Package ‘CausalQueries’

June 3, 2020

Type Package
Title Make, Update, and Query Binary Causal Models
Version 0.0.3
Description Users can declare binary causal models, update beliefs about causal types given data and calculate arbitrary estimands. Model definition makes use of 'dagitty' functionality. Updating is implemented in 'stan'. The approach used in 'CausalQueries' is a generalization of the 'biqq' models described in "Mixing Methods: A Bayesian Approach" (Humphreys and Jacobs, 2015, <DOI:10.1017/S0003055415000453>). The conceptual extension makes use of work on probabilistic causal models described in Pearl's Causality (Pearl, 2009, <DOI:10.1017/CBO9780511803161>).

License MIT + file LICENSE
Encoding UTF-8
LazyData true
RoxygenNote 7.1.0
Depends dplyr, methods, R (>= 3.4.0), Rcpp (>= 0.12.0)
Imports dagitty, gtools, stats, randomizr, rlang (>= 0.2.0), rstan (>= 2.18.1), rstantools (>= 2.0.0), stringr
LinkingTo BH (>= 1.66.0), Rcpp (>= 0.12.0), RcppEigen (>= 0.3.3.3.0), rstan (>= 2.18.1), StanHeaders (>= 2.18.0)
Suggests testthat, rmarkdown, knitr, DeclareDesign, covr
SystemRequirements GNU make
Biarch true
NeedsCompilation yes
Author Clara Bicalho [ctb],
Jasper Cooper [ctb],
Macartan Humphreys [aut],
Alan Jacobs [aut],
Merlin Heidemanns [ctb],
Lily Medina [aut, cre],
Julio Solis [ctb],
Georgiy Syunyaev [ctb]
Maintainer Lily Medina <lilymiru@gmail.com>
Repository CRAN
Date/Publication 2020-06-03 16:20:09 UTC

R topics documented:

CausalQueries-package .............................................. 3
collapse_data .................................................. 3
complements .................................................... 4
data_type_names ................................................ 6
decreasing ...................................................... 7
democracy_data .................................................. 8
expand_data ..................................................... 9
expand_wildcard .................................................. 9
get_ambiguities_matrix .......................................... 10
get_causal_types ................................................ 11
get_event_prob .................................................. 11
get_nodal_types .................................................. 12
get_parameters .................................................. 13
get_parameter_matrix .......................................... 14
get_parameter_names .......................................... 14
get_param_dist ................................................... 15
get_parents ..................................................... 15
get_priors ....................................................... 16
get_prior_distribution ......................................... 17
get_query_types .................................................. 17
get_type_prob .................................................... 19
get_type_prob_multiple ........................................ 20
increasing ...................................................... 21
interacts ......................................................... 21
interpret_type ................................................... 22
make_confounds_df .............................................. 23
make_data ....................................................... 24
make_events ..................................................... 26
make_model ........................................................ 27
make_parameters .................................................. 28
make_parameter_matrix ........................................ 30
make_priors ....................................................... 31
make_prior_distribution ....................................... 33
make_values_task_list ......................................... 34
non_decreasing ................................................. 35
non_increasing ................................................. 36
observe_data ..................................................... 36
query_distribution .............................................. 37
query_model ...................................................... 38
reveal_outcomes ................................................. 40
'CausalQueries' is a package that lets you declare binary causal models, update beliefs about causal types given data and calculate arbitrary estimands. Model definition makes use of dagitty functionality. Updating is implemented in 'stan'.

all_data_types

Creates dataframe with all data types (including NA types) that are possible from a model.

Usage

```r
all_data_types(
  model,
  complete_data = FALSE,
  possible_data = FALSE,
  given = NULL
)
```
collapse_data

Arguments

model A causal_model. A model object generated by make_model.
complete_data Logical. If ‘TRUE’ returns only complete data types (no NAs). Defaults to ‘FALSE’.
possible_data Logical. If ‘TRUE’ returns only complete data types (no NAs) that are *possible* given model restrictions. Note that in principle an intervention could make observationally impossible data types arise. Defaults to ‘FALSE’.
given A character. A quoted statement that evaluates to logical. Data conditional on specific values.

Value

A data.frame with all data types (including NA types) that are possible from a model.

Examples

all_data_types(make_model('X -> Y'))
model <- make_model('X -> Y') %>% set_restrictions(labels = list(Y = '00'), keep = TRUE)
all_data_types(model)
all_data_types(model, complete_data = TRUE)
all_data_types(model, possible_data = TRUE)
all_data_types(model, given = 'X==1')
all_data_types(model, given = 'X==1 & Y==1')

collapse_data

Make compact data with data strategies

Description

Take a ‘data.frame’ and return compact ‘data.frame’ of event types and strategies.

Usage

collapse_data(
  data,
  model,
  drop_NA = TRUE,
  drop_family = FALSE,
  summary = FALSE
)
Arguments

data A data.frame. Data of nodes that can take three values: 0, 1, and NA. In long form as generated by `make_events`.

model A causal_model. A model object generated by `make_model`.

drop NA Logical. Whether to exclude strategy families that contain no observed data. Exceptionally if no data is provided, minimal data on data on first node is returned. Defaults to `TRUE`.

drop family Logical. Whether to remove column `strategy` from the output. Defaults to `FALSE`.

summary Logical. Whether to return summary of the data. See details. Defaults to `FALSE`.

Value

A vector of data events

If `summary = TRUE` `collapse_data` returns a list containing the following components:

data_events A compact data.frame of event types and strategies.

observed_events A vector of character strings specifying the events observed in the data

unobserved_events A vector of character strings specifying the events not observed in the data

Examples

```r
model <- make_model('X -> Y')
df <- simulate_data(model, n = 10)
df[1,1] <- ''
collapse_data(df, model)

collapse_data(df, model, drop_NA = FALSE)

collapse_data(df, model, drop_family = TRUE)

collapse_data(df, model, summary = TRUE)
data <- simulate_data(model, n = 0)
collapse_data(data, model)

model <- make_model('X -> Y') %>% set_restrictions('X[1]==1')
df <- simulate_data(model, n = 10)
df[1,1] <- ''
collapse_data(df, model)
data <- data.frame(X = 0:1)
collapse_data(data, model)

model <- make_model('X->Y')
long_data <- simulate_data(model, n = 6)
```
complements

collapse_data(long_data, model)

---

### complements

#### Make statement for complements

**Description**

Generate a statement for X1, X1 complement each other in the production of Y

**Usage**

complements(X1, X2, Y)

**Arguments**

- **X1**: A character. The quoted name of the input node 1.
- **X2**: A character. The quoted name of the input node 2.
- **Y**: A character. The quoted name of the outcome node.

**Value**

A character statement of class statement

**See Also**

Other statements: decreasing(), increasing(), interacts(), non_decreasing(), non_increasing(), substitutes(), te()

**Examples**

complements('A', 'B', 'W')
**data_type_names**

**Data type names**

**Description**

Provides names to data types

**Usage**

```r
data_type_names(model, data)
```

**Arguments**

- `model` A causal_model. A model object generated by `make_model`.
- `data` A data.frame. Data of nodes that can take three values: 0, 1, and NA. In long form as generated by `make_events`

**Value**

A vector of strings of data types

**Examples**

```r
model <- make_model('X -> Y')
data <- simulate_data(model, n = 2)
data_type_names(model, data)
```

**decreasing**

*Make monotonicity statement (negative)*

**Description**

Generate a statement for Y monotonic (decreasing) in X

**Usage**

```r
decreasing(X, Y)
```

**Arguments**

- `X` A character. The quoted name of the input node
- `Y` A character. The quoted name of the outcome node

**Value**

A character statement of class `statement`
See Also

Other statements: `complements()`, `increasing()`, `interacts()`, `non_decreasing()`, `non_increasing()`, `substitutes()`, `te()`

Examples

decreasing('A', 'B')

democracy_data

Democracy Data

Description

A dataset containing information on inequality, democracy, mobilization, and international pressure. Made by `devtools::use_data(democracy_data,CausalQueries)`

Usage

democracy_data

Format

A data frame with 84 rows and 5 nodes:

Case  Case
D  Democracy
I  Inequality
P  International Pressure
M  Mobilization

Source

### expand_data

*Expand compact data object to data frame*

**Description**

Expand compact data object to data frame

**Usage**

```r
expand_data(data_events = NULL, model)
```

**Arguments**

- `data_events` A `data.frame`. It must be compatible with nodes in `model`. The default columns are `event`, `strategy` and `count`.
- `model` A `causal_model`. A model object generated by `make_model`.

**Value**

A `data.frame` with rows as data observation

**Examples**

```r
model <- make_model("X->M->Y")
make_events(model, n = 5) %>%
  expand_data(model)
make_events(model, n = 0) %>%
  expand_data(model)
```

### expand_wildcard

*Expand wildcard*

**Description**

Expand statement containing wildcard

**Usage**

```r
expand_wildcard(to_expand, join_by = "|", verbose = TRUE)
```
get_ambiguities_matrix

Arguments

to_expand A character vector of length 1L.
join_by A logical operator. Used to connect causal statements: AND ('&') or OR ('|'). Defaults to '|'.
verbose Logical. Whether to print expanded query on the console.

Value

A character string with the expanded expression. Wildcard '.' is replaced by 0 and 1.

