily(net, attribute = "gender", method = "categorical")


# default method auto-promotes character attributes to categorical
homophily_default <- homophily(net, attribute = "gender",
    n_permutations = 100)

# using different similarity metrics for continuous data
homophily_manhattan <- homophily(
    net,
    attribute = "gpa",
    method = "manhattan",
    n_permutations = 100
)

# for categorical attributes
homophily_grade <- homophily(
    net,
    attribute = "grade",
    method = "categorical",
    n_permutations = 100
)
```

`icews` *Event data slice from ICEWS*

Description

event data from icews for select countries from 2002 to 2014, and additional nodal variables from the world bank and polity iv.

Usage

```
data(icews)
```

Format

a data frame with dyadic observations and the following variables:

i source country

j target country

year year of observation

id observation identifier

verbCoop verbal cooperation score

matlCoop material cooperation score

verbConf verbal conflict score

matlConf material conflict score

i_year year for source country

j_year year for target country

i_polity2 polity iv democracy score for source country

j_polity2 polity iv democracy score for target country

i_iso3c iso 3166-1 alpha-3 code for source country

j_iso3c iso 3166-1 alpha-3 code for target country

i_region region for source country

j_region region for target country

i_gdp gdp for source country

j_gdp gdp for target country

i_log_gdp log gdp for source country

j_log_gdp log gdp for target country

i_pop population for source country

j_pop population for target country

i_log_pop log population for source country

j_log_pop log population for target country

Author(s)

cassy dorff, shahryar minhas

Source

icews coded event data, harvard dataverse.

References

boschee, e., lautenschlager, j., o'brien, s., shellman, s., starz, j., and ward, m. (2015). icews coded event data. harvard dataverse. doi:10.7910/dvn/28075.

Examples

```
data(icews)
icews[1:3, ]
```

is_netify *is this object a netify object?*

Description

is this object a netify object?

Usage

```
is_netify(x)
```

Arguments

x	an r object

Value

logical constant, TRUE if argument x is a netify object

Author(s)

colin henry

layer_netify *Create multilayer networks from multiple netify objects*

Description

layer_netify combines multiple netify objects into a single multilayer network structure. each input network becomes a layer in the resulting multilayer network, enabling analysis of multiple relationship types or network views simultaneously.

Usage

```
layer_netify(netlet_list, ..., layer_labels = NULL)
```

Arguments

netlet_list	a list of netify objects to layer together, or the first netify object when passing multiple objects as separate arguments via .... all objects must have compatible dimensions and attributes (see details).
...	additional netify objects. when provided, netlet_list should be a single netify object and all arguments are collected into a list. this allows calling layer_netify(net1, net2, net3) as a shorthand for layer_netify(list(net1, net2, net3)).
layer_labels	character vector specifying names for each layer. if NULL (default), uses names from netlet_list or generates generic labels ("layer1", "layer2", etc.). length must match the number of netify objects.

Details

compatibility requirements:

all netify objects in netlet_list must have identical:

- network type (cross-sectional or longitudinal)
- dimensions (same actors and time periods)
- mode (all unipartite or all bipartite)
- actor composition (same actors in same order)

mixed directedness:

layers are allowed to have different symmetry settings (e.g., some symmetric and others directed). when layers have mixed directedness, the `symmetric` attribute is stored as a named logical vector with one element per layer.

attribute handling:

nodal and dyadic attributes are combined when possible:

- if attributes are identical across layers, the first layer's attributes are used
- if attributes differ but have compatible structure, they are merged
- if attributes are incompatible, a warning is issued and attributes must be added manually using `add_node_vars()` or `add_dyad_vars()`

Value

a multilayer netify object with structure depending on input type:

- **cross-sectional input**: 3d array `[actors x actors x layers]`
- **longitudinal array input**: 4d array `[actors x actors x layers x time]`
- **longitudinal list input**: list of 3d arrays, one per time period

the returned object maintains the netify class and includes:

- combined nodal attributes (if compatible)
- combined dyadic attributes (if compatible)
- layer information accessible via `attr(obj, 'layers')`
- all standard netify attributes

Note

memory usage increases with the number of layers. for large networks with many layers, consider whether all layers are necessary for your analysis.

Author(s)

cassy dorff, shahryar minhas

Examples

```
# load example data
data(icews)

# example 1: cross-sectional multilayer network
icews_10 <- icews[icews$year == 2010, ]

# create separate networks for different interaction types
verbal_coop <- netify(
    icews_10,
    actor1 = "i", actor2 = "j",
    symmetric = FALSE,
    weight = "verbCoop",
    nodal_vars = c("i_log_gdp", "i_log_pop"),
    dyad_vars = "verbConf"
)

material_coop <- netify(
    icews_10,
    actor1 = "i", actor2 = "j",
    symmetric = FALSE,
    weight = "matlCoop",
    nodal_vars = "i_polity2",
    dyad_vars = "matlConf"
)

# layer them together
coop_multilayer <- layer_netify(
    netlet_list = list(verbal_coop, material_coop),
    layer_labels = c("verbal", "material")
)

# check structure
dim(get_raw(coop_multilayer)) # [actors x actors x 2]
attr(coop_multilayer, "layers") # "verbal" "material"


# example 2: longitudinal multilayer (array format)
verbal_longit <- netify(
    icews,
    actor1 = "i", actor2 = "j", time = "year",
    symmetric = FALSE,
    weight = "verbCoop",
    output_format = "longit_array"
)

material_longit <- netify(
    icews,
    actor1 = "i", actor2 = "j", time = "year",
    symmetric = FALSE,
    weight = "matlCoop",
    output_format = "longit_array"
```

```
)

# create longitudinal multilayer
longit_multilayer <- layer_netify(
    list(verbal_longit, material_longit),
    layer_labels = c("verbal", "material")
)

dim(get_raw(longit_multilayer)) # [actors x actors x 2 x years]
```

---

list_network_styles *list available network styles*

Description

shows all available preset network styles with descriptions

Usage

```
list_network_styles()
```

Value

a data frame with style names and descriptions

Author(s)

cassy dorff, shahryar minhas

---

list_palettes *list available color palettes*

Description

list available color palettes

Usage

```
list_palettes()
```

Value

character vector of available palette names

Author(s)

cassy dorff, shahryar minhas

`melt` *Melt methods for netify objects*

Description

convert netify matrices/arrays to long format data frames. these methods provide a consistent interface for melting different types of netify objects while leveraging c++ for performance.

Usage

```
melt(data, ...)

## S3 method for class 'netify'
melt(
  data,
  ...,
  remove_diagonal = TRUE,
  remove_zeros = TRUE,
  na.rm = TRUE,
  value.name = "value"
)
```

Arguments

`data`	a netify object
`...`	additional arguments (see details)
`remove_diagonal`	logical. remove diagonal elements (default: TRUE)
`remove_zeros`	logical. remove zero values (default: TRUE)
`na.rm`	logical. remove na values (default: TRUE)
`value.name`	character. name for value column (default: "value")

Details

the melt method converts netify objects from their matrix representation to a long format data frame suitable for analysis and visualization. the output format depends on the type of netify object:

- cross-sectional: returns columns `var1`, `var2`, `value`
- longitudinal: returns columns `var1`, `var2`, `time`, `value`
- multilayer: returns columns `var1`, `var2`, `layer`, `value` (and `time` if longitudinal)

Value

see method-specific documentation (e.g., melt.netify)

data frame with columns: var1, var2, value (and optionally time / layer). the var1 / var2 names are inherited from base r's as.data.frame.table / reshape2::melt heritage and are used by internal helpers (decompose_helpers, plot_homophily, etc.); rename them yourself downstream if you need snake_case (e.g. rlang::set_names(out, c("from", "to", "value", ...))). for a fully snake_case, dyad-attribute-merged edge frame, use unnetify() or as_tibble.netify() instead.

Author(s)

cassy dorff, shahryar minhas

---

merge.netify *merge method for netify objects (s3 alias for* bind_netifies*)*

Description

provides the base-r merge() generic dispatch for combining two netify objects along the time axis. for more than two inputs or programmatic use, prefer bind_netifies() directly.

Usage

```
## S3 method for class 'netify'
merge(x, y, ...)
```

Arguments

x, y	netify objects.
...	additional netify objects, or a names = argument.

Value

a longit_list netify object.

Author(s)

cassy dorff, shahryar minhas

mexico *Event data slice from UCDP on Mexico*

Description

event data from ucdp for mexico

Usage

```
data(mexico)
```

Format

a data frame with dyadic event observations.

Author(s)

cassy dorff, shahryar minhas

Source

ucdp georeferenced event dataset, version 23.1. https://ucdp.uu.se/downloads/

References

davies, s., pettersson, t., and oberg, m. (2023). organized violence 1989-2022, and the return of conflict between states. journal of peace research, 60(4), 691-708. doi:10.1177/00223433231185169.

sundberg, r. and melander, e. (2013). introducing the ucdp georeferenced event dataset. journal of peace research, 50(4), 523-532. doi:10.1177/0022343313484347.

Examples

```
data(mexico)
mexico[1:3, ]
```

mixing_matrix *Create attribute mixing matrices for network data*

Description

creates cross-tabulation matrices showing how connections are distributed across different attribute values. this reveals mixing patterns and assortativity in networks by examining the frequency of ties between actors with different attribute combinations.

Usage

```
mixing_matrix(
  netlet,
  attribute,
  row_attribute = NULL,
  normalized = TRUE,
  by_row = FALSE,
  include_weights = FALSE,
  other_stats = NULL,
  ...
)
```

Arguments

netlet	a netify object containing network data.
attribute	character string specifying the nodal attribute to analyze.
row_attribute	optional different attribute for matrix rows. if NULL, uses the same attribute for both dimensions.
normalized	logical. whether to return proportions instead of raw counts. default TRUE.
by_row	logical. if TRUE and normalized=TRUE, normalizes by row. default FALSE.
include_weights	logical. whether to use edge weights. default FALSE.
other_stats	named list of custom functions for additional statistics.
...	additional arguments passed to custom functions.

Details

mixing matrix elements represent ties between actors with attribute values i and j. for undirected networks, matrices are symmetrized. assortativity ranges from -1 (disassortative) to 1 (assortative).

Value

list containing:

`mixing_matrices` named list of mixing matrices per time/layer

`summary_stats` data frame with mixing statistics:

- `net` network/time identifier
- `layer` layer name
- `attribute` name of analyzed attribute(s)
- `assortativity` assortativity coefficient (-1 to 1)
- `diagonal_proportion` proportion of within-group ties
- `entropy` shannon entropy of mixing pattern
- `modularity` modularity based on attribute groups
- `n_groups` number of attribute categories
- `total_ties` total number of ties analyzed

Author(s)

cassy dorff, shahryar minhas

Examples

```
# who tends to befriend whom, by gender, in the bundled classroom data
data(classroom_edges)
data(classroom_nodes)
net <- netify(
    classroom_edges,
    actor1 = "from", actor2 = "to",
    symmetric = TRUE,
    nodal_data = classroom_nodes
)
mm <- mixing_matrix(net, attribute = "gender")
round(mm$mixing_matrices[[1]], 3)
mm$summary_stats
```

---

mutate_weights *Mutate edge weights in a netify object*

Description

`mutate_weights` applies mathematical transformations to edge weights in a netify object. this is useful for normalizing data, handling skewed distributions, creating binary networks, or applying any custom mathematical transformation to network weights.

Usage

```
mutate_weights(
  netlet,
  transform_fn = NULL,
  add_constant = 0,
  new_name = NULL,
  keep_original = TRUE
)
```

Arguments

`netlet`	a netify object with edge weights to transform
`transform_fn`	a function to apply to the weights. can be any function that takes a matrix and returns a matrix (e.g., `log`, `sqrt`, `function(x) x^2`). if NULL, only the `add_constant` operation is performed.
`add_constant`	numeric value to add to weights before applying `transform_fn`. useful for log transformations (e.g., `add_constant = 1` for `log(x + 1)`) or shifting distributions.

new_name	optional new name for the weight variable. if provided, updates the weight attribute and descriptive labels. if NULL, keeps the original name.
keep_original	logical. if TRUE (default), preserves the original weights as a dyadic variable. if FALSE, discards original weights to save memory.

Details

the function handles all netify object types:

- **cross-sectional**: transforms the single network matrix
- **longitudinal arrays**: transforms each time slice
- **longitudinal lists**: transforms each time period matrix

the function automatically updates network attributes:

- updates `is_binary` if transformation results in 0/1 values
- updates `detail_weight` with transformation description
- preserves all other network and nodal attributes

for longitudinal arrays, original weight preservation is not yet implemented and will show an informational message.

Value

a netify object with transformed weights. the original weights are optionally preserved as a dyadic variable named "original_weight".

Author(s)

cassy dorff, shahryar minhas

Examples

```
# load example data
data(icews)
icews_2010 <- icews[icews$year == 2010, ]

# create a weighted network
net <- netify(
    icews_2010,
    actor1 = "i", actor2 = "j",
    symmetric = FALSE,
    weight = "verbCoop"
)

# example 1: log transformation (common for skewed data)
net_log <- mutate_weights(
    net,
    transform_fn = log,
    add_constant = 1, # log(x + 1) to handle zeros
    new_name = "log_verbCoop"
```

```
)
print(net_log)

# example 2: square root transformation (moderate skewness)
net_sqrt <- mutate_weights(
    net,
    transform_fn = sqrt,
    new_name = "sqrt_verbCoop"
)

# example 3: binarization (convert to presence/absence)
net_binary <- mutate_weights(
    net,
    transform_fn = function(x) ifelse(x > 0, 1, 0),
    new_name = "verbCoop_binary"
)

# example 4: standardization (z-scores)
net_std <- mutate_weights(
    net,
    transform_fn = function(x) {
        mean_x <- mean(x, na.rm = TRUE)
        sd_x <- sd(x, na.rm = TRUE)
        return((x - mean_x) / sd_x)
    },
    new_name = "verbCoop_standardized"
)

# example 5: rank transformation
net_rank <- mutate_weights(
    net,
    transform_fn = function(x) rank(x, na.last = "keep"),
    new_name = "verbCoop_ranked"
)

# example 6: power transformation
net_power <- mutate_weights(
    net,
    transform_fn = function(x) x^0.5, # square root as power
    new_name = "verbCoop_power"
)

# example 7: min-max normalization (scale to 0-1)
net_norm <- mutate_weights(
    net,
    transform_fn = function(x) {
        min_x <- min(x, na.rm = TRUE)
        max_x <- max(x, na.rm = TRUE)
        return((x - min_x) / (max_x - min_x))
    },
    new_name = "verbCoop_normalized"
)
```

```
# example 8: winsorization (cap extreme values)
net_winsor <- mutate_weights(
    net,
    transform_fn = function(x) {
        q95 <- quantile(x, 0.95, na.rm = TRUE)
        return(pmin(x, q95)) # cap at 95th percentile
    },
    new_name = "verbCoop_winsorized"
)

# example 9: only add constant (no transformation function)
net_shifted <- mutate_weights(
    net,
    add_constant = 10,
    new_name = "verbCoop_shifted"
)

# example 10: don't keep original weights to save memory
net_log_compact <- mutate_weights(
    net,
    transform_fn = log1p, # log(1 + x), handles zeros automatically
    new_name = "log1p_verbCoop",
    keep_original = FALSE
)

# example 11: longitudinal network transformation

# create longitudinal network
net_longit <- netify(
    icews,
    actor1 = "i", actor2 = "j", time = "year",
    symmetric = FALSE,
    weight = "verbCoop",
    actor_time_uniform = FALSE
)

# transform across all time periods
net_longit_log <- mutate_weights(
    net_longit,
    transform_fn = log1p,
    new_name = "log_verbCoop"
)


# example 12: custom transformation with multiple operations
net_custom <- mutate_weights(
    net,
    transform_fn = function(x) {
        # complex transformation: log, then standardize
        x_log <- log(x + 1)
        x_std <- (x_log - mean(x_log, na.rm = TRUE)) / sd(x_log, na.rm = TRUE)
        return(x_std)
    },
```

```
    new_name = "verbCoop_log_std"
)
```

---

myanmar *event data slice from ucdp on myanmar*

Description

event data from ucdp for myanmar

Usage

```
data(myanmar)
```

Format

a data frame with dyadic event observations.

Author(s)

cassy dorff, shahryar minhas

Source

ucdp georeferenced event dataset, version 23.1. https://ucdp.uu.se/downloads/

References

davies, s., pettersson, t., and oberg, m. (2023). organized violence 1989-2022, and the return of conflict between states. journal of peace research, 60(4), 691-708. doi:10.1177/00223433231185169.

sundberg, r. and melander, e. (2013). introducing the ucdp georeferenced event dataset. journal of peace research, 50(4), 523-532. doi:10.1177/0022343313484347.

Examples

```
data(myanmar)
myanmar[1:3, ]
```

net_plot_data *Prepare netify data for network visualization*

Description

net_plot_data processes a netify object and generates all necessary components for network visualization. this function handles layout computation, aesthetic parameter organization, and data structuring for subsequent plotting with ggplot2 or other visualization tools.

Usage