Examples

# Position of parentheses matters for type of expansion
# In the "global expansion" versions of the entire statement are joined
expand_wildcard('('Y[X=1, M=.] > Y[X=1, M=.])')
# In the "local expansion" versions of indicated parts are joined
expand_wildcard('Y[X=1, M.=] > (Y[X=1, M.=])')

# If parentheses are missing global expansion used.
expand_wildcard('Y[X=1, M.=] > Y[X=1, M.=]')

# Expressions not requiring expansion are allowed
expand_wildcard('Y[X=1]')

get_ambiguities_matrix

Get ambiguities matrix

Description

Return ambiguities matrix if it exists; otherwise calculate it assuming no confounding. The ambiguities matrix maps from causal types into data types.

Usage

get_ambiguities_matrix(model)

Arguments

model A causal_model. A model object generated by make_model.

Value

A data.frame. Causal types (rows) corresponding to possible data realizations (columns).
### get_causal_types

**Examples**

```r
model <- make_model("X -> Y")
get_ambiguities_matrix(model = model)
```

---

**get_causal_types**  
*Get causal types*

**Description**

Return data frame with types produced from all combinations of possible data produced by a DAG.

**Usage**

```r
get_causal_types(model)
```

**Arguments**

- `model`  
  A causal_model. A model object generated by `make_model`.

**Value**

A data.frame indicating causal types of a model

**Examples**

```r
get_causal_types(make_model("X -> Y"))
```

---

### get_event_prob

**Draw event probabilities**

**Description**

Draw event probabilities

**Usage**

```r
get_event_prob(model, P = NULL, A = NULL, parameters = NULL, type_prob = NULL)
```
get_nodal_types

Get list of types for nodes in a DAG

Arguments

- **model** A causal_model. A model object generated by `make_model`.
- **P** A data.frame. Parameter matrix. Not required but may be provided to avoid repeated computation for simulations.
- **A** A data.frame. Ambiguity matrix. Not required but may be provided to avoid repeated computation for simulations.
- **parameters** A vector of real numbers in [0,1]. Values of parameters to specify (optional). By default, parameters is drawn from `model$parameters_df`.
- **type_prob** A numeric vector. Type probabilities. (Not required).

Value

An array of event probabilities

Examples

```r
model <- make_model('X -> Y')
get_event_prob(model = model)
get_event_prob(model = model, parameters = rep(1, 6))
get_event_prob(model = model, parameters = 1:6)
```

Description

As type labels are hard to interpret for large models, the type list includes an attribute to help interpret them. See `attr(types, interpret)`

Usage

```r
get_nodal_types(model, collapse = TRUE)
```
get_parameters

Examples

```r
model <- make_model('X -> K -> Y')
get_nodal_types(model)

model <- make_model('X -> K -> Y') %>%
  set_restrictions(statement = 'K[X=1]>K[X=0]') %>%
  set_confound(list(K = 'Y[X=1]>Y[X=0]'))
get_nodal_types(model)
```

---

get_parameters  Get parameters

Description

Extracts parameters as a named vector

Usage

```r
get_parameters(model, param_type = NULL)
```

Arguments

- `model`: A causal_model. A model object generated by `make_model`.
- `param_type`: A character. String specifying type of parameters to set ("flat", "prior_mean", "posterior_mean", "prior_draw", "posterior_draw", "define"). With `param_type` set to define use arguments to be passed to `make_priors`; otherwise `flat` sets equal probabilities on each nodal `param_type` in each parameter set; `prior_mean`, `prior_draw`, `posterior_mean`, `posterior_draw` take parameters as the means or as draws from the prior or posterior.

Value

A vector of draws from the prior or distribution of parameters

See Also

Other parameters: `make_parameters()`, `set_parameters()`

Examples

```r
get_parameters(make_model('X -> Y'))
```
get_parameter_matrix  Get parameter matrix

Description

Return parameter matrix if it exists; otherwise calculate it assuming no confounding. The parameter matrix maps from parameters into causal types. In models without confounding parameters correspond to nodal types.

Usage

get_parameter_matrix(model)

Arguments

model  A model created by make_model()

Value

A data.frame, the parameter matrix, mapping from parameters to causal types

Examples

model <- make_model('X -> Y')
get_parameter_matrix(model)

generate_names  Get parameter names

Description

Parameter names taken from $P$ matrix or model if no $P$ matrix provided

Usage

generate_names(model, include_paramset = TRUE)

Arguments

model  A causal_model. A model object generated by make_model.
include_paramset
   Logical. Whether to include the param set prefix as part of the name.

Value

A character vector with the names of the parameters in the model
get_param_dist  

Examples

get_parameter_names(make_model('X->Y'))

get_param_dist  Get a distribution of model parameters

Description

Using parameters, priors, or posteriors

Usage

get_param_dist(model, using, n_draws = 4000)

Arguments

model  A causal_model. A model object generated by make_model.
using  A character string. It indicates whether to use 'priors', 'posteriors' or 'parameters'.
n_draws  An integer. If no prior distribution is provided, generate prior distribution with n_draws number of draws.

Value

A matrix with the distribution of the parameters in the model

Examples

get_param_dist(model = make_model('X->Y'), using = 'priors', n_draws = 4)
get_param_dist(model = make_model('X->Y'), using = 'parameters')

get_parents  Get list of parents of all nodes in a model

Description

Get list of parents of all nodes in a model

Usage

get_parents(model)
get_priors

Arguments

model A causal_model. A model object generated by make_model.

Value

A list of parents in a DAG

Examples

model <- make_model('X -> K -> Y')
get_parents(model)

get_priors Get priors

Description

Extracts priors as a named vector

Usage

get_priors(model)

Arguments

model A model object generated by make_model().

Value

A vector indicating the hyperparameters of the prior distribution of the nodal types.

See Also

Other priors: make_par_values_multiple(), make_par_values(), make_priors(), make_values_task_list(), set_priors()

Examples

get_priors(make_model('X -> Y'))
get_prior_distribution

Get a prior distribution from priors

Description

Add to the model a `n_draws x n_param` matrix of possible parameters.

Usage

get_prior_distribution(model, n_draws = 4000)

Arguments

- `model` A `causal_model`. A model object generated by `make_model`.
- `n_draws` A scalar. Number of draws.

Value

A `data.frame` with dimension `n_param`x `n_draws` of possible lambda draws

See Also

Other prior_distribution: `make_prior_distribution()`, `set_prior_distribution()`

Examples

```r
make_model(\'X -> Y\') %>% set_prior_distribution(n_draws = 5) %>% get_prior_distribution()
make_model(\'X -> Y\') %>% get_prior_distribution(3)
```

get_query_types

Look up query types

Description

Find which nodal or causal types are satisfied by a query.

Usage

get_query_types(model, query, map = "causal_type", join_by = "|")
get_query_types

Arguments

model: A causal_model. A model object generated by make_model.
query: A character string. An expression defining nodal types to interrogate reveal_outcomes.
map: Types in query. Either nodal_type or causal_type. Default is causal_type.
join_by: A logical operator. Used to connect causal statements: AND ('&') or OR ('|'). Defaults to '|'.

Value

A list containing some of the following elements:
types: A named vector with logical values indicating whether a nodal_type or a causal_type satisfy 'query'.
query: A character string as specified by the user.
expanded_query: A character string with the expanded query. Only differs from 'query' if this contains wildcard '.
evaluated_nodes: Value that the nodes take given a query.
node: A character string of the node whose nodal types are being queried.
type_list: List of causal types satisfied by a query.

Examples

model <- make_model("X -> M -> Y; X->Y")
query <- '(Y[X=0] > Y[X=1])'
get_query_types(model, query, map="nodal_type")
get_query_types(model, query, map="causal_type")
get_query_types(model, query)

# Examples with map = "nodal_type"
query <- '(Y[X=0, M = .] > Y[X=1, M = 0])'
get_query_types(model, query, map="nodal_type")
query <- '(Y[] == 1)'
get_query_types(model, query, map="nodal_type")
get_query_types(model, query, map="nodal_type", join_by = '&')

# Root nodes specified with []
get_query_types(model, "(X[] == 1)", map="nodal_type")
query <- '(M[X=1] == M[X=0])'
get_query_types(model, query, map="nodal_type")

# Helpers
model <- make_model("M->Y; X->Y")
query <- complements('X', 'M', 'Y')
get_query_types(model, query, map="nodal_type")
# Examples with map = "causal_type"

```r
model <- make_model('X -> M -> Y; X->Y')
query <- 'Y[M=M[X=0], X=1]==1'
get_query_types(model, query, map= "causal_type")

query <- '(Y[X=1, M = 1] > Y[X=0, M = 1]) & (Y[X=1, M = 0] > Y[X=0, M = 0])'
get_query_types(model, query, "causal_type")

query <- 'Y[X=1] == Y[X=0]'
get_query_types(model, query, "causal_type")

query <- '(X == 1) & (M==1) & (Y ==1) & (Y[X=0] ==1)'
get_query_types(model, query, "causal_type")

query <- '(Y[X = .] ==1)'
get_query_types(model, query, "causal_type")
```

---

**get_type_prob**

**Get type probabilities**

**Description**

Gets probability of vector of causal types given a single realization of parameters, possibly drawn from model priors.

**Usage**

```r
get_type_prob(model, P = NULL, parameters = NULL)
```

**Arguments**

- **model**
  A causal_model. A model object generated by `make_model`.

- **P**
  A data.frame. Parameter matrix. Not required but may be provided to avoid repeated computation for simulations.

- **parameters**
  A vector of real numbers in [0,1]. Values of parameters to specify (optional). By default, parameters is drawn from `model$parameters_df`.

**Details**

By default, parameters is drawn from 'using' argument (either from priors, posteriors, or from `model$parameters`)

**Value**

A vector with probabilities of vector of causal types
get_type_prob_multiple

Draw matrix of type probabilities, before or after estimation

Description

Draw matrix of type probabilities, before or after estimation

Usage

```
get_type_prob_multiple(
  model,
  using = "priors",
  parameters = NULL,
  n_draws = 4000,
  param_dist = NULL
)
```

Arguments

- **model**: A causal_model. A model object generated by `make_model`.
- **using**: A character. It indicates whether to use 'priors', 'posteriors' or 'parameters'.
- **parameters**: A vector of real numbers in [0,1]. Values of parameters to specify (optional). By default, parameters is drawn from `model$parameters_df`.
- **n_draws**: An integer. If no prior distribution is provided, generate prior distribution with n_draws number of draws.
- **param_dist**: A matrix. Distribution of parameters. Optional for speed.

Value

A matrix of type probabilities.

Examples

```
model <- make_model('X -> Y')
get_type_prob_multiple(model, using = 'priors', n_draws = 3)
getype_prob_multiple(model, using = 'parameters', n_draws = 3)
```
increasing  Make monotonicity statement (positive)

Description
Generate a statement for Y monotonic (increasing) in X

Usage
increasing(X, Y)

Arguments
X  A character. The quoted name of the input node
Y  A character. The quoted name of the outcome node

Value
A character statement of class statement

See Also
Other statements: complements(), decreasing(), interacts(), non_decreasing(), non_increasing(), substitutes(), te()

Examples
increasing('A', 'B')

interacts  Make statement for any interaction

Description
Generate a statement for X1, X2 interact in the production of Y

Usage
interacts(X1, X2, Y)

Arguments
X1  A character. The quoted name of the input node 1.
X2  A character. The quoted name of the input node 2.
Y   A character. The quoted name of the outcome node.
Value

A character statement of class statement

See Also

Other statements: `complements()`, `decreasing()`, `increasing()`, `non_decreasing()`, `non_increasing()`, `substitutes()`, `te()`

Examples

```r
interacts('A', 'B', 'W')
get_query_types(model = make_model('X-> Y <- W'),
    query = interacts('X', 'W', 'Y'), map = "causal_type")
```

**interpret_type**  
Interpret or find position in nodal type

Description

Interprets the position of one or more digits (specified by position) in a nodal type. Alternatively returns nodal type digit positions that correspond to one or more given condition.

Usage

```r
interpret_type(model, condition = NULL, position = NULL)
```

Arguments

- **model**: A causal_model. A model object generated by `make_model`.
- **condition**: A vector of characters. Strings specifying the child node, followed by '|' (given) and the values of its parent nodes in `model`.
- **position**: A named list of integers. The name is the name of the child node in `model`, and its value a vector of digit positions in that node’s nodal type to be interpreted. See ‘Details’.

Details

A node for a child node X with k parents has a nodal type represented by X followed by $2^k$ digits. Argument position allows user to interpret the meaning of one or more digit positions in any nodal type. For example `position = list(X = 1:3)` will return the interpretation of the first three digits in causal types for X. Argument condition allows users to query the digit position in the nodal type by providing instead the values of the parent nodes of a given child. For example, `condition = 'X | Z=0 & R=1'` returns the digit position that corresponds to values X takes when Z = 0 and R = 1.
make_confounds_df

Value
A named list with interpretation of positions of the digits in a nodal type

Examples
model <- make_model('R -> X; Z -> X; X -> Y')
# Example using digit position
interpret_type(model, position = list(X = c(3,4), Y = 1))
# Example using condition
interpret_type(model, condition = c('X | Z=0 & R=1', 'X | Z=0 & R=0'))
# Return interpretation of all digit positions of all nodes
interpret_type(model)

make_confounds_df  Make a confounds dataframe

Description
Identifies confounded nodal types.

Usage
make_confounds_df(model)

Arguments
model A causal_model. A model object generated by make_model.

Value
A data.frame indicating which nodes are confounded

Examples
model <- make_model('X -> Y') %>%
set_confound('X <-> Y', add_confounds_df = FALSE)
make_confounds_df(model)

model <- make_model('X -> M -> Y; X <-> Y') %>%
set_restrictions(c('M[X=1] == M[X=0]', 'Y[M=1] == Y[M=0]'))
make_confounds_df(model)

model <- make_model('X -> M -> Y; X <-> M; M <-> Y') %>%
set_restrictions(c('M[X=1] == M[X=0]', 'Y[M=1] == Y[M=0]'))
make_confounds_df(model)

# The implied confounding is between X and M and also between X and Y
model <- make_model('X -> M -> Y') %>%
set_confound(list(X = 'Y[X=1] > Y[X=0]'), add_confound_df = FALSE)
make_confound_df(model)

model <- make_model('X -> M -> Y')
make_confound_df(model)

# Bad case
## Not run:
model <- make_model('X -> Y') %>%
  set_confound(list(X = 'X=1'))

## End(Not run)

# Complex confounding 1
model <- make_model('A -> X <- B; A <-> X; B <-> X')
model$confound_df

# Complex confounding 2
model <- make_model('A <- X -> B; A <-> X; B <-> X') %>%
  set_restrictions(c('A[X=0] == A[X=1]', 'B[X=0] == B[X=1]'))
  table(model$parameters_df$param_set)
model$confound_df

# Full confounding: X, A|X, B|A,X with 7 degrees of freedom
model <- make_model('A <- X -> B; A <-> X; B <-> X; A<->B') %>%
  set_restrictions(c('A[X=0] == A[X=1]', 'B[X=0] == B[X=1]'))
  table(model$parameters_df$param_set)
model$confound_df

### make_data

#### Make data

**Description**

Make data

**Usage**

```r
make_data(
  model,
  n = 1,
  parameters = NULL,
  param_type = NULL,
  nodes = NULL,
  n_steps = NULL,
  probs = NULL,
  subsets = TRUE,
  complete_data = NULL,
  ...
)
```
Arguments

- **model**: A causal_model. A model object generated by `make_model`.
- **n**: Non-negative integer. Number of observations.
- **parameters**: A vector of real numbers in [0,1]. Values of parameters to specify (optional). By default, parameters are drawn from `model$parameters_df`.
- **param_type**: A character. String specifying type of parameters to make ("flat", "prior_mean", "posterior_mean", "prior_draw", "posterior_draw", "define"). With `param_type` set to define use arguments to be passed to `make_priors`; otherwise `flat` sets equal probabilities on each nodal type in each parameter set; `prior_mean`, `prior_draw`, `posterior_mean`, `posterior_draw` take parameters as the means or as draws from the prior or posterior.
- **nodes**: A list. Which nodes to be observed at each step
- **n_steps**: A list. Number of observations to be observed at each step
- **probs**: A list. Observation probabilities at each step
- **subsets**: A list. Strata within which observations are to be observed at each step
- **complete_data**: A data.frame. Dataset with complete observations. Optional.
- **...**: additional arguments that can be passed to `link{make_parameters}`

Value

A data.frame with simulated data.

Examples

```r
# Simple draws
model <- make_model("X -> M -> Y")
make_data(model)
make_data(model, n = 3, nodes = c("X","Y"))
make_data(model, n = 3, param_type = "prior_draw")
make_data(model, n = 10, param_type = "define", parameters = 0:9)

# Data Strategies
# A strategy in which X, Y are observed for sure and M is observed
# with 50% probability for X=1, Y=0 cases

model <- make_model("X -> M -> Y")
make_data(
  model,
  n = 8,
  nodes = list(c("X", "Y"), "M"),
  probs = list(1, .5),
  subsets = list(NULL, "X==1 & Y==0"))
```
**make_events**

**Make data in compact form**

**Description**

Draw \( n \) events given event probabilities. Draws full data only. For incomplete data see `make_data`.

**Usage**

```r
make_events(
  model,
  n = 1,
  w = NULL,
  P = NULL,
  A = NULL,
  parameters = NULL,
  param_type = NULL,
  include_strategy = FALSE,
  ...
)
```

**Arguments**

- `model`: A causal_model. A model object generated by `make_model`.
- `n`: An integer. Number of observations.
- `w`: A numeric matrix. A ‘n_parameters x 1’ matrix of event probabilities with named rows.
- `P`: A data.frame. Parameter matrix. Not required but may be provided to avoid repeated computation for simulations.
- `A`: A data.frame. Ambiguity matrix. Not required but may be provided to avoid repeated computation for simulations.
- `parameters`: A vector of real numbers in \([0,1]\). Values of parameters to specify (optional). By default, parameters is drawn from `model$parameters_df`.
- `param_type`: A character. String specifying type of parameters to make (‘flat’, ‘prior_mean’, ‘posterior_mean’, ‘prior_draw’, ‘posterior_draw’, ‘define’). With `param_type` set to define use arguments to be passed to `make_priors`; otherwise flat sets equal probabilities on each nodal type in each parameter set; prior_mean, prior_draw, posterior_mean, posterior_draw take parameters as the means or as draws from the prior or posterior.
- `include_strategy`: Logical. Whether to include a ‘strategy’ vector. Defaults to FALSE. Strategy vector does not vary with full data but expected by some functions.
- `...`: Arguments to be passed to `make_priors` if `param_type == define`
`make_model`  

**Value**  
A `data.frame` of events

**Examples**

```r
model <- make_model('X -> Y')
make_events(model = model)
make_events(model = model, param_type = 'prior_draw')
make_events(model = model, include_strategy = TRUE)
```

**Description**

`make_model` uses `dagitty` syntax and functionality to specify nodes and edges of a graph. Implied causal types are calculated and default priors are provided under the assumption of no confounding. Models can be updated with specification of a parameter matrix, $P$, by providing restrictions on causal types, and/or by providing informative priors on parameters. The default setting for a causal model have flat (uniform) priors and parameters putting equal weight on each parameter within each parameter set. These can be adjust with `set_priors` and `set_parameters`.