```
net_plot_data(netlet, plot_args = list())
```

Arguments

netlet	a netify object (class "netify") containing the network to be visualized. cross-sectional, longitudinal, and multilayer netify objects are supported.
plot_args	a list of plotting arguments controlling visualization appearance and behavior. can include:

layout parameters:

- point_layout: pre-computed node positions as a data.frame or list of data.frames (for longitudinal networks). if provided, overrides layout algorithm selection
- layout: character string specifying the igraph layout algorithm. options: "nicely" (default), "fr" (fruchterman-reingold), "kk" (kamada-kawai), "circle", "star", "grid", "tree", "bipartite", and others. see get_node_layout for full list
- static_actor_positions: logical. if TRUE, maintains consistent node positions across time periods in longitudinal networks
- which_static: integer specifying which time period to use as the template for static positions
- seed: integer for reproducible random layouts

display options:

- remove_isolates: logical. remove unconnected nodes (default: TRUE)
- add_edges: logical. include edges in visualization (default: TRUE)
- curve_edges: logical. use curved edges instead of straight (default: FALSE)
- add_points: logical. display nodes as points (default: TRUE)
- add_text: logical. add text labels to nodes (default: FALSE)
- add_label: logical. add boxed labels to nodes (default: FALSE)

selective labeling:

- select_text: character vector of node names to label with text
- select_label: character vector of node names to label with boxes

additional aesthetic parameters are processed by adjust_plot_args and gg_params.

Details

this function serves as the data preparation layer for netify visualization, performing several operations:

data validation:

- ensures the input is a valid netify object
- handles single-layer and multilayer networks
- validates ego networks contain only one ego

layout computation:

- generates node positions using specified algorithm if not provided
- calculates edge endpoint coordinates based on node positions
- handles both cross-sectional and longitudinal layouts

data organization:

- merges layout information with network attributes
- processes plotting arguments and applies defaults
- organizes aesthetic parameters for ggplot2 compatibility
- removes isolates if requested

output structure:

the returned data is structured for direct use with ggplot2 or can be further customized. the separation of layout, aesthetics, and data allows for flexible visualization workflows.

Value

a list with three components for creating network visualizations:

- **plot_args**: processed plotting arguments with defaults applied and parameters validated. includes all layout and display settings
- **ggnet_params**: organized aesthetic parameters for ggplot2 mapping. contains separate specifications for nodes, edges, text, and labels with both static and dynamic (data-mapped) aesthetics
- **net_dfs**: data frames ready for plotting:
  - `nodal_data`: node information including positions (x, y), attributes, and any additional variables
  - `edge_data`: edge information including endpoint coordinates (x1, y1, x2, y2) and edge attributes

Note

this function is primarily designed for use with netify's plot method but can be called directly for custom visualization workflows.

for multilayer networks, the returned node and edge data include a `layer` column.

for ego networks with multiple egos, create separate visualizations and combine them using packages like patchwork.

Author(s)

cassy dorff, shahryar minhas

`netify` *Create network object from various data types*

Description

this function takes in various types of network data (dyadic datasets, matrices, arrays, lists, igraph objects, or network objects) and outputs a netify object.

Usage

```
netify(
  input,
  actor1 = NULL,
  actor2 = NULL,
  time = NULL,
  symmetric = TRUE,
  mode = "unipartite",
  weight = NULL,
  sum_dyads = FALSE,
  actor_time_uniform = TRUE,
  actor_pds = NULL,
  diag_to_NA = TRUE,
  missing_to_zero = TRUE,
  output_format = NULL,
  nodal_vars = NULL,
  dyad_vars = NULL,
  dyad_vars_symmetric = rep(symmetric, length(dyad_vars)),
  input_type = c("auto", "dyad_df", "netify_obj"),
  nodelist = NULL,
  force_dense = FALSE,
  ...
)
```

Arguments

`input` data object to netify. can be:

- a data.frame (or tibble/data.table) with dyadic data
- a matrix representing an adjacency matrix
- a 3d array representing longitudinal networks
- a list of matrices representing longitudinal networks
- an igraph object
- a network object (from the network package)

- a list of igraph or network objects

`actor1` character: name of the actor 1 variable in the data (required for data.frame inputs)

`actor2` character: name of the actor 2 variable in the data (required for data.frame inputs)

`time` character: name of the time variable in the data. can contain numeric, date, posixct/posixlt, or character values. non-numeric types will be converted to numeric indices while preserving original labels. if no time is provided then a cross-sectional network will be created.

`symmetric` logical: whether ties are symmetric, default is TRUE. for matrix, array, list, igraph, or network inputs this default is ignored unless explicitly set by the caller; symmetry is instead detected from the input itself. the default applies for data.frame inputs.

`mode` character: whether the network is unipartite or bipartite, default is unipartite. as with `symmetric`, this default applies to data.frame inputs and is auto-detected for matrix / array / list / igraph / network inputs unless explicitly set.

`weight` character: name of the weighted edge variable in the data, default is NULL

`sum_dyads` logical: whether to sum up the `weight` value when there exists repeating dyads

`actor_time_uniform`
logical: whether to assume actors are the same across the full time series observed in the data TRUE means that actors are the same across the full time series observed in the data and the outputted netify object will be in an array format. FALSE means that actors come in and out of the observed data and their "existence" should be determined by the data, meaning that their first year of existence will be determined by the time point of their first event and their last year of existence by the time point of their last event. outputted netify object will be in a list format.

`actor_pds` a data.frame indicating start and end time point for every actor, this can be created using `get_actor_time_info`, unless provided this will estimated for the user based on their choice of `actor_time_uniform`

`diag_to_NA` logical: whether diagonals should be set to na, default is TRUE. for matrix / array / list inputs, the default is ignored unless explicitly set: instead, `diag_to_NA` is auto-detected by inspecting whether the diagonal of the supplied matrix is already `na`. pass an explicit value to override the auto-detection.

`missing_to_zero`
logical: whether missing values should be set to zero, default is TRUE. as with `diag_to_NA`, this is auto-detected for matrix / array / list inputs based on whether the supplied data already contains off-diagonal `na`s.

`output_format` character: "cross_sec", "longit_array", or "longit_list". if not specified and time is NULL then output_format will be "cross_sec" and if time is specified then output_format will default to "longit_list". only applies to data.frame inputs.

`nodal_vars` character vector: names of the nodal variables in the input that should be added as attributes to the netify object (for data.frame inputs)

`dyad_vars` character vector: names of the dyadic variables in the input that should be added as attributes to the netify object (for data.frame inputs)

dyad_vars_symmetric
: logical vector: whether ties are symmetric, default is to use the same choice as the symmetric argument

input_type
: character: force specific input type interpretation. options are "auto" (default), "dyad_df", or "netify_obj". use "dyad_df" to force data.frame interpretation or "netify_obj" to force matrix/array/ igraph/network interpretation.

nodelist
: optional list of all actors (nodes) that should be included in the network. for unipartite networks, pass a character vector. for bipartite networks, pass partition-aware input such as `list(row = row_actors, col = col_actors)` or a data frame with actor and mode columns. a flat vector can include only actors already observed in a bipartite row or column mode because new isolates cannot be assigned to a partition from their names alone.

force_dense
: logical: when a `matrix::sparsematrix` input would densify to a large allocation (n > 5000 and density < 1%), `netify()` aborts with a guidance message. set `force_dense = TRUE` to override the guard and proceed with densification.

...
: additional arguments passed to `to_netify` when processing network objects

Value

a netify object

Author(s)

ha eun choi, cassy dorff, colin henry, shahryar minhas

See Also

`netify_workflows` for an overview of the create / explore / model workflow and how `netify()` fits into it; `add_node_vars`, `add_dyad_vars` for attaching attributes after construction; and `classroom_edges` / `classroom_nodes` for a small worked example.

Examples

```
# load example directed event data from icews
# this data comes in the form of a dyadic
# dataframe where all dyad pairs are listed
data(icews)

# from a data.frame: generate a longitudinal, directed and weighted network
# where the weights are matlConf
icews_matlConf <- netify(
    input = icews,
    actor1 = "i", actor2 = "j", time = "year",
    symmetric = FALSE, weight = "matlConf"
)

# from a matrix
adj_matrix <- matrix(rbinom(100, 1, 0.3), 10, 10)
net_from_matrix <- netify(adj_matrix)
```

```
# from an igraph object

library(igraph)
g <- sample_gnp(10, 0.3)
net_from_igraph <- netify(g)
```

`netify_edge` *Extract edges layer from netify plot components*

Description

extracts the edge layer from a netify plot components object, allowing for manual plot construction and customization. this function is part of the modular plotting system that enables fine-grained control over network visualization elements.

Usage

```
netify_edge(comp)
```

Arguments

`comp`	a netify_plot_components object returned from `plot(..., return_components = TRUE)`

Value

a custom object of class "netify_edge" that can be added to a ggplot object using the + operator. the object contains the edge layer with all its aesthetic mappings and data.

Author(s)

cassy dorff, shahryar minhas

See Also

`plot.netify`, `netify_node`, `assemble_netify_plot`

Examples

```
# create a netify object
mat <- matrix(c(NA, 1, 0, 0, NA, 1, 1, 0, NA), 3, 3,
    dimnames = list(c("alice", "bob", "carol"),
                    c("alice", "bob", "carol")))
net <- new_netify(mat, symmetric = FALSE)

# get plot components
comp <- plot(net, return_components = TRUE)
```

```
# build custom plot with edges
library(ggplot2)
ggplot() +
    netify_edge(comp)
```

`netify_label` *extract label layer from netify plot components*

Description

extracts the label layer from a netify plot components object. labels display actor names or other text annotations with background boxes, making them more visible against complex network backgrounds.

Usage

```
netify_label(comp)
```

Arguments

`comp`	a netify_plot_components object returned from `plot(..., return_components = TRUE)`

Value

a custom object of class "netify_label" that can be added to a ggplot object using the + operator. the object contains the label layer with all its aesthetic mappings and data.

Author(s)

cassy dorff, shahryar minhas

See Also

`plot.netify`, `netify_text`, `assemble_netify_plot`

Examples

```
# create a netify object
mat <- matrix(c(NA, 1, 0, 0, NA, 1, 1, 0, NA), 3, 3,
    dimnames = list(c("alice", "bob", "carol"),
                    c("alice", "bob", "carol")))
net <- new_netify(mat, symmetric = FALSE)

# get plot components with labels
comp <- plot(net, add_label = TRUE, return_components = TRUE)
```

```
# build custom plot with labels
library(ggplot2)
ggplot() +
    netify_label(comp)
```

`netify_label_repel` *extract label_repel layer from netify plot components*

Description

extracts the label_repel layer from a netify plot components object. label repel annotations display actor names with background boxes and automatic repositioning to avoid overlaps, providing optimal readability in dense networks.

Usage

```
netify_label_repel(comp)
```

Arguments

`comp`	a netify_plot_components object returned from `plot(..., return_components = TRUE)`

Value

a custom object of class "netify_label_repel" that can be added to a ggplot object using the + operator. the object contains the label_repel layer with all its aesthetic mappings and data.

Author(s)

cassy dorff, shahryar minhas

See Also

`plot.netify`, `netify_label`, `assemble_netify_plot`

Examples

```
# create a netify object
mat <- matrix(c(NA, 1, 0, 0, NA, 1, 1, 0, NA), 3, 3,
    dimnames = list(c("alice", "bob", "carol"),
                    c("alice", "bob", "carol")))
net <- new_netify(mat, symmetric = FALSE)

# get plot components with label_repel
comp <- plot(net, add_label_repel = TRUE, return_components = TRUE)
```

```
# build custom plot with repelled labels
library(ggplot2)
ggplot() +
    netify_label_repel(comp)
```

`netify_measurements` *Extract measurements and dimensions from a netify object*

Description

`netify_measurements` (also available as `measurements`) extracts information about the structure, dimensions, and attributes of a netify object. this function provides a standardized way to inspect network properties across different netify types.

Usage

```
netify_measurements(netlet)

measurements(netlet)
```

Arguments

`netlet`	a netify object (class "netify") to analyze. can be cross-sectional, longitudinal array, or longitudinal list format.

Details

the function will adapt its output based on the netify object type.

Value

a list containing measurements of the netify object with the following components (availability depends on netify type):

actor information:

- `row_actors`: character vector (or list) of row actor names
- `col_actors`: character vector (or list) of column actor names
- `n_row_actors`: integer (or list) count of row actors
- `n_col_actors`: integer (or list) count of column actors

temporal information:

- `time`: character vector of time period labels (NULL for cross-sectional)
- `n_time`: integer count of time periods (NULL for cross-sectional)

layer information:

- `layers`: character vector of layer names (NULL if single layer)
- `n_layers`: integer count of layers (NULL if single layer)

attribute information:

- `nvars`: character vector of nodal variable names
- `n_nvars`: integer count of nodal variables
- `dvars`: character vector of dyadic variable names
- `n_dvars`: integer count of dyadic variables

Author(s)

cassy dorff, shahryar minhas

`netify_node` *extract nodes layer from netify plot components*

Description

extracts the node (point) layer from a netify plot components object, allowing for manual plot construction and customization. nodes represent actors in the network and can have various aesthetic mappings like size, color, and shape.

Usage

```
netify_node(comp)
```

Arguments

`comp`	a netify_plot_components object returned from `plot(..., return_components = TRUE)`

Value

a custom object of class "netify_node" that can be added to a ggplot object using the + operator. the object contains the node layer with all its aesthetic mappings and data.

Author(s)

cassy dorff, shahryar minhas

See Also

`plot.netify`, `netify_edge`, `assemble_netify_plot`

Examples

```
# create a netify object
mat <- matrix(c(NA, 1, 0, 0, NA, 1, 1, 0, NA), 3, 3,
    dimnames = list(c("alice", "bob", "carol"),
                    c("alice", "bob", "carol")))
net <- new_netify(mat, symmetric = FALSE)

# get plot components
comp <- plot(net, return_components = TRUE)

# build custom plot with nodes
library(ggplot2)
ggplot() +
    netify_node(comp)
```

---

netify_predicates *Type predicates and convenience accessors for netify objects*

Description

these mirror the natural shape questions a user would ask: is this object bipartite? longitudinal? multilayer? how many actors does it have? they share a roxygen page with the small attribute-accessor helpers `is_binary()` and `nodal_data()`.

Usage

```
is_binary(x)

nodal_data(x)

is_bipartite(x)

is_bipartite_netify(x)

is_directed_netify(x)

is_longitudinal(x)

is_multilayer(x)

is_symmetric_netify(x)

n_actors(x)

n_periods(x)

n_layers(x)
```

Arguments

x	a netify object.

Value

is_binary() returns a single logical: TRUE when every off-diagonal cell of the underlying adjacency is 0, 1, or na. reads the cached "is_binary" attribute when available and falls back to probing the raw matrix / array / list.

nodal_data() returns the nodal-attribute data.frame stored on the netify object (the "nodal_data" attribute), or NULL if no nodal attributes have been attached. convenience wrapper so users do not have to remember the attr() call.

is_bipartite() returns a single logical. if igraph is loaded after netify, the bare is_bipartite() may be masked by igraph::is_bipartite() (which doesn't accept a netify). use is_bipartite_netify() (alias) or call as netify::is_bipartite() to avoid the collision.

is_bipartite_netify() is an alias for is_bipartite() that won't collide with igraph::is_bipartite() when both packages are attached.

is_directed_netify() returns a single logical. convenience alias for !istrue(attr(x, "symmetric")) that won't collide with igraph::is_directed().

is_longitudinal() returns a single logical.

is_multilayer() returns a single logical.

is_symmetric_netify() returns a single logical (or, for mixed-directedness multilayer, the per-layer vector).

n_actors() returns a single integer (number of unique actors across all periods / both modes; for bipartite a length-2 integer vector c(row, col)).

n_periods() returns a single integer (1 for cross-sectional).

n_layers() returns a single integer (1 for single-layer).

Author(s)

cassy dorff, shahryar minhas

---

netify_scale_labels *set scale labels for netify plots*

Description

provides a convenient way to set labels for aesthetic scales in netify plots. this function simplifies the process of labeling scales that may be spread across different layers (edges, nodes, text, labels).

Usage

```
netify_scale_labels(...)
```

Arguments

...	named arguments where the name is the aesthetic_component (e.g., "edge_alpha", "node_size", "edge_color") and the value is the label text to display in the legend

Details

this function provides a user-friendly interface for setting scale labels without needing to understand the complexity of ggnewscale. the naming convention is:

- `edge_*` for edge aesthetics (e.g., edge_color, edge_alpha)
- `node_*` or `point_*` for node aesthetics (both work)
- `text_*` for text label aesthetics
- `label_*` for boxed label aesthetics

Value

a custom object of class "netify_labels" that can be added to a netify plot using the + operator

Note

this function only works with plots created using netify's plot method. it will issue a warning if used with other ggplot objects.

Author(s)

cassy dorff, shahryar minhas

See Also

`plot.netify`

Examples