**Usage**

```r
make_model(statement, add_causal_types = TRUE)
```

**Arguments**

- `statement` A character. Statement describing causal relations using `dagitty` syntax. Only directed relations are permitted. For instance "X -> Y" or "X1 -> Y <- X2; X1 -> X2".
- `add_causal_types` Logical. Whether to create and attach causal types to `model`. Defaults to `TRUE`.

**Value**

An object of class `causal_model`.
An object of class "causal_model" is a list containing at least the following components:

- `dag` A `data.frame` with columns ‘parent’and ‘children’ indicating how nodes relate to each other.
- `node` A named list with the nodes in the model
- `statement` A character vector of the statement that defines the model
- `nodal_types` A named list with the nodal types in the model
- `parameters_df` A `data.frame` with descriptive information of the parameters in the model
Examples

```r
make_model(statement = "X -> Y")
modelXKY <- make_model("X -> K -> Y; X -> Y")

# Example where cyclicaly dag attempted
## Not run:
modelXKL <- make_model("X -> K -> L")
## End(Not run)

# Examples with confounding
model <- make_model("X->Y; X <-> Y")
model$P
model <- make_model("Y2 <- X -> Y1; X <-> Y1; X <-> Y2")
model$P
model$confounds_df
dim(model$P)
model$P
model <- make_model("X1 -> Y <- X2; X1 <-> Y; X2 <-> Y")
dim(model$P)
model$parameters_df

# A single node graph is also possible
model <- make_model("X")
plot(model)

# Unconnected nodes cannot
## Not run:
model <- make_model("X <-> Y")
plot(model)
## End(Not run)
```

---

**make_parameters**

*Make a 'true' parameter vector*

**Description**

A vector of 'true' parameters; possibly drawn from prior or posterior.

**Usage**

```r
make_parameters(
  model,
  parameters = NULL,
  param_type = NULL,
  warning = TRUE,
  normalize = TRUE,
  ...
)
```
**make_parameters**

Arguments

- **model**: A causal_model. A model object generated by `make_model`.
- **parameters**: A vector of real numbers in [0,1]. Values of parameters to specify (optional). By default, parameters is drawn from `model$parameters_df`.
- **param_type**: A character. String specifying type of parameters to make ("flat", "prior_mean", "posterior_mean", "prior_draw", "posterior_draw", "define"). With param_type set to define use arguments to be passed to `make_priors`; otherwise flat sets equal probabilities on each nodal type in each parameter set, prior_mean, prior_draw, posterior_mean, posterior_draw take parameters as the means or as draws from the prior or posterior.
- **warning**: Logical. Whether to warn about parameter renormalization.
- **normalize**: Logical. If parameter given for a subset of a family the residual elements are normalized so that parameters in param_set sum to 1 and provided params are unaltered.
- ... Options passed onto `make_priors`.

Value

A vector of draws from the prior or distribution of parameters

See Also

Other parameters: `get_parameters()`, `set_parameters()`

Examples

```r
# Simple examples
model <- make_model('X -> Y')
data <- simulate_data(model, n = 2)
model <- update_model(model, data)
make_parameters(model, parameters = c(.25, .75, 1.25, .25, .25, .25))
make_parameters(model, param_type = 'flat')
make_parameters(model, param_type = 'prior_draw')
make_parameters(model, param_type = 'prior_mean')
make_parameters(model, param_type = 'posterior_draw')
make_parameters(model, param_type = 'posterior_mean')

# Harder examples, using `define` and priors arguments to define
# specific parameters using causal syntax

# Using labels: Two values for two nodes with the same label
make_model('X -> M -> Y') %>% make_parameters(label = "01", parameters = c(0,1))

# Using statement:
make_model('X -> Y') %>%
  make_parameters(statement = c('Y[X=1]=Y[X=0]', parameters = c(.2,0)))
make_model('X -> Y') %>%
  make_parameters(statement = c('Y[X=1]>Y[X=0]', 'Y[X=1]<Y[X=0]', parameters = c(.2,0)))
```
# Normalize renormalizes values not set so that value set is not renormalized
make_parameters(make_model('X -> Y'),
    statement = 'Y[X=1]>Y[X=0]', parameters = .5)
make_parameters(make_model('X -> Y'),
    statement = 'Y[X=1]>Y[X=0]', parameters = .5, normalize = FALSE)

# May be built up
make_model('X -> Y') %>%
  set_confound(list(X = 'Y[X=1]>Y[X=0]')) %>
  set_parameters(confound = list(X='Y[X=1]>Y[X=0]',
    X='Y[X=1]<=Y[X=0]),
    parameters = list(c(.2, .8), c(.8, .2))) %>
  set_parameters(statement = 'Y[X=1]>Y[X=0]', parameters = .5) %>
get_parameters

---

**make_parameter_matrix**  
**Make parameter matrix**

**Description**

Calculate parameter matrix assuming no confounding. The parameter matrix maps from parameters into causal types. In models without confounding parameters correspond to nodal types.

**Usage**

```
make_parameter_matrix(model)
```

**Arguments**

- `model`  
  A causal_model. A model object generated by `make_model`.

**Value**

A data.frame, the parameter matrix, mapping from parameters to causal types

**Examples**

```r
model <- make_model('X -> Y')
make_parameter_matrix(model)
```
Description

A flexible function to generate priors for a model.

Usage

```r
make_priors(
    model,  
    alphas = NA,  
    distribution = NA,  
    node = NA,  
    label = NA,  
    statement = NA,  
    confound = NA,  
    nodal_type = NA,  
    param_names = NA,  
    param_set = NA
)
```

Arguments

- **model**: A model created with `make_model`
- **alphas**: Real positive numbers giving hyperparameters of the Dirichlet distribution
- **distribution**: String (or list of strings) indicating a common prior distribution (uniform, Jeffrey's or certainty)
- **node**: A string (or list of strings) indicating nodes for which priors are to be altered
- **label**: A string. Label for nodal type indicating nodal types for which priors are to be altered
- **statement**: A causal query (or list of queries) that determines nodal types for which priors are to be altered
- **confound**: A confound named list that restricts nodal types for which priors are to be altered. Adjustments are limited to nodes in the named list.
- **nodal_type**: A string. Label for nodal type indicating nodal types for which priors are to be altered
- **param_names**: A string. The name of specific parameter in the form of, for example, 'X.1', 'Y.01'
- **param_set**: A string. Indicates the name of the set of parameters to be modified (useful when setting confounds)
Details

Seven arguments govern *which* parameters should be altered. The default is ’all’ but this can be reduced by specifying

* label or nodal_type The label of a particular nodal type, written either in the form Y0000 or Y.Y0000
* node, which restricts for example to parameters associated with node ’X’
* statement, which restricts for example to nodal types that satisfy the statement ’Y[X=1] > Y[X=0]’
* confound, which restricts for example to nodal types that satisfy the statement ’Y[X=1] > Y[X=0]’
* param_set, which us useful when setting confound statements that produces several sets of parameters
* param_names, which restricts in specific parameters by naming them

Two arguments govern what values to apply:

* alphas is one or more non negative numbers and
* distribution indicates one of a common class: uniform, jeffreys, or ’certain’

Any arguments entered as lists or vectors of size > 1 should be of the same length as each other.

Value

A vector indicating the hyperparameters of the prior distribution of the nodal types.

For instance confound = list(X = Y[X=1]> Y[X=0]) adjust parameters on X that are conditional on nodal types for Y.

See Also

Other priors: get_priors(), make_par_values_multiple(), make_par_values(), make_values_task_list(), set_priors()

Examples

# Pass all nodal types
model <- make_model("Y <- X")
make_priors(model, alphas = .4)
make_priors(model, distribution = "jeffreys")

# Passing by names of node, parameter set or label
model <- make_model('X -> M -> Y')
make_priors(model, param_name = "X.1", alphas = 2)
make_priors(model, node = 'X', alphas = 3)
make_priors(model, param_set = 'Y', alphas = 5)
make_priors(model, node = c('X', 'Y'), alphas = 5)
make_priors(model, param_set = c('X', 'Y'), alphas = 5)
make_priors(model, node = list('X', 'Y'), alphas = list(3, 6))
make_priors(model, param_set = list('X', 'Y'), alphas = list(4, 6))
make_priors(model, node = c('X', 'Y'), distribution = c('certainty', 'jeffreys'))
make_prior_distribution

Make a prior distribution from priors

Description

Create a ‘n_param’x ‘n_draws’ database of possible lambda draws to be attached to the model.

Usage

make_prior_distribution(model, n_draws = 4000)

Arguments

model A causal_model. A model object generated by make_model.

n_draws A scalar. Number of draws.

Value

A ‘data.frame’ with dimension ‘n_param’x ‘n_draws’ of possible lambda draws
See Also

Other prior_distribution: `get_prior_distribution()`, `set_prior_distribution()`

Examples

```r
make_model('X -> Y') %>% make_prior_distribution(n_draws = 5)
```

---

### Description

A function to generate a list of parameter arguments.