```
# set labels for different scales
mat <- matrix(c(NA, 2, 0, 0, NA, 1, 3, 0, NA), 3, 3,
    dimnames = list(c("alice", "bob", "carol"),
                    c("alice", "bob", "carol")))
net <- new_netify(mat, symmetric = FALSE, weight = "strength")

plot(net, edge_alpha_var = "strength") +
    netify_scale_labels(
        edge_alpha = "connection strength",
        node_size = "node degree" # node_* is converted to point_*
    )
```

`netify_text` *extract text layer from netify plot components*

Description

extracts the text label layer from a netify plot components object. text labels display actor names or other text annotations directly on the plot without background boxes.

Usage

```
netify_text(comp)
```

Arguments

`comp`	a netify_plot_components object returned from `plot(..., return_components = TRUE)`

Value

a custom object of class "netify_text" that can be added to a ggplot object using the + operator. the object contains the text layer with all its aesthetic mappings and data.

Author(s)

cassy dorff, shahryar minhas

See Also

`plot.netify`, `netify_label`, `assemble_netify_plot`

Examples

```
# create a netify object
mat <- matrix(c(NA, 1, 0, 0, NA, 1, 1, 0, NA), 3, 3,
    dimnames = list(c("alice", "bob", "carol"),
                    c("alice", "bob", "carol")))
net <- new_netify(mat, symmetric = FALSE)

# get plot components with text labels
comp <- plot(net, add_text = TRUE, return_components = TRUE)

# build custom plot with text
library(ggplot2)
ggplot() +
    netify_text(comp)
```

`netify_text_repel` *extract text_repel layer from netify plot components*

Description

extracts the text_repel layer from a netify plot components object. text repel labels display actor names with automatic repositioning to avoid overlaps, making them more readable in dense networks.

Usage

```
netify_text_repel(comp)
```

Arguments

`comp`	a netify_plot_components object returned from `plot(..., return_components = TRUE)`

Value

a custom object of class "netify_text_repel" that can be added to a ggplot object using the + operator. the object contains the text_repel layer with all its aesthetic mappings and data.

Author(s)

cassy dorff, shahryar minhas

See Also

`plot.netify`, `netify_text`, `assemble_netify_plot`

Examples

```
# create a netify object
mat <- matrix(c(NA, 1, 0, 0, NA, 1, 1, 0, NA), 3, 3,
    dimnames = list(c("alice", "bob", "carol"),
                    c("alice", "bob", "carol")))
net <- new_netify(mat, symmetric = FALSE)

# get plot components with text_repel
comp <- plot(net, add_text_repel = TRUE, return_components = TRUE)

# build custom plot with repelled text
library(ggplot2)
ggplot() +
    netify_text_repel(comp)
```

`netify_to_amen` *Convert netify objects to amen format*

Description

`netify_to_amen` (also available as `to_amen`) transforms netify network objects into the data structure required by the amen package for advanced network modeling. this enables the use of social relations models (srm), additive and multiplicative effects (ame) models, and other network regression approaches implemented in amen.

Usage

```
netify_to_amen(netlet, lame = FALSE)

to_amen(netlet, lame = FALSE)
```

Arguments

`netlet`	a netify object (class "netify") containing network data. must be a single-layer network. for multilayer networks, first extract individual layers using `subset_netify`.
`lame`	logical. controls the output format for longitudinal data: • `FALSE` (default): formats output for compatibility with the standard version of the amen package, which uses array structures. y is returned as a 3d array `[n_actors x n_actors x n_time]`, xdyad as a 4d array `[n_actors x n_actors x n_covariates x n_time]`, and xrow/xcol as 3d arrays `[n_actors x n_attributes x n_time]`. this requires constant actor composition across time. • `TRUE`: formats output for compatibility with the netify-verse version called lame, which supports longitudinal network modeling with time-varying actor compositions. y is returned as a list of t matrices (one per time period), xdyad as a list of t 3d arrays, and xrow/xcol as lists of t matrices. actor sets can vary across time periods, making this suitable for panels where countries enter/exit. this parameter is ignored for cross-sectional data.

Details

variable requirements:

- all nodal attributes must be numeric (integer or double)
- all dyadic attributes must be numeric or logical matrices
- character or factor variables must be converted before using this function
- missing values (na) are preserved and can be handled by amen's models

when to use each format:

- use `lame = FALSE` when:
  - actor composition is constant across time
- use `lame = TRUE` when:
  - actors enter/exit the network over time
  - want access to other features in lame

Value

the structure of the returned list depends on the data type and `lame` parameter:

for cross-sectional data or longitudinal with lame = FALSE (standard amen):

y network adjacency data as a numeric matrix or array:
- cross-sectional: matrix of dimensions `[n_actors x n_actors]`
- longitudinal: array of dimensions `[n_actors x n_actors x n_time]`

contains edge weights or binary indicators. missing edges are preserved as na.

xdyad dyadic covariates as an array, or NULL if none exist:
- cross-sectional: `[n_actors x n_actors x n_covariates]`
- longitudinal: `[n_actors x n_actors x n_covariates x n_time]`

each slice contains one dyadic covariate across all actor pairs.

xrow sender/row actor attributes as a matrix or array, or NULL if none exist:
- cross-sectional: `[n_actors x n_attributes]`
- longitudinal: `[n_actors x n_attributes x n_time]`

contains numeric attributes for actors when they act as senders.

xcol receiver/column actor attributes, structured identically to xrow. for symmetric networks, xcol is identical to xrow. for bipartite networks, contains attributes for the second mode.

for longitudinal data with lame = TRUE (lame package):

y a list of length t (time periods), where each element is an n x n relational matrix. actor sets can vary across time periods.

xdyad a list of length t, where each element is an n x n x pd array of dyadic covariates, or NULL if none exist

xrow a list of length t, where each element is an n x pr matrix of nodal row covariates, or NULL if none exist

xcol a list of length t, where each element is an n x pc matrix of nodal column covariates, or NULL if none exist

Note

the function performs several validation checks:

- ensures single-layer networks (multilayer not supported). for multilayer networks, first extract individual layers using `subset_netify` (e.g., `subset(net, layers = "trade")`).
- verifies all nodal and dyadic attributes are model-ready numeric inputs

- maintains actor ordering from the original netify object

for multilayer longitudinal models that require a 4d array `[n, n, p, t]`, see `netify_to_dbn` instead.

bipartite networks. `amen::ame()` does not accept rectangular y matrices; passing the output of `to_amen()` on a bipartite netify to `amen::ame()` will fail. use `netify_to_lame` (which sets `mode = "bipartite"` and targets `lame::ame()`) for bipartite networks instead.

Author(s)

ha eun choi, cassy dorff, colin henry, shahryar minhas

cassy dorff, shahryar minhas

Examples

```
# load example data
data(icews)

# create a netify object
net <- netify(
    icews[icews$year == 2010, ],
    actor1 = "i", actor2 = "j",
    symmetric = FALSE,
    weight = "verbCoop"
)

# convert to amen format (standard)
amen_data <- netify_to_amen(net)
names(amen_data) # y, xdyad, xrow, xcol


# for longitudinal data with time-varying composition
longit_net <- netify(
    icews,
    actor1 = "i", actor2 = "j", time = "year",
    symmetric = FALSE,
    weight = "verbCoop"
)

# convert to lame format
lame_data <- netify_to_amen(longit_net, lame = TRUE)
```

---

netify_to_dbn *Convert netify objects to dbn format*

Description

`netify_to_dbn` (also available as `to_dbn`) transforms netify network objects into the array format required by the dbn package for dynamic bilinear network models. this enables the use of longitudinal latent space models for multilayer and single-layer networks.

Usage

```
netify_to_dbn(netlet)

to_dbn(netlet)
```

Arguments

netlet	a netify object (class "netify") containing longitudinal network data. must be of type longit_array or longit_list. supports both single-layer and multilayer networks.

Details

the dbn package expects a 4d array with dimensions [n, n, p, t] where n is the number of actors, p is the number of relation types (layers), and t is the number of time periods.

supported netify types:

- longit_array: directly extracts or reshapes the underlying array
- longit_list: converts to array format first (actors are unioned across time, missing entries become na)

cross-sectional networks are not supported since dbn is designed for longitudinal data. bipartite networks are also not supported because dbn expects square actor-by-actor arrays.

variable requirements:

- all nodal attributes must be numeric (integer or double)
- all dyadic attributes must be numeric or logical matrices
- character or factor variables must be converted before using this function

Value

a list containing:

y network adjacency data as a 4d array of dimensions [n_actors, n_actors, n_layers, n_time]. for single-layer networks, n_layers = 1 and the third dimension is labeled with the weight variable name (or "edge_value" for binary networks). missing edges are preserved as na.

xdyad dyadic covariates as a 4d array of dimensions [n_actors, n_actors, n_covariates, n_time], or NULL if none exist.

xrow sender/row actor attributes as a 3d array of dimensions [n_actors, n_attributes, n_time], or NULL if none exist.

xcol receiver/column actor attributes as a 3d array of dimensions [n_actors, n_attributes, n_time], or NULL if none exist. for symmetric networks, xcol is identical to xrow.

Author(s)

shahryar minhas

cassy dorff, shahryar minhas

Examples

```
# load example data
data(icews)

# create two longitudinal networks
verbal_net <- netify(
    icews,
    actor1 = "i", actor2 = "j", time = "year",
    symmetric = FALSE,
    weight = "verbCoop",
    output_format = "longit_array"
)

material_net <- netify(
    icews,
    actor1 = "i", actor2 = "j", time = "year",
    symmetric = FALSE,
    weight = "matlCoop",
    output_format = "longit_array"
)

# create multilayer network
multi_net <- layer_netify(
    list(verbal_net, material_net),
    layer_labels = c("verbal", "material")
)

# convert to dbn format
dbn_data <- netify_to_dbn(multi_net)
dim(dbn_data$y) # [n_actors, n_actors, 2, n_years]


# single-layer also works
dbn_single <- netify_to_dbn(verbal_net)
dim(dbn_single$y) # [n_actors, n_actors, 1, n_years]
```

`netify_to_igraph` *Convert netify objects to igraph format*

Description

transforms netify network objects into igraph objects (also available as `netify_to_igraph`), preserving network structure and optionally including nodal and dyadic attributes as vertex and edge attributes.

Usage

```
netify_to_igraph(
  netlet,
  add_nodal_attribs = TRUE,
  add_dyad_attribs = TRUE,
  .quiet_na = FALSE
)

to_igraph(
  netlet,
  add_nodal_attribs = TRUE,
  add_dyad_attribs = TRUE,
  .quiet_na = FALSE
)
```

Arguments

`netlet`	a netify object containing network data. single-layer networks return a single igraph (or a list of igraphs for longitudinal input); multilayer networks return a named list keyed by layer, with each element itself an igraph or list of igraphs.
`add_nodal_attribs`	logical. if TRUE (default), includes nodal attributes from the netify object as vertex attributes in the igraph object. set to FALSE to create a network with structure only.
`add_dyad_attribs`	logical. if TRUE (default), includes dyadic attributes from the netify object as edge attributes in the igraph object. set to FALSE to exclude edge covariates.
`.quiet_na`	logical. internal flag to suppress the one-shot na-to-zero cli alert when called from inside netify's own summary path. default `FALSE`; do not set in user code.

Details

the conversion process handles different netify structures:

- **cross-sectional**: direct conversion to a single igraph object
- **longitudinal arrays**: internally converted to list format, then each time slice becomes a separate igraph object
- **longitudinal lists**: each time period converted to separate igraph object

for directed networks, the resulting igraph object will be directed. for undirected networks, the igraph object will be undirected. edge weights in the netify object become edge weights in igraph.

when longitudinal data includes actors that appear or disappear over time, each time period's igraph object will contain only the actors present in that period.

Value

an igraph object or list of igraph objects:

cross-sectional networks returns a single igraph object

longitudinal networks returns a named list of igraph objects, with names corresponding to time periods

multilayer networks returns a named list keyed by layer; each element is itself an igraph or (for longitudinal) a list of igraphs

the resulting igraph object(s) will have:

- vertices named according to actors in the netify object
- edge weights from the netify weight variable (if present)
- vertex attributes for each nodal variable (if add_nodal_attribs = TRUE)
- edge attributes for each dyadic variable (if add_dyad_attribs = TRUE)

Note

this function requires the igraph package to be installed.

Author(s)

ha eun choi, cassy dorff, colin henry, shahryar minhas

cassy dorff, shahryar minhas

Examples

```
# load example data
data(icews)

# example 1: cross-sectional network with attributes
icews_10 <- icews[icews$year == 2010, ]

# create netify object with attributes
dvars <- c("matlCoop", "verbConf", "matlConf")
nvars <- c("i_polity2", "i_log_gdp", "i_log_pop")

verbCoop_net <- netify(
    icews_10,
    actor1 = "i", actor2 = "j",
    symmetric = FALSE,
    weight = "verbCoop",
    dyad_vars = dvars,
    dyad_vars_symmetric = rep(FALSE, length(dvars)),
    nodal_vars = nvars
)

# convert to igraph
ig <- netify_to_igraph(verbCoop_net)

# examine the result
ig
igraph::vcount(ig) # number of vertices
```

```
igraph::ecount(ig) # number of edges
igraph::vertex_attr_names(ig) # nodal attributes
igraph::edge_attr_names(ig) # edge attributes

# access specific attributes
igraph::V(ig)$i_polity2 # polity scores
igraph::E(ig)$matlCoop # material cooperation


# example 2: longitudinal network
verbCoop_longit <- netify(
    icews,
    actor1 = "i", actor2 = "j", time = "year",
    symmetric = FALSE,
    weight = "verbCoop",
    dyad_vars = dvars,
    dyad_vars_symmetric = rep(FALSE, length(dvars)),
    nodal_vars = nvars
)

# convert to list of igraph objects
ig_list <- netify_to_igraph(verbCoop_longit)

# examine structure
length(ig_list) # number of time periods
names(ig_list) # time period labels

# access specific time period
ig_2002 <- ig_list[["2002"]]
ig_2002


# example 3: convert without attributes
ig_structure_only <- netify_to_igraph(
    verbCoop_net,
    add_nodal_attribs = FALSE,
    add_dyad_attribs = FALSE
)

# only network structure, no attributes
igraph::vertex_attr_names(ig_structure_only) # only "name"
igraph::edge_attr_names(ig_structure_only) # only "weight" (if present)
```

`netify_to_lame` *Convert a netify object to the format expected by* `lame::ame()`

Description

`netify_to_lame()` (also available as `to_lame()`) is a thin specialization of \link{netify_to_amen}

for the optional lame workflow. lame::ame() accepts the same y / xdyad / xrow / xcol skeleton as amen::ame() but adds two things netify users care about:

Usage

```
netify_to_lame(
  netlet,
  lame = FALSE,
  family = NULL,
  pad = TRUE,
  fit_method = c("gibbs", "als"),
  bootstrap = 0L
)

to_lame(
  netlet,
  lame = FALSE,
  family = NULL,
  pad = TRUE,
  fit_method = c("gibbs", "als"),
  bootstrap = 0L
)
```

Arguments

netlet	a netify object.
lame	logical. as in netify_to_amen: pass TRUE when actor composition varies over time. default FALSE.
family	optional character. ame family to use. if NULL (default), inferred from the netify: "binary" for binary networks, "normal" for weighted, with a once-per-session info message naming the choice.
pad	logical. when lame = TRUE and the per-period list is returned (the function never actually pads in place; it always returns a list), controls whether the once-per-session info message points users at the padding + lame::lame() snippet baked into nl$ame_call. default TRUE (emit the message); set FALSE to silence it.
fit_method	one of "gibbs" (default – mcmc posterior via lame::ame() / lame::lame()) or "als" (fast alternating- least-squares point estimate via lame::ame_als()). for bipartite + binary networks where mcmc is slow, als with bootstrap > 0 gives a fast point estimate plus parametric/block bootstrap uncertainty intervals in a single call. the choice only affects the generated ame_call snippet – the y/xdyad/ xrow/xcol payload is identical.
bootstrap	integer. number of bootstrap replicates for als uncertainty intervals. ignored when fit_method = "gibbs" (mcmc draws are the uncertainty representation there). default 0 (no bootstrap). pass 200 for a reasonable interval estimate.

Details

- rectangular y for **bipartite** networks via mode = "bipartite" – the standard amen::ame() rejects this.
- ragged longitudinal panels via a generated padding snippet – amen requires constant actor composition.

this function wraps \link{netify_to_amen} and:

1. picks a family default appropriate to the netify ("binary" for binary nets, "normal" for weighted),
2. suggests the correct mode argument for lame::ame() ("unipartite" / "bipartite"),
3. for lame = TRUE longitudinal output, emits a copy-paste padding + lame::lame() snippet in the returned ame_call slot. the function itself does **not** pad the list – run the snippet (or the literal call) to do that step yourself.

bipartite + binary: the case to_amen() **cannot fit.** for a bipartite weighted-binary netify (e.g., person x event attendance, 0/1), this is the exact pipeline:

```
bp <- netify(df, actor1 = "person", actor2 = "event",
mode = "bipartite", weight = "attended")
nl <- to_lame(bp) # auto-detects binary
fit <- lame::ame(
y = nl$y, xrow = nl$xrow, xcol = nl$xcol,
mode = nl$mode, family = nl$family,
nscan = 1000, burn = 500
)
# uncertainty: posterior intervals are in fit$beta / fit$vc
```

ragged longitudinal panels. when lame = TRUE and actors enter / exit the network, this function:

1. builds the per-period list via netify_to_amen(lame=TRUE),
2. bakes a pad-then-fit snippet into nl$ame_call that the user runs to materialize the 3d [n, n, t] array and fit lame::lame(). unipartite snippets use lame::list_to_array(); bipartite snippets pad rectangular row-by-column arrays directly.

the returned y / xdyad / xrow / xcol are always the per-period list – pad only controls whether the once-per- session info message reminding the user of the snippet fires.

Value

a list with the same shape as netify_to_amen output, plus two helpful extras:

mode character: "unipartite" or "bipartite". pass this directly to lame::ame(mode = .).

family character: the suggested family ("binary" or "normal"). pass to lame::ame(family = .).

ame_call character: a literal lame::ame() call string the user can copy-paste.

Author(s)

shahryar minhas

cassy dorff, shahryar minhas

See Also

netify_to_amen (the underlying converter), netify_to_statnet, netify_to_dbn, netify_to_igraph.

---

netify_to_statnet *Convert netify objects to statnet network format*

Description

transforms netify network objects into statnet's network objects (also available as netify_to_network, to_statnet, and to_network), providing access to the extensive statistical modeling capabilities of the statnet suite, including ergms (exponential random graph models), descriptive statistics, and network visualization tools.

Usage

```
netify_to_statnet(netlet, add_nodal_attribs = TRUE, add_dyad_attribs = TRUE)

netify_to_network(netlet, add_nodal_attribs = TRUE, add_dyad_attribs = TRUE)

to_statnet(netlet, add_nodal_attribs = TRUE, add_dyad_attribs = TRUE)

to_network(netlet, add_nodal_attribs = TRUE, add_dyad_attribs = TRUE)
```

Arguments

netlet	a netify object containing network data. cross-sectional, longitudinal, and multilayer netlets are all supported; multilayer inputs are dispatched layer-by-layer and the layer results are returned in a named list keyed by layer.
add_nodal_attribs	logical. if TRUE (default), includes nodal attributes from the netify object as vertex attributes in the network object. set to FALSE to create a network with structure only.
add_dyad_attribs	logical. if TRUE (default), includes dyadic attributes from the netify object as edge attributes in the network object. each dyad variable is attached in two places on the resulting network: as a network-level attribute under its original name (the full $n \times n$ matrix) and as a per-edge attribute under <var>_e. the trailing _e disambiguates the per-edge edgelist from the network-level matrix. for ergm::edgecov() pass the *original* (matrix) name, not the _e alias, since edgecov() reads a network-level matrix attribute. the _e per-edge attribute is exposed for descriptive use (e.g. coloring edges in network::plot.network).

Details

the conversion process handles different netify structures:

- **cross-sectional**: direct conversion to a single network object
- **longitudinal arrays**: internally converted to list format, then each time slice becomes a separate network object
- **longitudinal lists**: each time period converted to separate network object

the statnet network format stores networks as an edgelist with attributes, making it memory-efficient for sparse networks. all nodal and dyadic attributes from the netify object are preserved and can be used in subsequent ergm modeling or network analysis.

for longitudinal data, each time period results in an independent network object. this format is suitable for discrete-time network analysis or pooled ergm estimation across time periods.

Value

a network object or list of network objects:

cross-sectional networks returns a single network object

longitudinal networks returns a named list of network objects with class `c("network.list", "list")`, names corresponding to time periods. the `network.list` class is the format that **tergm** cmle (`tergm(nl ~ form(.) + persist(.), estimate = "cmle", times = seq_along(nl))`) and `stergm` expect directly, so the output is plug-and-play for longitudinal ergms.

multilayer networks returns a named list of (per-time) network objects keyed by layer name

the resulting network object(s) will have:

- vertices named according to actors in the netify object
- edge weights from the netify weight variable stored as "weight" attribute
- vertex attributes for each nodal variable (if add_nodal_attribs = TRUE)
- edge attributes for each dyadic variable (if add_dyad_attribs = TRUE); per-edge attributes carry the _e suffix, network-level attributes keep the original name
- proper directedness based on the symmetric parameter of the netify object
- a `netify_na_cols` attribute (character vector) listing nodal columns that carry `na`s; pass this directly to `drop_na_actors()` to drop the offending actors before fitting ergm formulas that reference those columns

Note

this function requires the network package (part of statnet) to be installed.

for ergm modeling, the ergm package (also part of statnet) should be loaded after creating the network objects.

Author(s)

ha eun choi, cassy dorff, colin henry, shahryar minhas

cassy dorff, shahryar minhas

Examples

```
data(classroom_edges)
data(classroom_nodes)

net <- netify(
    classroom_edges,
    actor1 = "from", actor2 = "to",
    symmetric = TRUE,
    nodal_data = classroom_nodes,
    missing_to_zero = TRUE
)

ntwk <- netify_to_statnet(net)

# examine the result
ntwk
network::network.size(ntwk) # number of vertices
network::network.edgecount(ntwk) # number of edges
network::list.vertex.attributes(ntwk) # nodal attributes
network::get.vertex.attribute(ntwk, "gender") # gender by student

# check network properties
network::is.directed(ntwk)
network::has.loops(ntwk)

# longitudinal example
classroom_panel <- rbind(
    transform(classroom_edges[1:12, ], wave = 1),
    transform(classroom_edges[13:24, ], wave = 2)
)

longit_net <- netify(
    classroom_panel,
    actor1 = "from", actor2 = "to", time = "wave",
    symmetric = TRUE,
    missing_to_zero = TRUE
)

ntwk_list <- netify_to_statnet(longit_net)

# examine structure
length(ntwk_list) # number of time periods
names(ntwk_list) # time period labels


# use with ergm for modeling (requires ergm package)
library(ergm)
model <- ergm(ntwk ~ edges + nodematch("gender"))
```

`netify_workflows` *Common netify workflows – index of patterns*

Description

this is a documentation-only topic ("?netify_workflows") that indexes the standard pipelines netify users actually run, with the function calls in order. new users opening ?netify see the constructor; opening this topic sees the recipes.

from a friendship edgelist

```
library(netify)
net <- netify(friends, actor1 = "from", actor2 = "to",
symmetric = TRUE, nodal_data = people)
summary(net)
plot(net, node_color_by = "gender")
```

a ready-to-run worked example ships with the package:

```
data(classroom_edges); data(classroom_nodes)
net <- netify(classroom_edges, actor1 = "from", actor2 = "to",
symmetric = TRUE, nodal_data = classroom_nodes)
summary(net)
plot(net, node_color_by = "gender", node_size_by = "gpa")
```

from a bipartite (two-mode) edgelist

a bipartite edgelist links *two different kinds of things* (e.g. companies and jurisdictions, movies and actors, students and classes). build with mode = "bipartite" and pass row-mode / col-mode covariates separately.

```
net <- netify(filings, actor1 = "company", actor2 = "jurisdiction",
mode = "bipartite", weight = "n_filings")
net <- add_node_vars(net, company_meta,
actor = "company",
node_vars = c("industry", "hq"))
net <- add_node_vars(net, jurisdiction_meta,
actor = "jurisdiction",
node_vars = "tax_haven")
plot(net, style = "heatmap")
```

from a longitudinal dyadic event dataset (icews-style)

```
net <- netify(icews, actor1 = "i", actor2 = "j", time = "year",
symmetric = FALSE, weight = "verbCoop")
net <- add_node_vars(net, country_panel,
actor = "actor", time = "year",
node_vars = c("polity", "gdp_log"))
```

```
s <- summary(net)
plot(s)
```

multilayer (one call per layer)

```
# build one netify per layer, then combine with layer_netify()
channels <- sort(unique(slack_long$channel))
nets <- lapply(channels, function(ch) {
  netify(slack_long[slack_long$channel == ch, ],
         actor1 = "from", actor2 = "to",
         time = "month", weight = "n_messages")
})
ml <- layer_netify(setNames(nets, channels))
```

to ergm (`statnet`)

```
ntwk <- to_statnet(net)
library(ergm)
fit <- ergm(ntwk ~ edges + nodematch("gender") + nodecov("gpa"))
summary(fit)
```

to longitudinal ergm (`tergm 4.x`)

```
nl <- to_statnet(longit_net) # network.list
library(tergm)
fit <- tergm(nl ~ form(~ edges + nodematch("gender")) +
persist(~ edges + mutual),
estimate = "cmle", times = seq_along(nl))
```

to ame (gibbs / mcmc posterior)

```
lm_in <- to_lame(net) # auto-picks family from binary/weighted
library(lame)
# copy-paste the printed `lm_in$ame_call`:
fit <- lame::ame(y = lm_in$y, xdyad = lm_in$xdyad,
xrow = lm_in$xrow, xcol = lm_in$xcol,
family = "binary", nscan = 1000, burn = 500)
```

to ame (binary bipartite + als + bootstrap uncertainty)

```
# fast point estimate + parametric bootstrap cis in one call
lm_in <- to_lame(net_bp, fit_method = "als", bootstrap = 200)
library(lame)
fit <- lame::ame_als(y = lm_in$y, mode = "bipartite",
family = "binary", bootstrap = 200l)
confint(fit) # bootstrap cis
pred <- from_lame_fit(fit, value = "prob")
plot(pred, style = "heatmap")
```

NULL-model hypothesis test

```
# is observed transitivity surprising vs erdos-renyi at this density?
compare_to_null(net,
fn = function(n) c(transitivity = summary(n)$transitivity),
n_sim = 200, model = "erdos_renyi", seed = 1)
```

bootstrap any custom statistic

```
my_stat <- function(net) {
sa <- summary_actor(net, stats = "all")
c(max_betw = max(sa$betweenness, na.rm = TRUE))
}
bootstrap_netlet(net, fn = my_stat, n_boot = 200, seed = 1)
```

very large n (~10k+ actors)

netify stores adjacencies densely, so an n x n matrix consumes 8 * n^2 bytes (1.7 gb at n = 15k, 20 gb at n = 50k) per snapshot. the recipes below avoid the dense bottleneck where possible.

```
# 1. build directly from an edgelist data.frame -- skip any
#    pre-built sparse / dense matrix intermediate.
net <- netify(edges, actor1 = "from", actor2 = "to",
              weight = "n", symmetric = FALSE)

# 2. use the fast actor-summary path -- skips closeness /
#    betweenness / eigen / hits, which dominate wall-clock at
#    large n. netify auto-promotes at n >= netify.fast_threshold
#    (default 1500); tune via options(netify.fast_threshold = ...).
sa <- summary_actor(net)            # auto-promotes

# 3. for community detection / shortest paths / clustering, hand
#    off to igraph. its c backend handles 50k+ nodes comfortably.
ig <- to_igraph(net)
cl <- igraph::cluster_louvain(ig)

# 4. when you need a sparse object for downstream code (e.g.
#    spectral methods, irlba, glmnet), exit via to_igraph() and
#    grab the sparse adjacency from there.
ig <- to_igraph(net)
m  <- igraph::as_adjacency_matrix(ig, sparse = TRUE)  # requires matrix

# 5. if you must accept a pre-built sparse matrix, density < 1
#    with n > 5,000 aborts to protect against accidental
#    gigabyte allocations -- pass force_dense = TRUE to override.
net <- netify(m_sparse, force_dense = TRUE)
```

round-trip to / from other formats

```
# to igraph and back
```

```
ig <- to_igraph(net)
back <- netify(ig) # preserves directedness + weight

# to long data frame
edges <- unnetify(net, remove_zeros = TRUE)
tb <- tibble::as_tibble(net) # same thing, tibble class

# to base matrix
m <- as.matrix(net) # strips netify class
```

stacking

```
# combine two cross-sec into a 2-period longit
combined <- bind_netifies(n_2020, n_2021, names = c("2020", "2021"))
```

Author(s)

cassy dorff, shahryar minhas

See Also

`netify()` for the constructor; `summary.netify()` for graph-level stats; `plot.netify()` for visualization; `to_statnet()` / `to_amen()` / `to_lame()` / `to_igraph()` for handoff to modeling packages; `bootstrap_netlet()` / `compare_to_null()` / `simulate.netify()` for inference; and `bind_netifies()` for combining.

---

new_netify *low-level constructor for netify objects*

Description

`new_netify` (also available as `new_netlet`) is a low-level constructor that creates netify objects from raw matrix, array, or list data structures. this function automatically detects network properties and sets appropriate attributes, making it useful for converting existing network data into the netify format.

Usage

```
new_netify(data, ...)
```

Arguments

`data`	a network data structure to convert: • **matrix**: creates a cross-sectional netify object • **3d array**: creates a longitudinal array netify object (dimensions: actors x actors x time)

- **list of matrices**: creates a longitudinal list netify object (useful for time-varying actor composition)

... additional parameters to set as attributes on the netify object. common parameters include:

- `symmetric`: logical indicating if network is undirected
- `mode`: "unipartite" or "bipartite"
- `weight`: name of the edge weight variable
- `diag_to_NA`: whether to set diagonal to na
- `missing_to_zero`: whether to treat missing edges as zeros
- `nodal_data`: data frame of node attributes
- `dyad_data`: dyadic attributes (see netify documentation)

if not provided, these properties are automatically detected from the data.

Details

automatic property detection:

when properties are not explicitly provided, `new_netify` intelligently detects:

- **symmetry**: checks if matrix equals its transpose
- **mode**: infers unipartite/bipartite from dimensions and actor names
- **edge weights**: detects binary (0/1) vs. weighted networks
- **diagonal treatment**: checks if diagonal contains all nas
- **missing values**: determines if nas exist off-diagonal
- **actor composition**: for longitudinal data, detects if actors remain constant or vary over time

naming conventions:

if row/column names are not provided:

- unipartite networks: actors named "a1", "a2", ...
- bipartite networks: row actors "r1", "r2", ...; column actors "c1", "c2", ...
- time periods: named as "1", "2", ... if not specified

longitudinal data handling:

for longitudinal networks:

- arrays assume constant actor composition across time
- lists allow for time-varying actor composition
- each time slice in a list becomes a separate cross-sectional netify object
- properties are detected across all time periods (e.g., symmetric if all time slices are symmetric)

Value

a netify object with class "netify" and appropriate structure:

- for matrices: a single netify matrix with netify_type = "cross_sec"
- for arrays: a netify array with netify_type = "longit_array"
- for lists: a netify list with netify_type = "longit_list", where each element is itself a netify object

all netify objects include automatically detected or user-specified attributes for network properties, making them ready for use with netify functions.

Note

this is a low-level constructor primarily intended for package developers or advanced users. most users should use the higher-level `netify()` function, which provides more validation and preprocessing.

the function does not support multilayer networks directly. to create multilayer networks, create separate netify objects and combine them with `layer_netify()`.

while the function attempts to detect network properties automatically, explicitly providing these parameters is recommended for clarity.

Author(s)

cassy dorff, shahryar minhas

---

peek *Preview subsets of network data from netify objects*

Description

peek provides a convenient way to inspect portions of network data stored in netify objects. rather than displaying entire networks (which can be overwhelming for large datasets), this function allows you to examine specific actors, time periods, or layers for data exploration, verification, and debugging.

Usage

```
peek(
  netlet,
  actors = NULL,
  from = 3,
  to = 3,
  time = 1,
  layers = NULL,
  drop_dimensions = TRUE
)
```

Arguments

`netlet`	a netify object to preview. can be cross-sectional, longitudinal (array or list format), and/or multilayer.
`actors`	character vector of actor names or numeric indices to subset. selects these actors as both senders and receivers (extracts subgraph among these actors). overridden by `from` and `to` if specified.
`from`	specifies which actors to display as senders of ties (row actors in the adjacency matrix). can be: • **single number**: shows the first n actors (e.g., `from = 5` displays actors 1-5) • **numeric vector**: shows specific positions (e.g., `from = c(1,3,5)` displays the 1st, 3rd, and 5th actors) • **character vector**: shows named actors (e.g., `from = c("usa", "china")` displays these specific countries) • **NULL**: shows all row actors (default is 3) in bipartite networks, from represents actors in the first mode.
`to`	specifies which actors to display as receivers of ties (column actors in the adjacency matrix). accepts the same input types as `from`. in bipartite networks, columns represent actors in the second mode. default is 3.
`time`	for longitudinal networks, specifies which time periods to display. can be: • **single number**: shows the nth time period (e.g., `time = 1` shows the first time period) • **numeric vector**: shows specific time indices (e.g., `time = c(1,5,10)`) • **character vector**: shows named time periods (e.g., `time = c("2002", "2006")` displays these specific years) • **NULL**: shows all time periods default is 1 (first time period only). ignored for cross-sectional networks.
`layers`	for multilayer networks, specifies which layer(s) to display. must be a character vector matching layer names in the netify object (e.g., `layers = c("trade", "alliance")`). for single-layer networks, this parameter is ignored. default is NULL (shows all layers).
`drop_dimensions`	logical. whether to drop singleton array dimensions in the returned preview. default TRUE.

Details

purpose and design

`peek` is designed as a lightweight data exploration tool. unlike `subset_netify`, which creates new netify objects with all attributes preserved, `peek` returns only the raw network data for quick inspection. this makes it ideal for:

- verifying data structure and content
- checking specific relationships
- debugging data issues

- quick visual inspection of network patterns

understanding network directions

in directed networks:

- **from** represents actors sending ties (out-ties)
- **to** represents actors receiving ties (in-ties)
- cell `[i, j]` contains the tie from actor i to actor j

for example, if cell `["usa", "china"]` = 5, this means usa sends a tie of strength 5 to china.

smart selection behavior

the function includes several convenience features:

- single numbers are expanded to ranges (e.g., `from = 5` becomes first 5 actors)
- out-of-bounds indices are silently ignored (no errors during exploration)
- character names are matched to actor labels
- dimension reduction: if only one time period or layer is selected, that dimension is dropped from the output

Value

returns a subset of the raw network data (without netify attributes):

cross-sectional networks
- single layer: returns a matrix with selected rows and columns
- multilayer: returns a 3d array (rows x columns x layers)

longitudinal networks
- array format: returns an array with dimensions depending on selection
- list format: returns a list of matrices, one per selected time period

all returned objects preserve dimension names (actor names, time labels, layer names) for easy interpretation. single dimensions are automatically dropped.

Note

important distinctions:

- use `peek` for quick data inspection (returns raw matrices)
- use `subset_netify` to create new netify objects with attributes
- use `get_raw` to extract all raw data from a netify object

when multiple layers are present and no layer selection is specified, all layers are returned with a warning message to remind you about the multilayer structure.

Author(s)

cassy dorff, shahryar minhas

Examples

```
# load example data
data(icews)

# example 1: basic usage with cross-sectional network
icews_10 <- icews[icews$year == 2010, ]
net <- netify(
    icews_10,
    actor1 = "i", actor2 = "j",
    weight = "verbCoop"
)

# default: see first 3 actors (both sending and receiving)
peek(net)

# see first 5 senders and first 5 receivers
peek(net, from = 5, to = 5)

# example 2: specific actors by name
# see ties from us and china to russia, india, and brazil
peek(net,
    from = c("united states", "china"),
    to = c("russia", "india", "brazil")
)

# use actors parameter to see subgraph
peek(net,
    actors = c("united states", "china", "russia")
)

# example 3: longitudinal network
net_longit <- netify(
    icews,
    actor1 = "i", actor2 = "j", time = "year",
    weight = "matlConf"
)

# see first 5 actors in specific years
peek(net_longit,
    from = 5, to = 5,
    time = c("2002", "2006", "2010")
)

# see all actors in year 2010
peek(net_longit,
    from = NULL, to = NULL,
    time = "2010"
)

# example 4: using numeric indices
# see specific positions in the network
peek(net,
```

```
    from = c(1, 3, 5, 7), # 1st, 3rd, 5th, 7th senders
    to = 1:10
) # first 10 receivers

# example 5: quick inspection patterns
# see who usa interacts with
peek(net, from = "united states", to = 10) # usa's ties to first 10 countries
peek(net, from = 10, to = "united states") # first 10 countries' ties to usa
```

`pivot_dyad_to_network` *Pivot a dyadic variable to become the network*

Description

`pivot_dyad_to_network` swaps a dyadic attribute with the main network in a netify object. this is useful when you want to analyze a different relationship type that was stored as a dyadic attribute. for example, if your network represents trade relationships but you have fdi stored as a dyadic attribute, you can pivot to make fdi the main network.

Usage

```
pivot_dyad_to_network(
  netlet,
  dyad_var,
  make_network_dyad_var = TRUE,
  network_var_name = NULL,
  symmetric = NULL,
  weight_type = NULL,
  diag_to_NA = NULL,
  missing_to_zero = NULL
)
```

Arguments

`netlet`	a netify object containing the network and dyadic attributes
`dyad_var`	character string naming the dyadic variable to become the new network. must exist in the netlet's dyad_data attribute.
`make_network_dyad_var`	logical. if TRUE (default), the current network will be preserved as a dyadic attribute.
`network_var_name`	character string specifying the name for the old network when converted to a dyadic attribute. if NULL (default), uses the current weight attribute name if available, otherwise "old_network".
`symmetric`	logical or NULL. specifies whether the new network should be treated as symmetric. if NULL (default), attempts to detect from the dyadic variable's symmetry setting or data structure.

`weight_type` character string describing the type of weight for the new network (e.g., "trade_volume", "fdi_amount"). if NULL (default), uses the dyad_var name.

`diag_to_NA` logical. whether to set diagonal values to na in the new network. if NULL (default), inherits from the original netlet.

`missing_to_zero` logical. whether to treat missing values as zeros in the new network. if NULL (default), inherits from the original netlet.

Details

the function handles different netify types appropriately:

- for cross-sectional networks: performs a simple matrix swap
- for longitudinal arrays: swaps matrices across all time periods
- for longitudinal lists: swaps matrices for each time period
- for multilayer networks: swaps within each layer

when the new network has different properties than the original (e.g., different symmetry or weight type), the function updates the netify attributes accordingly. for bipartite networks, the new network is always treated as asymmetric.

if the dyadic variable was originally specified with symmetry information via `add_dyad_vars()`, that information is used unless overridden by the symmetric parameter.

Value

a netify object with the dyadic variable as the main network and (optionally) the old network preserved as a dyadic attribute. all other attributes and dyadic variables are preserved.

Author(s)

shahryar minhas

Examples

```
# load example data
data(icews)

# create a netify object with verbal cooperation as the main network
# and material cooperation as a dyadic attribute
icews_10 <- icews[icews$year == 2010, ]

net <- netify(
    icews_10,
    actor1 = "i", actor2 = "j",
    symmetric = FALSE,
    weight = "verbCoop"
)

net <- add_dyad_vars(
    net,
```

```
    icews_10,
    actor1 = "i", actor2 = "j",
    dyad_vars = "matlCoop",
    dyad_vars_symmetric = FALSE
)

# check the current network
print(net)

# pivot to make material cooperation the main network
net_pivoted <- pivot_dyad_to_network(
    net,
    dyad_var = "matlCoop",
    network_var_name = "verbCoop"
)

# the main network is now material cooperation
print(net_pivoted)

# the old network (verbal cooperation) is preserved as a dyadic attribute
attr(net_pivoted, "dyad_data")[["1"]][["verbCoop"]][1:5, 1:5]
```

`plot.netify` *Plotting method for netify objects*

Description

creates customizable network visualizations from netify objects using ggplot2. supports cross-sectional and longitudinal networks with extensive options for mapping network attributes to visual properties.

Usage

```
## S3 method for class 'netify'
plot(x, auto_format = TRUE, ...)
```

Arguments

`x`	a 'netify' object containing network data to visualize.
`auto_format`	logical. if TRUE (default), automatically adjusts plot parameters based on network characteristics such as size, density, and structure. this includes "intelligent" defaults for: • node size (smaller for larger networks) • edge transparency (lower for denser networks) • text labels (enabled for small networks) • curved edges (for small dense networks) • isolate removal (for large networks)

set to FALSE to disable all automatic formatting. individual parameters can still be overridden even when auto_format is TRUE.

... additional arguments controlling plot appearance:

Details

naming conventions:

the function supports two naming styles for parameters:

- **recommended**: use `node_*` for node attributes and `*_by` for variable mappings (e.g., `node_size_by = "degree"`)
- **legacy**: use `point_*` for nodes and `*_var` for variables (e.g., `point_size_var = "degree"`)

default behaviors:

- for weighted networks, edge transparency maps to weight by default
- for directed networks, arrows are added automatically
- for longitudinal networks, time periods are shown as facets
- isolates are removed by default (set `remove_isolates = FALSE` to keep)

customization tips:

- use `mutate_weight` to handle skewed weight distributions
- combine fixed and variable aesthetics (e.g., fixed color with variable size)
- add ggplot2 layers after the plot call for further customization
- use `select_text` for selective labeling in dense networks

Value

a ggplot2 object that can be further customized with additional layers, scales, themes, etc. for longitudinal networks, includes facets for each time period.

if `return_components = TRUE`, returns a list of plot components that can be manually assembled or modified.

layout parameters

`layout` character string specifying the igraph layout algorithm. options include: `"nicely"` (default), `"fr"` (fruchterman-reingold), `"kk"` (kamada-kawai), `"circle"`, `"star"`, `"grid"`, `"tree"`, `"bipartite"` (for bipartite networks), `"randomly"`, and others. for ego networks, additional options are available: `"radial"` (ego-centric with optional grouping) and `"concentric"` (ego at center with alters in rings). see `get_node_layout` and `get_ego_layout` for full details.

`point_layout` optional data.frame or list of data.frames containing pre-computed node positions with columns 'actor', 'x', and 'y'. overrides `layout` if provided.

`static_actor_positions` logical. for longitudinal networks, should node positions remain constant across time? default is `FALSE`.

`which_static` integer. when `static_actor_positions = TRUE`, which time period's layout to use as template? if `NULL` (default), creates composite layout from all time periods.

`seed` integer for reproducible layouts. default is 6886.

display control

`add_edges` logical. display edges? default is `TRUE`.

`add_points` logical. display nodes as points? default is `TRUE`.

`add_text` logical. add text labels to nodes? default is `FALSE`.

`add_text_repel` logical. add text labels with automatic repositioning to avoid overlaps? default is `FALSE`. when `TRUE`, overrides `add_text`. uses ggrepel for positioning.

`add_label` logical. add boxed labels to nodes? default is `FALSE`.

`add_label_repel` logical. add boxed labels with automatic repositioning to avoid overlaps? default is `FALSE`. when `TRUE`, overrides `add_label`. uses ggrepel for positioning.

`remove_isolates` logical. remove unconnected nodes? default is `TRUE`.

`curve_edges` logical. use curved edges? default is `FALSE`.

`use_theme_netify` logical. apply netify theme? default is `TRUE`.

`facet_type` character. for multilayer longitudinal networks, controls faceting style: `"grid"` (default) creates a 2d grid with time x layer, `"wrap"` creates wrapped facets with combined time-layer labels.

`facet_ncol` integer. number of columns for facet_wrap layouts. only used when `facet_type = "wrap"` or for single-dimension faceting.

`rescale_edge_weights` logical. for multilayer networks, should edge weights be rescaled to a common 0-1 range across all layers? this is useful when layers have very different weight scales. default is `FALSE`.

subsetting parameters

`node_filter` a formula, quoted expression, or function to filter nodes. the expression can reference any nodal attribute. for example: `node_filter = ~ degree_total > 5` shows only nodes with total degree greater than 5.

`edge_filter` a formula, quoted expression, or function to filter edges. the expression can reference any edge attribute, including 'weight' for weighted networks. for example: `edge_filter = ~ weight > 0.5` shows only edges with weight greater than 0.5.

`time_filter` for longitudinal networks, a vector of time periods to include in the plot. can be numeric indices or character labels matching the time dimension. if NULL (default), all time periods are plotted. for cross-sectional networks, this parameter is ignored.

node aesthetics

fixed aesthetics (same for all nodes):

`node_size` **or** `point_size` numeric. size of all nodes.

`node_color` **or** `point_color` color of node borders.

`node_fill` **or** `point_fill` fill color of nodes (note that fill will only work with certain shapes).

`node_shape` **or** `point_shape` shape of nodes (see `?pch`).

`node_alpha` **or** `point_alpha` transparency (0-1).

`node_stroke` **or** `point_stroke` width of node borders.

variable aesthetics (mapped to data):

`node_size_by` **or** `point_size_var` column name for size mapping.

`node_color_by` **or** `point_color_var` column name for border color.

`node_fill_by` **or** `point_fill_var` column name for fill color (note that fill will only work with certain shapes).

`node_shape_by` **or** `point_shape_var` column name for shape.

`node_alpha_by` **or** `point_alpha_var` column name for transparency.

edge aesthetics

fixed aesthetics:

`edge_color` color for all edges. default is "black".

`edge_linewidth` width for all edges. default is 0.5.

`edge_linetype` line type (1=solid, 2=dashed, etc.).

`edge_alpha` transparency (0-1).

`edge_curvature` curvature amount when `curve_edges = TRUE`.

`edge_arrow` arrow specification for directed networks. example: `arrow(length = unit(0.2, "cm"))`.

variable aesthetics:

`edge_color_by` **or** `edge_color_var` column name for color mapping.

`edge_linewidth_by` **or** `edge_linewidth_var` column name for width.

`edge_linetype_by` **or** `edge_linetype_var` column name for line type.

`edge_alpha_by` **or** `edge_alpha_var` column name for transparency. for weighted networks, defaults to the weight variable if not specified.

text and label options

selective labeling:

`select_text` character vector of node names to show as text. when used, text labels will automatically use `geom_text_repel` to avoid overlaps.

`select_text_display` alternative text to display (same length as `select_text`).

`select_label` character vector of node names to show with boxes. when used, labels will automatically use `geom_label_repel` to avoid overlaps.

`select_label_display` alternative labels (same length as `select_label`).

text aesthetics:

`text_size` fixed size for all text. default is 3.88.

`text_color` fixed color for all text. default is "black".

`text_alpha` fixed transparency for text.

`text_size_by` variable to map to text size.

text_color_by variable to map to text color.

label (boxed text) aesthetics have similar parameters with label_ prefix.

text repel parameters (when add_text_repel = TRUE):

text_repel_force force of repulsion between overlapping text. default is 1.

text_repel_max_overlaps maximum number of overlaps to tolerate. default is 10.

text_repel_box_padding padding around text. default is 0.25.

text_repel_point_padding padding around points. default is 0.

text_repel_segment_color color of connecting segments. default is "grey50".

label repel parameters (when add_label_repel = TRUE):

similar to text_repel but with label_repel_ **prefix**

label_repel_label_padding padding around label boxes. default is 0.25.

label_repel_label_r radius of label box corners. default is 0.15.

scale labels

customize legend titles:

node_size_label **or** point_size_label legend title for size.

node_color_label **or** point_color_label legend title for color.

edge_alpha_label legend title for edge transparency.

edge_color_label legend title for edge color.

highlighting parameters

highlight character vector of node names to highlight with different colors. non-highlighted nodes will be colored grey. highlighted nodes can also be automatically enlarged if highlight_size_increase is greater than 1.

highlight_color named vector of colors for highlighted nodes. if NULL, uses default distinct colors (red, blue, green for up to 3 nodes, or a color palette for more). names should match the values in the highlight parameter. example: c('usa' = 'blue', 'china' = 'red', 'russia' = 'green').

highlight_label title for the highlight legend. default is "highlighted".

highlight_size_increase numeric factor(s) to increase size of highlighted nodes. can be a single value (applied to all highlighted nodes) or a vector of length length(highlight) + 1 where each value corresponds to a highlighted node and the last value applies to "other" nodes. default is 1 (no size increase). example: c(3, 1, 1, 0.5) for 3 highlighted nodes where the first is 3x larger, the next two are normal size, and all others are half size.

show_other_in_legend logical. include "other" category in legend? default is FALSE. when FALSE, only highlighted nodes appear in the legend.

ego layout parameters

for ego networks (created with ego_netify), additional layout options control the ego-centric visualization:

`ego_group_by` character string specifying a nodal attribute to use for grouping alters in ego layouts. for "radial" layout, creates sectors. for "concentric" layout, determines ring assignment.

`ego_order_by` character string specifying a nodal attribute to use for ordering alters within groups or rings. common options include "degree_total".

`ego_weight_to_distance` logical. for weighted networks with "radial" layout, should edge weights determine distance from ego? higher weights place alters closer to ego. default is FALSE.

`ego_ring_gap` numeric (0-1). gap between concentric rings as proportion of radius. only for "concentric" layout. default is 0.3.

`ego_size` numeric. relative size of central area reserved for ego. larger values create more space between ego and alters. default is 0.1.

special parameters

`mutate_weight` function to transform edge weights before plotting. example: `log1p` for log(x+1) transformation. applied before mapping to aesthetics.

`return_components` logical. return plot components instead of assembled plot? useful for manual customization. default is FALSE.

`style` either a style function (e.g., `style_rose`) that applies a complete visual style including colors, shapes, and layout preferences, or the string `"heatmap"` to render the adjacency matrix as a tile plot instead of a node-link diagram. when `style = "heatmap"` and edge weights cross zero (signed network), the fill scale automatically uses a diverging palette centred at zero; otherwise a sequential viridis ramp is used. optional `low`, `mid`, and `high` arguments override the diverging palette endpoints.

Author(s)

cassy dorff, shahryar minhas

Examples