### Usage

```r
make_values_task_list(
  distribution = NA,
  x = NA,
  node = NA,
  label = NA,
  statement = NA,
  confound = NA,
  nodal_type = NA,
  param_names = NA,
  param_set = NA
)
```

### Arguments

- **distribution**: A string (or list of strings) indicating a common prior distribution (uniform, jeffreys or certainty)
- **x**: Real positive numbers. For priors these are hyperparameters of the Dirichlet distribution. For parameters these are probabilities.
- **node**: A string (or list of strings) indicating nodes for which priors are to be altered
- **label**: A string. Label for nodal type indicating nodal types for which priors are to be altered
- **statement**: A causal query (or list of queries) that determines nodal types for which priors are to be altered
- **confound**: A confound named list that restricts nodal types for which priors are to be altered. Adjustments are limited to nodes in the named list.
- **nodal_type**: String. Label for nodal type indicating nodal types for which priors are to be altered
- **param_names**: String. The name of specific parameter in the form of, for example, ‘X.1’, ‘Y.01’
- **param_set**: String. Indicates the name of the set of parameters to be modified (useful when setting confounds)
### `non_decreasing`

**Value**

An array of parameter arguments.

For instance `confound = list(X = Y[X=1]> Y[X=0])` adjust parameters on X that are conditional on nodal types for Y.

**See Also**

Other priors: `get_priors()`, `make_par_values_multiple()`, `make_par_values()`, `make_priors()`, `set_priors()`

**Examples**

```r
CausalQueries:::make_values_task_list(node = '/quotesingle.Var X/quotesingle.Var', x = 3)
CausalQueries:::make_values_task_list(node = c('/quotesingle.Var X/quotesingle.Var', '/quotesingle.Var Y/quotesingle.Var'), x = 2:3)
CausalQueries:::make_values_task_list(node = c('/quotesingle.Var X/quotesingle.Var', '/quotesingle.Var Y/quotesingle.Var'), x = list(1, 2:4))
```

<table>
<thead>
<tr>
<th>non_decreasing</th>
<th>Make monotonicity statement (non negative)</th>
</tr>
</thead>
</table>

**Description**

Generate a statement for Y weakly monotonic (increasing) in X

**Usage**

```r
non_decreasing(X, Y)
```

**Arguments**

- **X**
  - A character. The quoted name of the input node
- **Y**
  - A character. The quoted name of the outcome node

**Value**

A character statement of class statement

**See Also**

Other statements: `complements()`, `decreasing()`, `increasing()`, `interacts()`, `non_increasing()`, `substitutes()`, `te()`

**Examples**

```r
non_decreasing('A', 'B')
```
**non_increasing**

*Make monotonicity statement (non positive)*

**Description**

Generate a statement for Y weakly monotonic (not increasing) in X

**Usage**

```r
non_increasing(X, Y)
```

**Arguments**

- `X` : A character. The quoted name of the input node
- `Y` : A character. The quoted name of the outcome node

**Value**

A character statement of class statement

**See Also**

Other statements: `complements()`, `decreasing()`, `increasing()`, `interacts()`, `non_decreasing()`, `substitutes()`, `te()`

**Examples**

```r
non_increasing('A', 'B')
```

---

**observe_data**

*Observe data, given a strategy*

**Description**

Observe data, given a strategy

**Usage**

```r
observe_data(
  complete_data,
  observed = NULL,
  nodes_to_observe = NULL,
  prob = 1,
  m = NULL,
  subset = NULL
)
```
query_distribution

Arguments

- complete_data: A data.frame. Data observed and unobserved.
- observed: A data.frame. Data observed.
- nodes_to_observe: A list. Nodes to observe.
- prob: A scalar. Observation probability.
- m: A integer. Number of units to observe; if specified, \(m\) overrides \(prob\).
- subset: A character. Logical statement that can be applied to rows of complete data. For instance observation for some nodes might depend on observed values of other nodes; or observation may only be sought if data not already observed!

Value

A data.frame with logical values indicating which nodes to observe in each row of `complete_data`.

Examples

```r
model <- make_model("X -> Y")
df <- simulate_data(model, n = 8)
# Observe X values only
observe_data(complete_data = df, nodes_to_observe = "X")
# Observe half the Y values for cases with observed X = 1
observe_data(complete_data = df,
             observed = observe_data(complete_data = df, nodes_to_observe = "X"),
             nodes_to_observe = "Y", prob = .5,
             subset = "X==1")
```

---

query_distribution Calculate query distribution

Description

Calculated distribution of a query from a prior or posterior distribution of parameters

Usage

```r
query_distribution(
    model,  
    query,  
    given = TRUE,  
    using = "priors",  
    parameters = NULL,  
    type_distribution = NULL,  
    verbose = FALSE,  
    join_by = "|
    )
```
query_model

Arguments

- **model**: A causal_model. A model object generated by `make_model`.
- **query**: A character. A query on potential outcomes such as "Y[X=1] - Y[X=0]"
- **given**: A character. A quoted expression evaluates to logical statement. given allows estimand to be conditioned on *observational* distribution.
- **using**: A character. Whether to use ‘priors’, ‘posteriors’ or ‘parameters’
- **parameters**: A vector of real numbers in [0,1]. A true parameter vector to be used instead of parameters attached to the model in case 'using' specifies 'parameters'
- **type_distribution**: A numeric vector. If provided saves calculation, otherwise calculated from model; may be based on prior or posterior
- **verbose**: Logical. Whether to print mean and standard deviation of the estimand on the console.
- **join_by**: A character. The logical operator joining expanded types when query contains wildcard (.). Can take values "&" (logical AND) or "|" (logical OR). When restriction contains wildcard (.) and join_by is not specified, it defaults to "|", otherwise it defaults to NULL.

Value

A vector of draws from the distribution of the potential outcomes specified in ‘query’

Examples

```r
model <- make_model("X -> Y") %>%
  set_prior_distribution()

distribution <- query_distribution(model, query = "(Y[X=1] - Y[X=0])")
distribution <- query_distribution(model, query = "(Y[X=1] - Y[X=0])", given = "X==1")
distribution <- query_distribution(model, query = "(Y[X=1] - Y[X=0])", given = "Y[X=1]==1")
distribution <- query_distribution(model, query = "(Y[X=1] > Y[X=0])")
distribution <- query_distribution(model, query = "(Y[X=.] == 1)", join_by = "&")
distribution <- query_distribution(model, query = "(Y[X=1] - Y[X=0])", using = "parameters")
df <- simulate_data(model, n = 3)
updated_model <- update_model(model, df)
query_distribution( updated_model , query = "(Y[X=1] - Y[X=0])", using = "posteriors")
```

query_model  Generate estimands dataframe

Description

Calculated from a parameter vector, from a prior or from a posterior distribution
Usage

query_model(
  model,
  queries = NULL,
  given = NULL,
  using = list("priors"),
  parameters = NULL,
  stats = NULL,
  digits = 3,
  n_draws = 4000,
  expand_grid = FALSE,
  query = NULL
)

Arguments

model A causal_model. A model object generated by make_model.
queries A vector of characters. Query on potential outcomes such as "Y[X=1] - Y[X=0]."
given A character. A quoted expression that evaluates to a logical statement. Allows estimand to be conditioned on *observational* (or counterfactual) distribution.
using A character. Whether to use ‘priors’, ‘posteriors’ or ‘parameters’.
parameters A vector of real numbers in [0,1]. Values of parameters to specify (optional). By default, parameters is drawn from model$parameters_df.
stats Functions to be applied to estimand distribution. If ‘NULL’, defaults to mean and standard deviation.
digits An integer. Decimal digits in output table.
n_draws An integer. Number of draws.
expand_grid Logical. If TRUE then all combinations of provided lists are examined. If not then each list is cycled through separately. Defaults to ‘FALSE’.
query alias for queries

Value

A data.frame with columns ‘Query’, ‘Given’ and ‘Using’ defined by corresponding input values. Further columns are generated as specified in ‘stats’.

Examples

model <- make_model("X -> Y") %>% set_prior_distribution(n_draws = 10000)
estimands_df <- query_model(
  model,
  query = list(ATE = "Y[X=1] - Y[X=0]", Share_positive = "Y[X=1] > Y[X=0]"),
  using = c("parameters", "priors"),
  expand_grid = TRUE)
```r
estimands_df <- query_model(  
  model,  
  query = list(ATE = "Y[X=1] - Y[X=0]", Share_positive = "Y[X=1] > Y[X=0]"),  
  using = c("parameters", "priors"),  
  expand_grid = FALSE)
```

```r
estimands_df <- query_model(  
  model,  
  using = list("parameters", "priors"),  
  query = list(ATE = "Y[X=1] - Y[X=0]", Is_B = "Y[X=1] > Y[X=0]"),  
  given = list(TRUE, "Y==0 & X==1"),  
  expand_grid = TRUE,  
  digits = 3)
```

```r
# An example: a stat representing uncertainty of token causation
token_var <- function(x) mean(x)*(1-mean(x))
estimands_df <- query_model(  
  model,  
  using = list("parameters", "priors"),  
  query = "Y[X=1] > Y[X=0]",  
  stats = c(mean = mean, sd = sd, token_var = token_var))
```

---

**reveal_outcomes**

**Reveal outcomes**

**Description**

Reveal outcomes for all causal types. Calculated by sequentially calculating endogenous nodes. If a do operator is applied to any node then it takes the given value and all its descendants are generated accordingly.

**Usage**

```
reveal_outcomes(model, dos = NULL, node = NULL)
```

**Arguments**

- `model` A causal_model. A model object generated by `make_model`.
- `dos` A named list. Do actions defining node values, e.g., `list(X = 0, M = 1)`.
- `node` A character. An optional quoted name of the node whose outcome should be revealed. If specified all values of parents need to be specified via dos.