```
data(classroom_edges)
data(classroom_nodes)

net <- netify(
    classroom_edges,
    actor1 = "from", actor2 = "to",
    symmetric = TRUE,
    nodal_data = classroom_nodes,
    missing_to_zero = TRUE
)

plot(net)

plot(net,
```

```
    node_size_by = "gpa",
    node_color_by = "gender",
    node_size_label = "gpa",
    node_color_label = "gender"
)

pieces <- plot(net,
    node_alpha = 0.8,
    node_size_by = "gpa",
    return_components = TRUE
)

pieces$base +
    netify_edge(pieces) +
    netify_node(pieces) +
    theme_netify()
```

`plot.netify_comparison`

Plot method for `netify_comparison` *objects*

Description

renders a pairwise similarity heatmap from the matrix output of `compare_networks()`. picks an appropriate matrix based on the `method` the comparison was run with (correlation / qap / jaccard / hamming / spectral). for qap comparisons, the plot renders the qap correlation matrix. p-values remain available in `x$significance_tests$qap_pvalues`.

Usage

```
## S3 method for class 'netify_comparison'
plot(x, ...)
```

Arguments

`x`	a `netify_comparison` object returned by `compare_networks()`.
`...`	additional arguments:
	`metric` which matrix to render. default: auto-pick the primary metric for the comparison's method. pass `"correlation"` / `"jaccard"` / `"hamming"` / `"qap"` / `"spectral"` to override.
	`show_values` logical. overlay numeric values on each tile. default `TRUE` when n <= 6, else `FALSE`.
	`palette` two-color gradient endpoints. default `c("#f0f0f0", "#1f78b4")` for correlation-like metrics.

Value

a `ggplot` object.

Author(s)

cassy dorff, shahryar minhas

`plot.summary.netify` *plot graph-level summary statistics*

Description

plot graph-level summary statistics

Usage

```
## S3 method for class 'summary.netify'
plot(x, ...)
```

Arguments

`x`	a graph-level summary returned by `summary()` for a netify object.
`...`	additional arguments passed to `plot_graph_stats()`.

Value

a ggplot object displaying graph-level summary statistics.

`plot.summary_actor` *plot method for summary_actor output*

Description

s3 method that dispatches `plot()` on a `summary_actor` data frame to `plot_actor_stats()` so the `summary_actor(net) |> plot()` idiom works without the user having to remember the helper name. pass any `plot_actor_stats()` argument through `....`

Usage

```
## S3 method for class 'summary_actor'
plot(x, ...)
```

Arguments

`x`	a `summary_actor` data frame from `summary_actor()`.
`...`	additional arguments passed to `plot_actor_stats()` (e.g. `across_actor`, `specific_stats`, `specific_actors`).

Value

a ggplot object.

Author(s)

cassy dorff, shahryar minhas

`plot_actor_stats` *Visualize actor-level network statistics*

Description

`plot_actor_stats` creates visualizations of actor-level statistics from `summary_actor()` output. the function automatically adapts to the data structure (cross-sectional/longitudinal, single/multilayer) and offers two main visualization approaches: distribution across actors or tracking specific actors.

Usage

```
plot_actor_stats(
  summary_df,
  longitudinal = ifelse("time" %in% colnames(summary_df), TRUE, FALSE),
  multilayer = ifelse("layer" %in% colnames(summary_df), TRUE, FALSE),
  across_actor = TRUE,
  specific_stats = NULL,
  specific_actors = NULL
)
```

Arguments

`summary_df`	a data frame from `summary_actor()` containing actor-level statistics. must include an "actor" column. may include "time" column for longitudinal data and "layer" column for multilayer networks.
`longitudinal`	logical indicating whether to treat data as longitudinal. if NULL (default), automatically detected based on presence of "time" column. set to FALSE if only one unique time point exists.
`multilayer`	logical indicating whether to treat data as multilayer. if NULL (default), automatically detected based on presence of "layer" column. set to FALSE if only one unique layer exists.
`across_actor`	logical. if TRUE (default), visualizes distribution of statistics across all actors. if FALSE, focuses on tracking specific actors. when TRUE with `specific_actors` provided, shows distribution for only those actors.
`specific_stats`	character vector of statistic names to plot. if NULL (default), plots all available statistics. must match column names in `summary_df`.
`specific_actors`	character vector of actor names to highlight or focus on. if NULL (default) with `across_actor = FALSE`, includes all actors (with warning if > 25 actors). must match values in the "actor" column.

Details

visualization logic:

the function chooses appropriate visualizations based on data structure:

- **distribution plots** (`across_actor = TRUE`): show how statistics are distributed across the actor population
- **actor-specific plots** (`across_actor = FALSE`): track individual actors, with specified actors highlighted in color while others appear in gray

all plots use `theme_stat_netify()` for consistent styling across netify visualizations.

for multilayer longitudinal data with `across_actor = FALSE`, consider using `specific_stats` to avoid overcrowded facets.

Value

a ggplot object that can be further customized or saved. the plot type depends on the data structure and parameters:

- **cross-sectional, across actors**: density plots with rug plots
- **cross-sectional, specific actors**: beeswarm plots
- **longitudinal, across actors**: ridge density plots over time
- **longitudinal, specific actors**: line plots over time

all plots are faceted by statistic and, when applicable, by layer.

Author(s)

cassy dorff, shahryar minhas

---

plot_graph_stats *Visualize network-level statistics*

Description

`plot_graph_stats` creates line or bar plots to visualize network-level statistics across multiple networks, time points, or layers. this function is designed to work with output from `summary()` for netify objects or similar functions that produce network-level summary statistics.

Usage

```
plot_graph_stats(summary_df, type = "line", specific_stats = NULL)
```

Arguments

summary_df	a data frame containing network-level statistics, typically from summary() for a netify object. must include a "net" column identifying each network or time point. may include a "layer" column for multilayer networks. all other columns should contain numeric statistics to plot.
type	character string specifying the plot type. options are: • "line": line plot with points (default). best for temporal data • "bar": bar plot with grouped bars. required for multilayer non-temporal data
specific_stats	character vector of statistic names to plot. if NULL (default), plots all numeric columns in summary_df. must match column names exactly.

Details

data structure detection:

the function automatically detects the structure of your data:

- **longitudinal**: multiple unique values in "net" column
- **multilayer**: multiple unique values in "layer" column
- **single network**: only one row (returns error with suggestion)

plot type selection:

- line plots are preferred for temporal data to show trends
- bar plots are automatically selected for multilayer non-temporal data
- bar plots can be useful for comparing discrete time points

faceting behavior:

each statistic gets its own facet panel with:

- independent y-axis scales (scales = "free_y")
- shared x-axis across all panels
- automatic layout based on number of statistics

Value

a ggplot object displaying the specified statistics. the plot structure depends on the data:

- **single time/network**: returns error (single row not plottable)
- **multiple times/networks**: line or bar plot faceted by statistic
- **multilayer temporal**: line plots colored by layer
- **multilayer non-temporal**: grouped bar plots by layer

all plots are faceted by statistic with free y-axis scales for better comparison across different value ranges.

Note

the function requires at least two networks/time points to create a meaningful plot. for single network summaries, consider using a table format instead.

Author(s)

ha eun choi, cassy dorff, shahryar minhas

`plot_homophily` *Visualize homophily analysis results*

Description

creates visualizations for homophily analysis results from `homophily()`. the function can create different types of plots including similarity distributions, comparison plots across multiple attributes, and temporal evolution plots.

Usage

```
plot_homophily(
  homophily_results,
  netlet = NULL,
  type = c("distribution", "comparison", "temporal"),
  attribute = NULL,
  method = "correlation",
  sample_size = NULL,
  colors = c("#2E86AB", "#F18F01"),
  ...
)
```

Arguments

`homophily_results`
: data frame output from `homophily()` or a list of such data frames for comparison plots.

`netlet`
: optional. the netify object used in the analysis. required for distribution plots to extract actual similarity data.

`type`
: character string specifying the plot type:

  "distribution" shows similarity score distributions for connected vs unconnected pairs (requires netlet)

  "comparison" compares homophily across multiple attributes

  "temporal" shows homophily evolution over time (for longitudinal data)

`attribute`
: character string. for distribution plots, specifies which attribute to visualize. should match the attribute used in `homophily()`.

method	character string. for distribution plots, the similarity method used. should match the method used in homophily().
sample_size	integer. for distribution plots with large networks, the number of dyad pairs to sample for visualization. default is NULL (use all pairs).
colors	character vector of two colors for connected/unconnected or significant/non-significant pairs. default uses package theme colors.
...	additional arguments passed to ggplot2 functions.

Value

a ggplot2 object that can be further customized.

Author(s)

cassy dorff, shahryar minhas

Examples

```
# load example data
data(classroom_edges)
data(classroom_nodes)

# create a network with nodal attributes
ntwk <- netify(
    classroom_edges,
    actor1 = "from", actor2 = "to",
    symmetric = TRUE,
    nodal_data = classroom_nodes
)

# run homophily analysis
homophily_result <- homophily(
    ntwk,
    attribute = "gender",
    method = "categorical",
    n_permutations = 100
)

# create distribution plot
plot_homophily(
    homophily_result,
    netlet = ntwk,
    type = "distribution",
    attribute = "gender",
    method = "categorical"
)
```

plot_mixing_matrix *Visualize attribute mixing matrix results*

Description

creates heatmap visualizations for attribute mixing matrices from `mixing_matrix()`. the function creates a tile plot showing how different attribute categories interact in the network.

Usage

```
plot_mixing_matrix(
  mixing_results,
  which_matrix = 1,
  show_values = TRUE,
  value_digits = 2,
  color_scale = c("#F18F01", "white", "#2E86AB"),
  midpoint = NULL,
  text_size = 4,
  text_color = "black",
  text_color_threshold = NULL,
  tile_border_color = "white",
  tile_border_size = 0.5,
  reorder_categories = FALSE,
  diagonal_emphasis = TRUE,
  ...
)
```

Arguments

`mixing_results`	output from `mixing_matrix()` containing mixing matrices and summary statistics.
`which_matrix`	integer or character. which matrix to plot if multiple are present. default is 1 (first matrix).
`show_values`	logical. whether to display values in each tile. default TRUE.
`value_digits`	integer. number of decimal places for displayed values. default 2.
`color_scale`	character vector of three colors for low, mid, and high values. default uses a diverging color scale.
`midpoint`	numeric. the midpoint for the diverging color scale. default NULL automatically calculates based on data range.
`text_size`	numeric. size of value labels in tiles. default 4.
`text_color`	character. color of text labels. default "black".
`text_color_threshold`	numeric. if provided, values above this threshold (0-1 scale) will use white text, values below will use black text. default NULL uses text_color for all.

`tile_border_color`
: character. color of tile borders. default "white".

`tile_border_size`
: numeric. width of tile borders. default 0.5.

`reorder_categories`
: logical. whether to reorder categories by similarity. default FALSE.

`diagonal_emphasis`
: logical. whether to emphasize diagonal cells (within-group mixing). default TRUE.

`...`
: additional arguments passed to ggplot2 functions.

Value

a ggplot2 object that can be further customized.

Author(s)

cassy dorff, shahryar minhas

Examples

```
# create a network with categorical attributes
data(classroom_edges)
data(classroom_nodes)
net <- netify(
    classroom_edges,
    actor1 = "from", actor2 = "to",
    symmetric = TRUE,
    nodal_data = classroom_nodes
)

# run mixing matrix analysis
mixing_result <- mixing_matrix(
    net,
    attribute = "gender"
)

# create visualization
plot_mixing_matrix(mixing_result)
```

---

plot_mixing_matrix_facet

create a multi-panel mixing matrix visualization

Description

creates a faceted plot showing multiple mixing matrices, useful for comparing patterns across time periods or network layers.

Usage

```
plot_mixing_matrix_facet(
  mixing_results,
  matrices_to_plot = NULL,
  ncol = NULL,
  shared_scale = TRUE,
  ...
)
```

Arguments

mixing_results	output from mixing_matrix() with multiple matrices
matrices_to_plot	integer vector. which matrices to include. default NULL plots all.
ncol	integer. number of columns in facet layout. default NULL auto-calculates.
shared_scale	logical. whether to use the same color scale across panels. default TRUE.
...	additional arguments passed to plot_mixing_matrix for each panel

Value

a ggplot2 object with faceted mixing matrices

Author(s)

cassy dorff, shahryar minhas

Examples

```
# create temporal network
data(classroom_edges)
data(classroom_nodes)
classroom_panel <- rbind(
    transform(classroom_edges[1:20, ], wave = 1),
    transform(classroom_edges[21:40, ], wave = 2)
)
classroom_nodes_panel <- rbind(
    transform(classroom_nodes, wave = 1),
    transform(classroom_nodes, wave = 2)
)
net_temporal <- netify(
    classroom_panel,
    actor1 = "from", actor2 = "to", time = "wave",
    symmetric = TRUE,
    nodal_data = classroom_nodes_panel
)

# run mixing matrix analysis across time
mixing_temporal <- mixing_matrix(
    net_temporal,
    attribute = "gender"
```

```
)

# create faceted visualization
plot_mixing_matrix_facet(mixing_temporal, ncol = 2)
```

`plot_with_style` *Apply style to netify plot*

Description

helper function to properly apply styles to netify plots

Usage

```
plot_with_style(netlet, style_fun, ...)
```

Arguments

`netlet`	a netify object
`style_fun`	a style function (e.g., style_rose)
`...`	additional plot parameters

Value

a ggplot object

Author(s)

cassy dorff, shahryar minhas

`print.netify` *Print method for netify objects*

Description

displays a formatted summary of a netify object, including network type, dimensions, summary statistics, and available attributes.

Usage

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

Arguments

x	a netify object
...	additional parameters (not used)

Details

the print method displays:

- network type (unipartite/bipartite, symmetric/asymmetric)
- edge weight specification
- temporal structure (cross-sectional or number of time periods)
- actor counts (total unique actors, or separate row/column counts for bipartite)
- summary statistics (density, reciprocity, transitivity, etc.)
- available nodal and dyadic attributes

for longitudinal networks, summary statistics are averaged across time periods. for multilayer networks, statistics are shown separately for each layer.

Value

invisibly returns the input netify object. called for its side effect of printing network information to the console.

Author(s)

ha eun choi, cassy dorff, colin henry, shahryar minhas

---

print.netify_comparison

Print method for netify_comparison objects

Description

provides a clear, formatted output for network comparison results. handles different comparison types (temporal, cross-network, multilayer) with appropriate formatting.

Usage

```
## S3 method for class 'netify_comparison'
print(x, ..., n = 20)
```

Arguments

x	a netify_comparison object from compare_networks()
...	additional arguments (currently unused)
n	maximum number of rows to print for summary tables (default 20)

Value

invisibly returns the input object

Author(s)

cassy dorff, shahryar minhas

Examples

```
# compare two networks
data(icews)
net1 <- netify(icews[icews$year == 2010,], actor1 = "i", actor2 = "j")
net2 <- netify(icews[icews$year == 2011,], actor1 = "i", actor2 = "j")
comp <- compare_networks(list("2010" = net1, "2011" = net2))
print(comp)
```

---

print.netify_plot_components

print netify plot components

Description

prints a summary of the components available in a netify_plot_components object. this helps users understand what layers and elements are available for manual plot construction.

Usage

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

Arguments

x	a netify_plot_components object returned from `plot(..., return_components = TRUE)`
...	additional arguments (currently unused)

Value

invisibly returns the input object

Author(s)

cassy dorff, shahryar minhas

Examples

```
# create plot components
mat <- matrix(c(NA, 1, 0, 0, NA, 1, 1, 0, NA), 3, 3,
    dimnames = list(c("alice", "bob", "carol"),
                    c("alice", "bob", "carol")))
net <- new_netify(mat, symmetric = FALSE)
comp <- plot(net, return_components = TRUE)

# print summary
print(comp)
```

`read_external` *Read a network from common file formats into a netify object*

Description

thin wrappers around `igraph::read_graph()` for the formats users coming from gephi / pajek / networkx most often hand over (graphml, pajek `.net`, gml). each reader loads the file via igraph and immediately runs `netify()` on the result so the user gets a netify back in one call, with edge weights auto-detected and directedness preserved.

Usage

```
read_graphml(file, ...)

read_pajek(file, ...)

read_gml(file, ...)
```

Arguments

`file`	path to the input file.
`...`	passed to `netify(igraph_obj, ...)` – typically `symmetric=`, `mode=`, or `weight=` overrides.

Value

a netify object.

Author(s)

cassy dorff, shahryar minhas

remove_ego_edges *remove ego-alter edges from ego network*

Description

this function removes all edges between the ego and alters in an ego network, leaving only the alter-alter connections visible. this is useful for visualizing the structure among alters without the clutter of ego connections.

Usage

```
remove_ego_edges(netlet)
```

Arguments

netlet	an ego network created with ego_netify()

Value

a modified netify object with ego edges removed

Author(s)

cassy dorff, shahryar minhas

Examples

```
mat <- matrix(c(NA, 1, 1, 1, NA, 0, 1, 0, NA), 3, 3,
    dimnames = list(c("alice", "bob", "carol"),
                    c("alice", "bob", "carol")))
net <- new_netify(mat, symmetric = FALSE)
ego_net <- ego_netify(net, ego = "alice")
alter_only_net <- remove_ego_edges(ego_net)
plot(alter_only_net)
```

reset_scales *reset aesthetic scales in ggplot*

Description

creates a scale reset object that can be added to a ggplot to reset color, fill, alpha, and size scales. this is necessary when using multiple layers with different aesthetic mappings (e.g., different colors for edges vs nodes).

Usage

```
reset_scales()
```

Details

this function addresses the limitation in ggplot2 where each aesthetic can only have one scale. by resetting scales between layers, you can have different color mappings for edges and nodes, for example.

Value

a custom object of class "netify_scale_reset" that can be added to a ggplot object using the + operator

Author(s)

cassy dorff, shahryar minhas

See Also

new_scale_color

Examples

```
# create a plot with different colors for edges and nodes
mat <- matrix(c(NA, 1, 0, 0, NA, 1, 1, 0, NA), 3, 3,
    dimnames = list(c("alice", "bob", "carol"),
                    c("alice", "bob", "carol")))
net <- new_netify(mat, symmetric = FALSE)
comp <- plot(net, return_components = TRUE)

ggplot2::ggplot() +
    netify_edge(comp) +
    ggplot2::scale_color_manual(values = c("gray", "red")) +
    reset_scales() + # reset before adding nodes
    netify_node(comp) +
    ggplot2::scale_color_viridis_c()
```

---

simulate.netify *Simulate NULL-model networks from a netify object*

Description

generates nsim new netify objects from one of three standard NULL models, holding the actor set fixed (and, where applicable, the observed density / degree sequence). useful for sanity-checking whether an observed network statistic (transitivity, modularity, etc.) is surprising relative to a chance benchmark, without reaching for statnet::ergm for a simple NULL.

Usage

```
## S3 method for class 'netify'
simulate(
  object,
  nsim = 1L,
  seed = NULL,
  model = c("erdos_renyi", "configuration", "dyad_permutation"),
  ...
)
```

Arguments

object	a netify object (cross-sectional or per-period; for longitudinal input each period is simulated independently).
nsim	integer. number of simulated draws to return.
seed	optional integer. if supplied, sets a local rng seed and restores the user's global set.seed() stream afterward. if NULL, simulation uses and advances the current rng stream normally.
model	character. one of: "erdos_renyi" independent bernoulli edges matched to the observed density (and directedness). "configuration" configuration-model rewire that preserves the observed degree sequence (in/out for directed inputs). uses igraph::sample_degseq(). "dyad_permutation" permute dyads (snijders-borgatti vertex relabel + symmetric reshuffle). preserves density and, conditional on permutation symmetry, degree distribution shape.
...	passed to the underlying model implementation.

Value

a list of length nsim of netify objects with the same class / mode / symmetry as the input.

Author(s)

cassy dorff, shahryar minhas

---

style_bipartite_network

preset style for bipartite networks

Description

a complete visual style optimized for displaying bipartite networks with two distinct node sets

Usage

```
style_bipartite_network()
```

Value

list of plot arguments

Author(s)

cassy dorff, shahryar minhas

---

style_black_yellow *black and yellow network style*

Description

a network style with dark green, black, and yellow colors.

Usage

```
style_black_yellow()
```

Value

list of plot arguments

Author(s)

cassy dorff, shahryar minhas

---

style_bronze_block *bronze block network style*

Description

a network style with bold, substantial visuals.

Usage

```
style_bronze_block()
```

Value

list of plot arguments

Author(s)

cassy dorff, shahryar minhas

style_crimson_silver *crimson and silver network style*

Description

a network style with sharp contrasts and sleek design.

Usage

```
style_crimson_silver()
```

Value

list of plot arguments

Author(s)

cassy dorff, shahryar minhas

style_cyberpunk *cyberpunk-inspired network style*

Description

a futuristic network style with neon colors on dark background

Usage

```
style_cyberpunk()
```

Value

list of plot arguments

Author(s)

cassy dorff, shahryar minhas

style_dark2 *colorbrewer dark2-based network style*

Description

a network style using colorbrewer's dark2 palette for high contrast and accessibility

Usage

```
style_dark2()
```

Value

list of plot arguments

Author(s)

cassy dorff, shahryar minhas

style_green_gold *green and gold network style*

Description

a network style with balanced green, gold, and blue colors.

Usage

```
style_green_gold()
```

Value

list of plot arguments

Author(s)

cassy dorff, shahryar minhas

style_lime_magenta *lime and magenta network style*

Description

a network style with bright green and magenta contrast.

Usage

```
style_lime_magenta()
```

Value

list of plot arguments

Author(s)

cassy dorff, shahryar minhas

style_navy_maroon *navy and maroon network style*

Description

a network style with navy blue and maroon colors.

Usage

```
style_navy_maroon()
```

Value

list of plot arguments

Author(s)

cassy dorff, shahryar minhas

style_orange_teal *orange and teal network style*

Description

a network style with gray background and bright accents.

Usage

```
style_orange_teal()
```

Value

list of plot arguments

Author(s)

cassy dorff, shahryar minhas

style_pastel *pastel rainbow network style*

Description

a soft, cheerful network style using pastel colors

Usage

```
style_pastel()
```

Value

list of plot arguments

Author(s)

cassy dorff, shahryar minhas

style_racing_blue *racing blue network style*

Description

a sleek style with clean lines, blue accents, and strong contrast.

Usage

```
style_racing_blue()
```

Value

list of plot arguments

Author(s)

cassy dorff, shahryar minhas

style_random *apply random network style*

Description

randomly selects and returns one of the available network styles

Usage

```
style_random(seed = NULL)
```

Arguments

seed	optional seed for reproducibility

Value

a style list that can be passed to plot()

Author(s)

cassy dorff, shahryar minhas

style_red_blue *red and blue network style*

Description

a network style with red accents and clean lines.

Usage

```
style_red_blue()
```

Value

list of plot arguments

Author(s)

cassy dorff, shahryar minhas

style_retro80s *retro 80s-inspired network style*

Description

a network style channeling 1980s design aesthetics with bold colors and geometric shapes

Usage

```
style_retro80s()
```

Value

list of plot arguments

Author(s)

cassy dorff, shahryar minhas

style_rose *rose network style*

Description

a network style with soft rose, purple, and gold colors.

Usage

```
style_rose()
```

Value

list of plot arguments

Author(s)

cassy dorff, shahryar minhas

style_scientific_blue *scientific blue network style*

Description

a network style with clean blue and orange colors.

Usage

```
style_scientific_blue()
```

Value

list of plot arguments

Author(s)

cassy dorff, shahryar minhas

style_slate_silver *slate and silver network style*

Description

a network style with steel gray and silver colors.

Usage

```
style_slate_silver()
```

Value

list of plot arguments

Author(s)

cassy dorff, shahryar minhas

style_solarized *solarized-inspired network style*

Description

a network style based on the popular solarized color scheme, optimized for reduced eye strain

Usage

```
style_solarized()
```

Value

list of plot arguments

Author(s)

cassy dorff, shahryar minhas

style_sunburst *sunburst network style*

Description

a playful style with energetic yellows and bold outlines.

Usage

```
style_sunburst()
```

Value

list of plot arguments

Author(s)

cassy dorff, shahryar minhas

style_temporal_network *preset style for temporal/longitudinal networks*

Description

a complete visual style optimized for displaying networks over time

Usage

```
style_temporal_network()
```

Value

list of plot arguments

Author(s)

cassy dorff, shahryar minhas

style_tufte *minimal tufte-inspired network style*

Description

a minimalist network style inspired by edward tufte's data visualization principles

Usage

```
style_tufte()
```

Value

list of plot arguments

Author(s)

cassy dorff, shahryar minhas

subset.netify *Subset netify objects*

Description

extracts a subset of a netify object based on specified actors, time periods, and/or layers while preserving all netify attributes and structure.

Usage

```
## S3 method for class 'netify'
subset(
  x,
  actors = NULL,
  time = NULL,
  layers = NULL,
  from = NULL,
  to = NULL,
  ...
)
```

Arguments

x	a netify object to subset
actors	character vector of actor names or numeric indices to subset. extracts the subgraph among these actors (includes ties both from and to these actors). default is NULL, which includes all actors.
time	time periods to subset. can be: • numeric vector: used as indices to the time dimension • character vector: matched against time dimension labels • NULL: includes all time periods (default)
layers	character vector of layer names to subset from multilayer networks. for single-layer networks, this is ignored. for multilayer networks, at least one layer must be specified.
from	character vector of actor names or numeric indices for actors sending ties (row actors). overrides actors. set to NULL to include all sending actors. in bipartite networks, this refers to actors in the first mode.
to	character vector of actor names or numeric indices for actors receiving ties (column actors). overrides actors. set to NULL to include all receiving actors. in bipartite networks, this refers to actors in the second mode.
...	additional arguments (currently unused)

Details

this function is a netify-aware wrapper around the peek function, which handles the raw data extraction. while peek returns raw matrices/arrays, subset additionally:

- preserves and updates all netify attributes
- filters nodal and dyadic attribute data to match the subset
- adjusts the netify type when dimensions change (e.g., longitudinal to cross-sectional)
- maintains consistency between network data and attributes

the from and to parameters allow precise control over which ties to include:

- use actors to get all ties among a set of actors (subgraph extraction)
- use from to get all ties sent by specific actors
- use to to get all ties received by specific actors
- use both from and to to get ties between specific sets of actors

for bipartite networks, from refers to actors in the first mode (e.g., people) and to refers to actors in the second mode (e.g., organizations).

Value

a netify object containing the requested subset with:

- subsetted adjacency matrix/matrices

- corresponding nodal attributes (filtered to included actors/times)
- corresponding dyadic attributes (filtered to included actor pairs/times)
- updated netify attributes reflecting the new dimensions

the returned object's structure depends on the subset:

- if one time period is selected from longitudinal data, returns cross-sectional
- if one layer is selected from multilayer data, returns single-layer
- otherwise, maintains the original structure type

Note

when subsetting longitudinal data to a single time period, the function automatically converts the result to a cross-sectional netify object. similarly, subsetting multilayer data to a single layer produces a single-layer object.

Author(s)

cassy dorff, shahryar minhas

Examples

```
# load example directed event data from icews
data(icews)

# generate a longitudinal netify object
# with both dyadic and nodal attributes
icews_matlConf <- netify(
    icews,
    actor1 = "i", actor2 = "j", time = "year",
    symmetric = FALSE, weight = "matlConf",
    nodal_vars = c("i_polity2", "i_log_gdp", "i_log_pop"),
    dyad_vars = c("matlCoop", "verbCoop", "verbConf"),
    dyad_vars_symmetric = c(FALSE, FALSE, FALSE)
)

# subset to a few countries using s3 method
icews_subset <- subset(
    icews_matlConf,
    actors = c(
        "United States", "United Kingdom",
        "Russian Federation", "China"
    )
)

# subset to a few countries and a few years
icews_subset_2 <- subset(
    icews_matlConf,
    actors = c(
        "United States", "United Kingdom",
```

```
        "Russian Federation", "China"
    ),
    time = c("2010", "2011")
)

# can also use subset_netify directly
icews_subset_3 <- subset_netify(
    netlet = icews_matlConf,
    actors = c(
        "United States", "United Kingdom",
        "Russian Federation", "China"
    ),
    time = c("2010", "2011")
)
```

`summary.netify` *Calculate graph-level statistics for netify objects*

Description

computes graph-level statistics for netify objects, including density, reciprocity, centralization measures, and custom metrics. handles cross-sectional and longitudinal networks, as well as multilayer structures.

Usage

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

Arguments

`object`	a netify object containing network data
`...`	additional arguments, including:
	`other_stats` named list of custom functions to calculate additional graph-level statistics. each function should accept the current netify slice and return a named vector of scalar values.

Details

the function automatically adapts calculations based on network properties:

- **bipartite networks**: reports row and column actors separately
- **directed networks**: calculates separate statistics for in/out ties
- **weighted networks**: includes weight-based statistics
- **multilayer networks**: processes each layer independently

- **longitudinal networks**: calculates statistics for each time period

competition index interpretation:

the competition measure (hhi) captures how concentrated network ties are among actors. low values indicate distributed activity across many actors (competitive), while high values indicate concentration among few actors (monopolistic). this is particularly useful for analyzing power dynamics or resource distribution in networks.

custom statistics:

add custom graph-level metrics using the `other_stats` parameter:

```
# example: community detection
modularity_stat <- function(net) {
  g <- netify_to_igraph(net)
  comm <- igraph::cluster_walktrap(g)
  c(modularity = igraph::modularity(comm),
    n_communities = length(unique(comm$membership)))
}

summary(net, other_stats = list(community = modularity_stat))
```

Value

a data frame with one row per network/time period containing:

basic network properties:

`net` network/time identifier

`layer` layer name (for multilayer networks)

`num_actors` number of actors (or `num_row_actors` and `num_col_actors` for bipartite networks)

`density` proportion of possible ties that exist

`num_edges` total number of edges (count of every non-zero entry; for signed networks, negative-weight ties are included alongside positive ones)

`prop_edges_missing` proportion of potential edge dyads that are na. uses the same denominator as `density` (off-diagonal cells for unipartite with `diag_to_NA`, halved for symmetric networks, all cells for bipartite), so `density` + `prop_edges_missing` + `observed_zero_fraction` = 1.

`prop_unknown_edges` proportion of potential edges that are na because they were *unobserved* at the data-entry stage (as opposed to structurally absent or on-diagonal). only appears when the netlet was built with `missing_to_zero` = `FALSE`; otherwise unobserved dyads have been filled with 0 and the column would be identically zero.

for weighted networks only:

`mean_edge_weight` average weight of realized non-zero edges. signed networks include negative ties; observed zero non-ties and missing dyads are excluded.

`sd_edge_weight` standard deviation of realized non-zero edge weights

`median_edge_weight` median realized non-zero edge weight

`min_edge_weight`, `max_edge_weight` range of realized non-zero edge weights

structural measures:

`competition` **(or** `competition_row`/`competition_col`**)** herfindahl-hirschman index measuring concentration of ties. calculated as $\sum_{i=1}^{n}(s_i)^2$ where s_i is actor i's share of total ties. ranges from 1/n (equal distribution) to 1 (one actor has all ties).

`sd_of_actor_means` **(or** `sd_of_row_means`/`sd_of_col_means`**)** standard deviation of actors' average tie strengths, measuring heterogeneity in actor activity levels

`transitivity` global clustering coefficient (probability that two neighbors of a node are connected). returns `na` when the coefficient is undefined.

for directed networks only:

`covar_of_row_col_means` correlation between actors' sending and receiving means. the column name is retained for compatibility.

`reciprocity` pearson correlation between the adjacency matrix and its transpose. this measures the linear association between outgoing and incoming tie weights for each dyad (i.e., how similar a_{ij} is to a_{ji} across all dyads). values near 1 indicate strong reciprocity, while values near 0 indicate no relationship. note: this differs from the traditional edge-based reciprocity (proportion of mutual dyads) used by igraph and other packages.

`mutual` proportion of mutual dyads: the fraction of connected dyad pairs where both directions are present. ranges from 0 (no mutual ties) to 1 (all ties are reciprocated). this is the traditional edge-based reciprocity measure used by igraph and most network analysis textbooks.

Note

for large longitudinal or multilayer networks, computation can be intensive. consider using `subset_netify` to analyze specific time periods or layers.

density uses the full potential-dyad denominator. missing edges (na values) are not counted as present ties; they are reported separately in `prop_edges_missing`.

Author(s)

cassy dorff, shahryar minhas

References

dorff, c., gallop, m., & minhas, s. (2023). "networks of violence: predicting conflict in nigeria." *journal of politics*, 85(1).

Examples

```
# load example data
data(icews)


# basic usage
net <- netify(
    icews,
    actor1 = "i", actor2 = "j", time = "year",
    symmetric = FALSE,
```

```
    weight = "verbCoop"
)

# get summary
summary(net)



# add custom statistics - community detection
comm_stats <- function(mat) {
    g <- netify_to_igraph(mat)
    comm <- igraph::cluster_spinglass(g)
    c(
        n_communities = length(comm$csize),
        modularity = comm$modularity
    )
}

# apply to subset for efficiency
sub_net <- subset_netify(net, time = as.character(2013:2014))
summary(sub_net, other_stats = list(community = comm_stats))
```

`summary.netify_comparison` *summary method for netify_comparison objects*

Description

provides a concise summary of network comparison results.

Usage

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

Arguments

`object`	a netify_comparison object from compare_networks()
`...`	additional arguments (currently unused)

Value

a summary data frame or list depending on comparison type

Author(s)

cassy dorff, shahryar minhas

summary_actor *Calculate actor-level network statistics*

Description

computes actor-level statistics for netify objects, including degree, strength, centrality measures, and custom metrics. handles different network types (directed/undirected, weighted/unweighted) appropriately.

Usage