**Details**

`reveal_outcomes` starts off by creating types (via `get_nodal_types`). It then takes types of endogenous and reveals their outcome based on the value that their parents took. Exogenous nodes outcomes correspond to their type.
Value

A data.frame object of revealed data for each node (columns) given causal / nodal type (rows).

Examples

```r
model <- make_model("X -> Y")
reveal_outcomes(model)

model <- make_model("X1->Y;X2->M;M->Y")
reveal_outcomes(model, dos = list(X1 = 1, M = 0))

model <- make_model("X->M->Y")
reveal_outcomes(model, dos = list(M = 1), node = "Y")
```

---

**set_ambiguities_matrix**

*Set ambiguity matrix*

Description

Add an ambiguities matrix to a model

Usage

```r
set_ambiguities_matrix(model, A = NULL)
```

Arguments

- `model`: A causal_model. A model object generated by `make_model`.
- `A`: A data.frame. Ambiguity matrix. Not required but may be provided to avoid repeated computation for simulations.

Value

An object of type causal_model with the ambiguities matrix attached

Examples

```r
model <- make_model('X -> Y') %>%
set_ambiguities_matrix()
model$A
```
Description

Adjust parameter matrix to allow confounding.

Usage

set_confound(model, confound = NULL, add_confounds_df = TRUE)

Arguments

model A causal_model. A model object generated by make_model.
confound A named list. It relates nodes to statements that identify causal types with which they are confounded
add_confounds_df Logical. Attach a dataframe with confound links. Defaults to TRUE.

Details

Confounding between X and Y arises when the nodal types for X and Y are not independently distributed. In the X -> Y graph, for instance, there are 2 nodal types for X and 4 for Y. There are thus 8 joint nodal types:

| | t^X | | | | | | | | Sum |
|-----|----|--------------------|--------------------|-----------|
| | | 0 | 1 | Pr(t^X=0 & t^Y=00) | Pr(t^X=1 & t^Y=00) | Pr(t^Y=00) |
|-----|----|--------------------|--------------------|-----------|
| t^Y | 00 | . | . | . | . | . |
| | 10 | . | . | . | . | . |
| | 01 | . | . | . | . | . |
| | 11 | . | . | . | . | . |
|-----|----|--------------------|--------------------|-----------|
| | Sum | Pr(t^X=0) | Pr(t^X=1) | 1 |

This table has 8 interior elements and so an unconstrained joint distribution would have 7 degrees of freedom. A no confounding assumption means that Pr(t^X | t^Y) = Pr(t^X), or Pr(t^X, t^Y) = Pr(t^X)Pr(t^Y). In this case there would be 3 degrees of freedom for Y and 1 for X, totalling 4 rather than 7.

set_confound lets you relax this assumption by increasing the number of parameters characterizing the joint distribution. Using the fact that P(A,B) = P(A)P(B|A) new parameters are introduced to capture P(BlA=a) rather than simply P(B).

The simplest way to allow for confounding is by adding a bidirected edge, such as via: set_confound(model, list("X <-> Y"). In this case the descendent node has a distribution conditional on the value of the ancestor node.
Ordering of conditioning can also be controlled however via `set_confound(model, list(X = 'Y'))` in which case X is given a distribution conditional on nodal types of Y.

More specific confounding statements are also possible using causal syntax. A statement of the form `list(X = 'Y[X=1]==1')` can be interpreted as: 'Allow X to have a distinct conditional distribution when Y has types that involve Y[X=1]==1.' In this case nodal types for Y would continue to have 3 degrees of freedom. But there would be parameters assigning the probability of X when t^Y = 01 or t^Y=11 and other parameters for residual cases. Thus 6 degrees of freedom in all. This is still short of an unconstrained distribution, though an unconstrained distribution can be achieved with repeated application of statements of this form, for instance via `list(X = 'Y[X=1]>Y[X=0]'), X = 'Y[X=1]==Y[X=0]').`

Similarly a statement of the form `list(Y = 'X==1')` can be interpreted as: 'Allow Y to have a distinct conditional distribution when X=1.' In this case there would be two distributions over nodal types for Y, producing 2*3 = 6 degrees of freedom. Nodal types for X would continue to have 1 degree of freedom. Thus 7 degrees of freedom in all, corresponding to a fully unconstrained joint distribution.

### Value

An object of class `causal_model`. It essentially returns a list containing the elements comprising a model (e.g. 'statement', 'nodal_types' and 'DAG') with the parameter matrix updated according to 'confound'.

### Examples

```r
model <- make_model('X -> Y') %>%
  set_confound(list('X <-> Y'))
get_parameters(model)

# In this case we notionally place a distribution but in fact Y has degenerate support
make_model('X -> Y -> Z') %>%
  set_restrictions(c(increasing('X', 'Y')), keep = TRUE) %>%
  set_confound(list('X <-> Y')) %>
  get_parameter_matrix()

# X nodes assigned conditional on Y
make_model('X -> Y') %>%
  set_confound(list(X = 'Y')) %>
  get_parameter_matrix()

# Y nodes assigned conditional on X
make_model('X -> Y') %>%
  set_confound(list(Y = 'X')) %>
  get_parameter_matrix()

model <- make_model('X -> Y', M -> Y)
set_confound(list(X = '(Y[X=1]>Y[X=0])**
M = 'Y',
```
\[ X = '(Y[X=1] < Y[X=0])') \]

confound = list(A = '(D[A=., B=1, C=1] > D[A=., B=0, C=0])')
model <- make_model('A -> B -> C -> D; B -> D') %>%
set_confound(confound = confound)

# Example where two parents are confounded
model <- make_model('A -> B <- C') %>%
  set_confound(list(A = 'C==1')) %>%
  set_parameters(c(0,1,1,0,.5,.5,rep(.0625, 16)))
cor(simulate_data(model, n = 20))

model <- make_model('X -> Y')
confound <- list(X = '(Y[X=1] > Y[X=0])', X = '(Y[X=1] == 1)')
model <- set_confound(model = model, confound = confound)

model <- make_model('X -> Y <- S; S -> W') %>%
  set_restrictions(c(
    increasing('X', 'Y'), increasing('S', 'W'),
    increasing('S', 'Y'), decreasing('S', 'Y')))
model1 <- set_confound(model, list(X = 'S==1', S = 'W[S=1]==1'), add_confounds_df = TRUE)
model1$confounds_df
model2 <- set_confound(model, list(S = 'X==1', S = 'W[S=1]==1'), add_confounds_df = TRUE)
model2$confounds_df

---

**set_confound**

**Set confounds**

**Description**

alias for set_confound. See set_confound.

**Usage**

set_confound(...)

**Arguments**

... arguments passed to set_confound

**Value**

An object of class causal_model. It essentially returns a list containing the elements comprising a model (e.g. 'statement', 'nodal_types' and 'DAG') with the parameter matrix updated according to 'confound'.
set_confounds_df

**Description**

Normally a confounds_df is added to a model whenever confounding is set. The confounds_df can be manually provided however using set_confounds_df.

**Usage**

```r
set_confounds_df(model)
```

**Arguments**

- `model`: A causal_model. A model object generated by `make_model`.

**Value**

An object of class causal_model. It essentially returns a list containing the elements comprising a model (e.g. 'statement', 'nodal_types' and 'DAG') with the confound_df attached to the parameter matrix in the model.

**Examples**

```r
model <- make_model('X -> Y') %>%
  set_confound(list('X <-> Y'), add_confounds_df = FALSE)
model$confounds_df
set_confounds_df(model)$confounds_df

# An example where a restriction is applied after a confounding relation is set removes
model <- make_model('X -> Y') %>%
  set_confound(list(X = '(Y[X=1] > Y[X=0])')) %>%
  set_restrictions('(Y[X=1] > Y[X=0])')
```

---

set_parameters

**Set parameters**

**Description**

Add a true parameter vector to a model. Parameters can be created using arguments passed to `make_parameters` and `make_priors`. 
Usage

`set_parameters(
  model,
  parameters = NULL,
  param_type = NULL,
  warning = FALSE,
  ...
)

Arguments

model | A causal_model. A model object generated by `make_model`.
parameters | A vector of real numbers in [0,1]. Values of parameters to specify (optional). By default, parameters is drawn from `model$parameters_df`.
param_type | A character. String specifying type of parameters to set (‘flat’, ‘prior_mean’, ‘posterior_mean’, ‘prior_draw’, ‘posterior_draw’, ‘define’). With param_type set to define use arguments to be passed to `make_priors`; otherwise flat sets equal probabilities on each nodal param_type in each parameter set; prior_mean, prior_draw, posterior_mean, posterior_draw take parameters as the means or as draws from the prior or posterior.
warning | Logical. Whether to warn about parameter renormalization
... | Arguments to be passed to `make_parameters`.

Details

Argument `param_type` is passed to `make_priors` and specifies one of ‘flat’, ‘prior_mean’, ‘posterior_mean’, ‘prior_draw’, ‘posterior_draw’, and ‘define’. With param_type set to define use arguments to be passed to `make_priors`; otherwise flat sets equal probabilities on each nodal param_type in each parameter set; prior_mean, prior_draw, posterior_mean, posterior_draw take parameters as the means or as draws from the prior or posterior.

Value

An object of class causal_model. It essentially returns a list containing the elements comprising a model (e.g. ‘statement’, ‘nodal_types’ and ‘DAG’) with true vector of parameters attached to it.

See Also

Other parameters: `get_parameters()`, `make_parameters()`

Examples

```r
make_model(’X->Y’) %>% set_parameters(1:6) %>% get_parameters()

make_model(’X -> Y’) %>%
set_confound(list(X = ’Y[X=1]>Y[X=0]’)) %>%
set_parameters(confound = list(X=’Y[X=1]>Y[X=0]’, X=’Y[X=1]<=Y[X=0]’),
  parameters = list(c(.2, .8), c(.8, .2))) %>%
```
set_parameter_matrix


```r
set_parameters(statement = 'Y[X=1]>Y[X=0]', parameters = .5) %>%
get_parameters
```

---

**Description**

Add a parameter matrix to a model

**Usage**

```r
set_parameter_matrix(model, P = NULL)
```

**Arguments**

- `model`: A `causal_model`. A model object generated by `make_model`.
- `P`: A `data.frame`. Parameter matrix. Not required but may be provided to avoid repeated computation for simulations.

**Value**

An object of class `causal_model`. It essentially returns a list containing the elements comprising a model (e.g. 'statement', 'nodal_types' and 'DAG') with the parameter matrix attached to it.

**Examples**

```r
model <- make_model('X -> Y')
P <- diag(8)
colnames(P) <- rownames(model$causal_types)
model <- set_parameter_matrix(model, P = P)
```

---

**set_priors**

**Set prior distribution**

**Description**

A flexible function to add priors to a model.
Usage

```r
set_priors(
    model,
    priors = NULL,
    distribution = NA,
    alphas = NA,
    node = NA,
    label = NA,
    statement = NA,
    confound = NA,
    nodal_type = NA,
    param_names = NA,
    param_set = NA
)
```

Arguments

- `model`: A model created with `make_model`
- `priors`: A optional vector of positive reals indicating priors over all parameters. These are interpreted as arguments for Dirichlet distributions—one for each parameter set. To see the structure of parameter sets examine `model$parameters_df`
- `distribution`: String (or list of strings) indicating a common prior distribution (uniform, jefreys or certainty)
- `alphas`: Real positive numbers giving hyperparameters of the Dirichlet distribution
- `node`: A string (or list of strings) indicating nodes for which priors are to be altered
- `label`: String. Label for nodal type indicating nodal types for which priors are to be altered
- `statement`: A causal query (or list of queries) that determines nodal types for which priors are to be altered
- `confound`: A confound statement (or list of statements) that restricts nodal types for which priors are to be altered
- `nodal_type`: String. Label for nodal type indicating nodal types for which priors are to be altered
- `param_names`: String. The name of specific parameter in the form of, for example, ’X.1’, ’Y.01’
- `param_set`: String. Indicates the name of the set of parameters to be modified (useful when setting confounds)

Details

Four arguments govern *which* parameters should be altered. The default is ’all’ but this can be reduced by specifying

* `label`: The label of a particular nodal type, written either in the form Y0000 or Y.Y0000
* `node`: which restricts for example to parameters associated with node ’X’
* `statement`: which restricts for example to nodal types that satisfy the statement ’Y[X=1] > Y[X=0]’
* confound, which restricts for example to nodal types that satisfy the statement ‘Y[X=1] > Y[X=0]’

Two arguments govern what values to apply:

* alphas is one or more non negative numbers and
* distribution indicates one of a common class: uniform, jeffreys, or 'certain'

Any arguments entered as lists or vectors of size > 1 should be of the same length as each other.

For more examples and details see \texttt{make_priors}

\textbf{Value}

An object of class \texttt{causal_model}. It essentially returns a list containing the elements comprising a model (e.g. 'statement', 'nodal_types' and 'DAG') with the 'priors' attached to it.

\textbf{See Also}

Other priors: \texttt{get_priors()}, \texttt{make_par_values_multiple()}, \texttt{make_par_values()}, \texttt{make_priors()}, \texttt{make_values_task_list()}

\textbf{Examples}

```r
library(dplyr)
# Set priors to the model
model <- make_model("X -> Y") %>%
  set_priors(alphas = 3)
get_priors(model)
model <- make_model("X -> Y") %>%
  set_priors(distribution = "jeffreys")
get_priors(model)

# Pass all nodal types
model <- make_model("Y <- X") %>%
  set_priors(.4)
get_priors(model)
model <- make_model("Y <- X") %>%
  set_priors(.7)
get_priors(model)
model <- make_model("Y <- X") %>%
  set_priors(distribution = "jeffreys")
get_priors(model)

# Passing by names of node, parameter set or label
model <- make_model("X -> M -> Y")
model_new_priors <- set_priors(model, param_name = "X.1", alphas = 2)
get_priors(model_new_priors)
model_new_priors <- set_priors(model, node = "X", alphas = 3)
get_priors(model_new_priors)
model_new_priors <- set_priors(model, param_set = "Y", alphas = 5)
get_priors(model_new_priors)
model_new_priors <- set_priors(model, node = c("X", "Y"), alphas = 3)
get_priors(model_new_priors)
```
model_new_priors <- set_priors(model, param_set = c('X', 'Y'), alphas = 5)
get_priors(model_new_priors)
model_new_priors <- set_priors(model, node = list('X', 'Y'), alphas = list(3, 6))
get_priors(model_new_priors)
model_new_priors <- set_priors(model, param_set = list('X', 'Y'), alphas = list(4, 6))
get_priors(model_new_priors)
model_new_priors <- set_priors(model, node = c('X', 'Y'), distribution = c('certainty', 'jeffreys'))
get_priors(model_new_priors)
model_new_priors <- set_priors(model, statement = c('Y[X=1] > Y[X=0]', 'M[X=1]== M[X=0]'), alphas = c(2, .5))
get_priors(model_new_priors)

# A more complex example
model <- make_model('X -> Y') %>%
  set_confound(list(X = 'Y[X=1] > Y[X=0]'))%>
  set_priors(statement = 'X[=1]',
              confound = list(X = 'Y[X=1] > Y[X=0]', X = 'Y[X=1]<Y[X=0]'),
              alphas = c(2.65))
get_priors(model)
**set_prior_distribution**  

*Add prior distribution draws*

**Description**

Add 'n_param x n_draws' database of possible lambda draws to the model.

**Usage**

```
set_prior_distribution(model, n_draws = 4000)
```

**Arguments**

- **model**: A causal_model. A model object generated by `make_model`.
- **n_draws**: A scalar. Number of draws.

**Value**

An object of class causal_model. It essentially returns a list containing the elements comprising a model (e.g. 'statement', 'nodal_types' and 'DAG') with the 'prior_distribution' attached to it.

**See Also**

Other prior_distribution: `get_prior_distribution()`, `make_prior_distribution()`

**Examples**

```
make_model('X -> Y') %>% set_prior_distribution(n_draws = 5) %>% get_prior_distribution()
```

---

**set_restrictions**  

*Restrict a model*

**Description**

Restrict a model’s parameter space. This reduces the number of nodal types and in consequence the number of unit causal types.

**Usage**

```
set_restrictions(
  model,
  statement = NULL,
  join_by = '|',
  labels = NULL,
  keep = FALSE
)
```
Arguments

model A causal_model. A model object generated by make_model.

statement A quoted expressions defining the restriction. If values for some parents are not
specified, statements should be surrounded by parentheses, for instance \((Y[A = 1] > Y[A=0])\) will be interpreted for all combinations of other parents of \(Y\) set
at possible levels they might take.

join_by A string. The logical operator joining expanded types when statement contains
wildcard (\(\_\)). Can take values \('&'\) (logical AND) or \('|'\) (logical OR). When
restriction contains wildcard (\(\_\)) and join_by is not specified, it defaults to \('|'\),
otherwise it defaults to NULL. Note that join_by joins within statements, not
across statements.

labels A list of character vectors specifying nodal types to be kept or removed from the
model. Use get_nodal_types to see syntax. Note that labels gets overwritten
by statement if statement is not NULL.

keep Logical. If ‘FALSE’, removes and if ‘TRUE’ keeps only causal types specified
by statement or labels.

Details

Restrictions are made to nodal types, not to unit causal types. Thus for instance in a model \(X \rightarrow M \rightarrow Y\), one cannot apply a simple restriction so that \(Y\) is nondecreasing in \(X\), however one can restrict so
that \(M\) is nondecreasing in \(X\) and \(Y\) nondecreasing in \(M\). To have a restriction that \(Y\) be nondecreasing
in \(X\) would otherwise require restrictions on causal types, not nodal types, which implies a form of
undeclared confounding (i.e. that in cases in which \(M\) is decreasing in \(X\), \(Y\) is decreasing in \(M\)).

Since restrictions are to nodal types, all parents of a node are implicitly fixed. Thus for model
make_model(\(X \rightarrow Y <-W\)) the request set_restrictions(\(\{Y[X=1] == 0\}\)) is interpreted as
set_restrictions(\(\{Y[X=1,W=0] == 0 | Y[X=1,W=1] == 0\}\)).

Statements with implicitly controlled nodes should be surrounded by parentheses, as in these ex-
amples.

Note that prior probabilities are redistributed over remaining types.

Value

An object of class model. The causal types and nodal types in the model are reduced according to
the stated restriction.