```
summary_actor(
  netlet,
  invert_weights_for_igraph = TRUE,
  other_stats = NULL,
  stats = c("all", "fast")
)
```

Arguments

`netlet`	a netify object containing network data
`invert_weights_for_igraph`	logical. if TRUE (default), inverts edge weights when calculating closeness and betweenness centrality, as igraph interprets weights as distances. set to FALSE if your weights already represent distances.
`other_stats`	optional named list of custom functions to calculate additional actor-level statistics. each function should accept a matrix and return a vector with one value per actor.
`stats`	one of `"all"` (default) or `"fast"`. the `"fast"` path returns only degree- and strength-style columns and skips closeness, betweenness, eigenvector, and hits. when the user does not pass `stats` explicitly, the default auto-promotes to `"fast"` once the number of actors reaches `getoption("netify.fast_threshold", 1500l)`; passing `stats = "all"` explicitly always honors that request.

Details

the function automatically adapts calculations based on network properties:

centrality measures:

- **degree**: count of direct connections. for directed networks, calculated separately for incoming (in-degree) and outgoing (out-degree) ties.
- **closeness**: measures how quickly an actor can reach all others. based on the inverse of the sum of shortest path distances.
- **betweenness**: measures how often an actor lies on shortest paths between other actors, indicating brokerage potential.

- **eigenvector**: measures importance based on connections to other important actors. computed using the principal eigenvector of the adjacency matrix.
- **authority/hub**: for directed networks only. authority scores measure importance as targets of ties from important sources. hub scores measure importance as sources of ties to important targets.

weight handling:

by default, the function assumes larger weights indicate stronger relationships. when calculating closeness and betweenness centrality, weights are shifted to a positive scale when needed and inverted (1/weight) because igraph treats edge weights as distances. for distance-based networks where larger values already represent distances or weaker relationships, set `invert_weights_for_igraph = FALSE`. eigenvector, authority, and hub scores use the positive shifted strength scale without distance inversion. strength summaries are calculated over realized non-zero ties. observed zero non-ties and missing dyads are excluded from strength averages, standard deviations, and medians; negative signed ties remain part of the realized-tie distribution.

custom statistics:

add custom metrics using the `other_stats` parameter. each function receives the adjacency matrix and should return a vector with one value per actor:

```
# example: maximum tie weight for each actor
max_out <- function(mat) apply(mat, 1, max, na.rm = TRUE)
max_in <- function(mat) apply(mat, 2, max, na.rm = TRUE)

stats <- summary_actor(net,
  other_stats = list(max_out = max_out, max_in = max_in))
```

mathematical formulations:

for symmetric unweighted networks:

- degree: $d_i = \sum_{j=1}^{n} a_{ij}$
- proportion of ties: $p_i = \frac{d_i}{n-1}$
- network share: $s_i = \frac{d_i}{\sum_{j=1}^{n} d_j}$
- closeness: $c_i = \frac{1}{\sum_j d(i,j)}$
- betweenness: $b_i = \sum_{s \neq i \neq t} \frac{\sigma_{st}(i)}{\sigma_{st}}$

for symmetric weighted networks, additional measures:

- strength sum: $s_i^{sum} = \sum_{j=1}^{n} w_{ij}$
- strength average: mean of the actor's realized non-zero tie weights
- strength standard deviation: standard deviation of the actor's realized non-zero tie weights

for asymmetric networks, statistics are calculated separately for rows (out) and columns (in), with totals where applicable.

Value

a data frame with actor-level statistics. columns always include:

`actor` actor name/identifier

`time` time period (for longitudinal networks)

`layer` layer name (for multilayer networks)

additional columns depend on network type:

for undirected networks:

`degree` number of realized non-zero connections. for weighted networks, degree remains a count; use strength columns for sums and moments of edge weights.

`prop_ties` proportion of possible ties realized. calculated as $p_i = \frac{d_i}{n-1}$, where d_i is the degree and n is the total number of actors.

`network_share` actor's share of total network connections. for unweighted networks this is based on degree; for weighted networks this is based on absolute realized tie mass, so signed networks do not produce negative shares.

`closeness` closeness centrality. calculated as $c_i = \frac{1}{\sum_j d(i,j)}$, where $d(i,j)$ is the shortest path distance to every other actor j.

`betweenness` betweenness centrality. calculated as $b_i = \sum_{s \neq i \neq t} \frac{\sigma_{st}(i)}{\sigma_{st}}$, where σ_{st} is the total number of shortest paths from node s to node t and $\sigma_{st}(i)$ is the number of those paths that pass through i.

`eigen_vector` eigenvector centrality, based on the principal eigenvector of the adjacency matrix.

for directed networks:

`degree_in`, `degree_out` incoming and outgoing realized non-zero connection counts. for weighted networks, degree columns remain counts and strength columns summarize edge weights.

`prop_ties_in`, `prop_ties_out`, `prop_ties_total` proportion of possible in, out, and total ties.

`network_share_in`, `network_share_out`, `network_share_total` actor's share of incoming, outgoing, and total network connections. for weighted networks these shares use absolute realized tie mass.

`degree_total` sum of in and out degree

`closeness_in`, `closeness_out`, `closeness_all` directed closeness centrality

`betweenness` betweenness centrality

`authority_score`, `hub_score` authority and hub scores (asymmetric only)

for weighted networks, additional columns:

`strength_sum` sum of edge weights. directed networks return `strength_sum_in`, `strength_sum_out`, and `strength_sum_total`.

`strength_avg` average weight among realized non-zero ties. directed networks return `strength_avg_in`, `strength_avg_out`, and `strength_avg_total`.

`strength_std` standard deviation of weights among realized non-zero ties. directed networks return `strength_std_in`, `strength_std_out`, and `strength_std_total`.

`strength_median` median weight among realized non-zero ties. directed networks return `strength_median_in`, `strength_median_out`, and `strength_median_total`.

Note

for longitudinal networks, statistics are calculated separately for each time period. for multilayer networks, statistics are calculated separately for each layer unless layers have been aggregated beforehand.

missing values (na) in the network are excluded from calculations. isolates (actors with no connections) receive appropriate values (0 for degree, na for some centrality measures).

the function handles both cross-sectional and longitudinal data structures, as well as single-layer and multilayer networks. for ego networks created with netify, the function appropriately handles the ego-alter structure.

Author(s)

cassy dorff, shahryar minhas

Examples

```
# load example data
data(icews)

# basic usage with directed network
net <- netify(
    icews,
    actor1 = "i", actor2 = "j", time = "year",
    symmetric = FALSE,
    weight = "verbCoop"
)

# get actor statistics
actor_stats <- summary_actor(net)
head(actor_stats)

# add custom statistics
max_out <- function(mat) apply(mat, 1, max, na.rm = TRUE)
max_in <- function(mat) apply(mat, 2, max, na.rm = TRUE)

actor_stats_custom <- summary_actor(
    net,
    other_stats = list(
        max_out = max_out,
        max_in = max_in
    )
)
head(actor_stats_custom)
```

`theme_netify` *theme_netify function*

Description

this function returns a customized theme for netify plots. it is based on the `theme_minimal` function from the `ggplot2` package. it removes axis text and titles from the plot.

Usage

```
theme_netify()
```

Value

a customized theme object for netify plots.

Author(s)

cassy dorff, shahryar minhas

`theme_publication_netify` *ggplot theme for netify network plots*

Description

a ggplot theme with a larger base font, italicized legend titles, and (by default) stripped axes / panel chrome – the right default for a force-directed network layout. drop on top of any `plot.netify()` output.

Usage

```
theme_publication_netify(base_size = 12, for_network = TRUE)
```

Arguments

`base_size`	base font size, passed to `theme_minimal()`. default 12.
`for_network`	logical. when `TRUE` (default) removes axis text / titles / ticks / panel border, which is the right behavior for a force-directed network layout. set `FALSE` if you want to keep axes on (e.g. for a heatmap or actor-stat plot).

Value

a ggplot2 theme object.

Author(s)

cassy dorff, shahryar minhas

theme_publication_netify_ts *ggplot theme for netify time-series / stat plots*

Description

companion to `theme_publication_netify()` for plots that keep their axes – actor-stat time series, similarity heatmaps, mixing matrices, etc. same typographic settings (italic legend titles, bold strip text, larger base font) but axes and gridlines are retained.

Usage

```
theme_publication_netify_ts(base_size = 12)
```

Arguments

`base_size`	base font size. default 12.

Value

a ggplot2 theme object.

Author(s)

cassy dorff, shahryar minhas

theme_stat_netify *theme_stat_netify function*

Description

this function returns a customized theme for netify stat plots.

Usage

```
theme_stat_netify()
```

Value

a customized theme object for netify stat plots.

Author(s)

cassy dorff, shahryar minhas

`tidy.netify` *Tidy a netify object into a long edge data frame*

Description

`tidy.netify` is an s3 method for the `tidy()` generic from the broom package. it returns one row per edge with all attached nodal and dyadic attributes – equivalent to `unnetify(x, remove_zeros = TRUE)` but exposed under the broom convention so the netify object plays nicely with broom-style workflows. (broom is not a hard dependency; this method is registered as an s3 method on `tidy` and only triggers when the generic is available, e.g., when the user has `library(broom)` loaded.)

Usage

```
tidy.netify(x, remove_zeros = TRUE, ...)
```

Arguments

`x`	a netify object.
`remove_zeros`	logical. drop zero-weight edges? default `TRUE` (matches the typical broom expectation that the returned frame is actually-observed observations).
`...`	additional arguments passed to `unnetify()`.

Value

a tibble (or data.frame if `tibble` isn't installed) with one row per (non-zero) edge. columns include `from`, `to`, optional `time` (longitudinal), the edge weight column, dyadic covariates, and nodal covariates merged in with `_from` / `_to` suffixes. zero-edge inputs return a 0-row tibble with the schema preserved.

Author(s)

cassy dorff, shahryar minhas

See Also

`unnetify()` for the underlying converter, and `glance.netify` for one-row-per-network summary statistics.

Examples

```
data(icews)
icews_10 <- icews[icews$year == 2010, ]
net <- netify(icews_10, actor1 = "i", actor2 = "j",
symmetric = FALSE, weight = "verbCoop")
td <- tidy.netify(net)
head(td)
```

`to_netify` *Convert igraph, network, or matrix objects to netify format*

Description

converts various network object types (igraph, network, matrices/arrays, or lists of these) into netify objects (also available as `to_netify`). automatically extracts adjacency matrices and any nodal/dyadic attributes from the input objects.

Usage

```
to_netify(net_obj, weight = NULL, ...)
```

Arguments

`net_obj`	an r object to convert: `igraph`, `network`, matrix, array, or a list of these objects.
`weight`	optional. name of the weight attribute in `net_obj` to be used as the main edge weight in the netify object. default is `NULL`. important to specify for `igraph` and `network` objects as they do not have a default weight.
`...`	additional arguments passed to `new_netify`. can include `nodal_data` or `dyad_data` to override extracted attributes.

Details

the function handles different input types:

- **igraph**: extracts adjacency matrix, vertex attributes as nodal data, and edge attributes as dyadic data
- **network**: extracts adjacency matrix, vertex attributes as nodal data, and edge attributes as dyadic data
- **matrix**: direct conversion, no attribute extraction
- **array**: assumes 3d arrays represent longitudinal networks
- **list**: must contain all objects of the same type (all igraph, all network, or all matrices)

for longitudinal data (lists or 3d arrays), the function creates a netify object with time-indexed components. actor ordering is preserved from the input objects and made consistent across all components (adjacency, nodal, and dyadic data).

Value

a netify object with:

- adjacency matrix or list of matrices
- nodal attributes (if present in the input)
- dyadic attributes (if present in the input)
- weight specification (if provided)

Note

when converting from igraph or network objects, specify the `weight` parameter to designate which edge attribute should be used as the primary edge weight in the netify object.

Author(s)

cassy dorff, shahryar minhas

Examples

```
# from igraph
g <- igraph::sample_gnp(10, 0.3)
igraph::E(g)$weight <- runif(igraph::ecount(g))
igraph::V(g)$group <- sample(c("a", "b"), igraph::vcount(g), replace = TRUE)

net <- to_netify(g, weight = "weight")

# from network
adj <- matrix(rbinom(100, size = 1, prob = 0.3), 10, 10)
diag(adj) <- 0
n <- network::network(adj, directed = TRUE, matrix.type = "adjacency")
network::set.vertex.attribute(n, "group", sample(1:2, 10, replace = TRUE))

net <- to_netify(n)

# from matrix
adj_mat <- matrix(rnorm(100), 10, 10)
net <- to_netify(adj_mat)

# from list of matrices (longitudinal)
mat_list <- list(
    "2001" = matrix(rnorm(100), 10, 10),
    "2002" = matrix(rnorm(100), 10, 10)
)
net <- to_netify(mat_list)
```

---

unnetify *Convert netify objects back to dyadic data frames*

Description

`unnetify` (also available as `netify_to_df`) reverses the netify transformation by converting network objects back into dyadic (edge-level) data frames. this function combines network structure with any associated nodal and dyadic attributes, creating a data frame where each row represents a dyad (edge) with all relevant attributes attached.

Usage

```
unnetify(netlet, remove_zeros = FALSE)

netify_to_df(netlet, remove_zeros = FALSE)
```

Arguments

`netlet`	a netify object to be converted to dyadic format.
`remove_zeros`	logical. if TRUE, removes dyads with zero edge weights from the output, resulting in a data frame of only non-zero relationships. if FALSE (default), includes all possible dyads in the network.

Details

this function essentially reverses the netify process, making it useful for:

- exporting network data for analysis in other software
- creating dyadic datasets for regression analysis
- inspecting network data in familiar data frame format
- merging network results back with other dyadic covariates

nodal attributes are attached twice per dyad - once for the "from" actor (suffixed "_from") and once for the "to" actor (suffixed "_to"). this applies to both directed and undirected networks, ensuring that both actors' attributes are available for dyadic analysis.

the function handles both cross-sectional and longitudinal netify objects, automatically detecting the structure and adjusting the output accordingly.

Value

a data frame with one row per dyad containing:

- **from**: name/id of the first actor in the dyad
- **to**: name/id of the second actor in the dyad
- **time**: time period (for longitudinal networks)
- **weight column**: edge weight values (column named after the weight parameter used in netify)
- **from_id**: unique identifier combining actor and time (longitudinal only)
- **to_id**: unique identifier combining actor and time (longitudinal only)
- **dyadic attributes**: any edge-level covariates from the original data
- **nodal attributes**: actor-level covariates merged onto dyads, suffixed with "_from" and "_to" to match the corresponding actor

Note

for large networks, setting `remove_zeros = FALSE` can result in very large data frames, as it includes all possible dyads (n x (n-1) for directed networks or n x (n-1) / 2 for undirected networks).

Author(s)

cassy dorff, shahryar minhas

Examples

```
# load example data
data(icews)

# create a netify object with attributes
icews_10 <- icews[icews$year == 2010, ]

verbCoop_net <- netify(
    icews_10,
    actor1 = "i", actor2 = "j",
    symmetric = FALSE,
    weight = "verbCoop",
    nodal_vars = c("i_polity2", "i_log_gdp"),
    dyad_vars = c("verbConf", "matlConf")
)

# convert back to dyadic data frame
dyad_df <- unnetify(verbCoop_net)

# examine structure
head(dyad_df)
names(dyad_df)

# remove zero-weight dyads for more compact output
dyad_df_nonzero <- unnetify(verbCoop_net, remove_zeros = TRUE)
nrow(dyad_df_nonzero) # much smaller than full dyadic dataset

# note how nodal attributes are added
# for directed network: _from and _to suffixes
head(dyad_df[, c("from", "to", "i_polity2_from", "i_polity2_to")])

# longitudinal example
verbCoop_longit <- netify(
    icews,
    actor1 = "i", actor2 = "j",
    time = "year",
    symmetric = FALSE,
    weight = "verbCoop",
    nodal_vars = c("i_polity2", "i_log_gdp")
)

# convert longitudinal network
dyad_df_longit <- unnetify(verbCoop_longit, remove_zeros = TRUE)

# check time periods are included
table(dyad_df_longit$time)

# each dyad now has associated time period
```

```
# note: weight column is named after the weight variable (verbCoop)
head(dyad_df_longit[, c("from", "to", "time", "verbCoop")])

# use the output for further analysis

# for example, regression analysis
lm(verbCoop ~ i_polity2_from + i_polity2_to + verbConf, data = dyad_df)
```

---

validate_netify *Deep coherence check on a netify object*

Description

netify_check() validates only class membership. validate_netify() goes further: it verifies that the netify's internal pieces still agree with each other after any user-side surgery (e.g., manually edited attr(., "nodal_data"), subset() followed by an attribute overwrite, etc.). use this when you've hand-modified a netlet and want to confirm it's still well-formed before passing to to_statnet() / to_amen() / plot().

Usage

```
validate_netify(netlet, verbose = TRUE)
```

Arguments

netlet	a netify object.
verbose	logical. if TRUE (default), print a per-check status banner; otherwise return silently.

Value

invisibly returns a list with one logical per check (TRUE = passed). the function cli::cli_abort()s on any failure unless verbose = TRUE, in which case failures are reported per-check and the function returns invisibly with the full failure list.

Author(s)

cassy dorff, shahryar minhas

Examples

```
data(icews)
net <- netify(icews[icews$year == 2010, ],
actor1 = "i", actor2 = "j", symmetric = FALSE, weight = "verbCoop")
validate_netify(net)
```

Index