See Also

Other restrictions: restrict_by_labels(), restrict_by_query()

Examples

# 1. Restrict parameter space using statements
model <- make_model('X->Y') %>%
  set_restrictions(statement = c('X[1] == 0'))
model <- make_model('X->Y') %>%
  set_restrictions(non_increasing('X', 'Y'))

model <- make_model('X -> Y <- W') %>%
  set_restrictions(decreasing('X', 'Y'))

model$parameters_df

model <- make_model('X->Y') %>%
  set_restrictions(statement = decreasing('X', 'Y'))

model$parameters_df

model <- make_model('X->Y') %>%
  set_restrictions(c(increasing('X', 'Y'), decreasing('X', 'Y')))

model$parameters_df

# Restrict to define a model with monotonicity
model <- make_model('X->Y') %>%
  set_restrictions(statement = c('Y[X=1] < Y[X=0]'))

get_parameter_matrix(model)

# Restrict to a single type in endogenous node
model <- make_model('X->Y') %>%
  set_restrictions(statement = '(Y[X = 1] == 1)',
                   join_by = '&', keep = TRUE)

get_parameter_matrix(model)

# Use of | and &
# Keep node if *for some value of B* Y[A = 1] == 1
model <- make_model('A->Y<->B') %>%
  set_restrictions(statement = '(Y[A = 1] == 1)',
                   join_by = '|', keep = TRUE)

dim(get_parameter_matrix(model))

# Keep node if *for all values of B* Y[A = 1] == 1
model <- make_model('A->Y<->B') %>%
  set_restrictions(statement = '(Y[A = 1] == 1)',
                   join_by = '&', keep = TRUE)

dim(get_parameter_matrix(model))

# Restrict multiple nodes
model <- make_model('X->Y<->M; X -> M') %>%
  set_restrictions(statement = c('(Y[X = 1] == 1)', '(M[X = 1] == 1)'),
                     join_by = '&', keep = TRUE)

get_parameter_matrix(model)

# Restrictions on levels for endogenous nodes aren't allowed
## Not run:
model <- make_model('X->Y') %>%
  set_restrictions(statement = '(Y == 1)')

## End(Not run)
# 2. Restrict parameter space Using labels:
model <- make_model('X->Y') %>%
set_restrictions(labels = list(X = '0', Y = '00'))

# Restrictions can be with wildcards
model <- make_model('X->Y') %>%
set_restrictions(labels = list(Y = '?0'))
get_parameter_matrix(model)

# Running example: there are only four causal types
model <- make_model('S -> C -> Y <- R <- X; X -> C -> R') %>%
set_restrictions(labels = list(C = '1000', R = '0001', Y = '0001'), keep = TRUE)
get_parameter_matrix(model)

simulate_data

simulate_data is an alias for make_data

Description

simulate_data is an alias for make_data

Usage

simulate_data(...)  

Arguments

... arguments for make_model

Value

A data.frame with simulated data.

Examples

simulate_data(make_model("X->Y"))
substitutes

**Description**

Generate a statement for X1, X1 substitute for each other in the production of Y

**Usage**

`substitutes(X1, X2, Y)`

**Arguments**

- **X1** A character. The quoted name of the input node 1.
- **X2** A character. The quoted name of the input node 2.
- **Y** A character. The quoted name of the outcome node.

**Value**

A character statement of class `statement`

**See Also**

Other statements: `complements()`, `decreasing()`, `increasing()`, `interacts()`, `non_decreasing()`, `non_increasing()`, `te()`

**Examples**

```r
get_query_types(model = make_model('A -> B <- C'),
    query = substitutes('A', 'C', 'B'), map = "causal_type")

query_model(model = make_model('A -> B <- C'),
    queries = substitutes('A', 'C', 'B'),
    using = 'parameters')
```
Make treatment effect statement (positive)

Description
Generate a statement for \((Y(1) - Y(0))\). This statement when applied to a model returns an element in \((1,0,-1)\) and not a set of cases. This is useful for some purposes such as querying a model, but not for uses that require a list of types, such as set_restrictions.

Usage
\(\text{te}(X, Y)\)

Arguments
\begin{itemize}
\item \(X\) A character. The quoted name of the input node
\item \(Y\) A character. The quoted name of the outcome node
\end{itemize}

Value
A character statement of class statement

See Also
Other statements: \texttt{complements()}, \texttt{decreasing()}, \texttt{increasing()}, \texttt{interacts()}, \texttt{non_decreasing()}, \texttt{non_increasing()}, \texttt{substitutes()}

Examples
\begin{verbatim}
te('A', 'B')
model <- make_model('X->Y') %>% set_restrictions(increasing('X', 'Y')) query_model(model, list(ate = te('X', 'Y')), using = 'parameters')

# set_restrictions breaks with te because it requires a listing
# of causal types, not numeric output.

## Not run:
model <- make_model('X->Y') %>% set_restrictions(te('X', 'Y'))

## End(Not run)
\end{verbatim}
**update_model**  

*Fit causal model using 'stan'*

**Description**

Takes a model and data and returns a model object with data attached and a posterior model

**Usage**

```r
update_model(model, data = NULL, data_type = "long", keep_fit = FALSE, ...)
```

**Arguments**

- `model`  
  A causal_model. A model object generated by `make_model`.  
- `data`  
  A data.frame. Data of nodes that can take three values: 0, 1, and NA. In long form as generated by `make_events`  
- `data_type`  
  Either 'long' (as made by `simulate_data`) or 'compact' (as made by `collapse_data`). Compact data must have entries for each member of each strategy family to produce a valid simplex.  
- `keep_fit`  
  Logical. Whether to append the stanfit object to the model. Defaults to 'FALSE'  
- `...`  
  Options passed onto `stan` call.

**Value**

An object of class causal_model. It essentially returns a list containing the elements comprising a model (e.g. 'statement', 'nodal_types' and 'DAG') with the 'posterior_distribution' returned by stan attached to it.

**Examples**

```r
model <- make_model(\'X->Y\')
data_long <- simulate_data(model, n = 4)
data_short <- collapse_data(data_long, model)

model_1 <- update_model(model, data_long)

## Not run:
# Throws error unless compact data indicated:
model_3 <- update_model(model, data_short)

## End(Not run)

model_4 <- update_model(model, data_short, data_type = \'compact\')
```
# It is possible to implement updating without data, in which case the posterior
# is a stan object that reflects the prior
model5 <- update_model(model)

# Advanced: Example of a model with tailored parameters.
# We take a model and add a tailored P matrix (which maps from parameters
# to causal types) and a tailored parameters_df which reports that
# all parameters are in one family.
# Parameters in this example are not connected with nodal types in any way.
model <- make_model('X->Y')
P <- diag(8)
colnames(P) <- rownames(model$causal_types)
model <- set_parameter_matrix(model, P = P)
model$parameters_df <- data.frame(
  param_names = paste0('x',1:8),
  param_set = 1, priors = 1, parameters = 1/8)

# Update fully confounded model on strongly correlated data

data <- make_data(make_model('X->Y'), n = 100,
  parameters = c(.5, .5, .1,.1,.7,.1))
fully_confounded <- update_model(model, data, keep_fit = TRUE)
fully_confounded$stan_fit
query_model(fully_confounded, 'Y[X = 1] > Y[X=0]', using = 'posteriors')
# To see the confounding:
with(fully_confounded$posterior_distribution %>% data.frame(),
  {par(mfrow = c(1,2))
   plot(x1, x5, main = 'joint distribution of X0.Y00, X0.Y01')
   plot(x1, x6, main = 'joint distribution of X0.Y00, X1.Y01'))
# Index

*Topic **datasets**
- democracy_data, 8

all_data_types, 3

CausalQueries-package, 3
collapse_data, 4, 57
complements, 6, 8, 21, 22, 35, 36, 55, 56
dagitty, 27
data_type_names, 7
decreasing, 6, 7, 21, 22, 35, 36, 55, 56
democracy_data, 8
expand_data, 9
expand_wildcard, 9

get_ambiguities_matrix, 10
get_causal_types, 11
get_event_prob, 11
get_nodal_types, 12, 40
get_parameter_matrix, 14
get_parameter_names, 14
get_parameters, 13, 29, 46
get_parents, 15
get_prior_distribution, 17, 34, 51
get_priors, 16, 32, 35, 49
get_query_types, 17
get_type_prob, 19
get_type_prob_multiple, 20

increasing, 6, 8, 21, 22, 35, 36, 55, 56
interacts, 6, 8, 21, 22, 35, 36, 55, 56
interpret_type, 22

make_confound, 42
make_confound_df, 45

make_data, 24, 57
make_events, 5, 7, 26, 57
make_model, 4, 5, 7, 9–20, 22, 23, 25, 26, 27, 29, 30, 33, 38–42, 45–47, 51, 52, 54, 57

make_par_values, 16, 32, 35, 49
make_par_values_multiple, 16, 32, 35, 49
make_parameter_matrix, 30
make_parameters, 13, 28, 45, 46
make_prior_distribution, 17, 33, 51
make_priors, 16, 29, 31, 35, 45, 49
make_values_task_list, 16, 32, 34, 49

non_decreasing, 6, 8, 21, 22, 35, 36, 55, 56
non_increasing, 6, 8, 21, 22, 35, 36, 55, 56
observe_data, 36

query_distribution, 37
query_model, 38

restrict_by_labels, 52
restrict_by_query, 52
reveal_outcomes, 18, 40

set_ambiguities_matrix, 41
set_confound, 42
set_confound_df, 45
set_parameter_matrix, 47
set_parameters, 13, 29, 45
set_prior_distribution, 17, 34, 51
set_priors, 16, 32, 35, 47
set_restrictions, 51
simulate_data, 54, 57
stan, 57
stanfit, 57
substitutes, 6, 8, 21, 22, 35, 36, 55, 56

te, 6, 8, 21, 22, 35, 36, 55, 56
update_model, 57