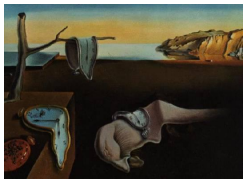


Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *doi: 10.18637/jss.v000.i00*

Making, Updating, and Querying Causal Models using `CausalQueries`

Till Tietz
WZB

Lily Medina
UC Berkeley

Macartan Humphreys
WZB

Abstract

A guide to the R package `CausalQueries` for making, updating, and querying causal models

Keywords: causal models, stan, bayes.

1. Introduction: Causal models

- Embed the *methods* and the *software* into the respective relevant literature.
- For the latter both competing and complementary software should be discussed (within the same software environment and beyond), bringing out relative (dis)advantages. All software mentioned should be properly `@cited`'d. (See also Using BibTeX for more details on BibTeX.)

Learning about causal models…

In R, …

The strength of `CausalQueries` The limitation of `CausalQueries`

2. Models and software

Nodes and nodal types

3. Making models

A model is defined in one step using a `dagitty` syntax in which the structure of the model is provided as a statement.

For instance:

```
model <- make_model("X -> M -> Y <- X")
```

The statement (in quotes) provides the names of nodes. An arrow ("`->`" or "`<-`") connecting nodes indicates that one node is a potential cause of another (that is, whether a given node is a "parent" or "child" of another.

Formally a statement like this is interpreted as:

1. Functional equations:

- $Y = f(M, X, \theta^Y)$
- $M = f(X, \theta^M)$
- $X = \theta^X$

2. Distributions on shocks:

- $\Pr(\theta^i = \theta^i_k) = \lambda^i_k$

3. Independence assumptions:

- $\theta_i \perp\!\!\!\perp \theta_j, i \neq j$

where function f maps from the set of possible values of the parents of i to values of node i given θ^i. Units with the same value on θ^i react in the same way to the parents of i. Indeed in this discrete world we think of θ^i as fully dictating the functional form of f, indicating what outcome is observed on i for any value of i's parents.

In addition, it is also possible to indicate "unobserved confounding", that is, the presence of an unobserved variable that might influence observed variables. In this case condition 3 above is relaxed. We describe how this is done in greater detail in section @ref(confounding).

For instance:

```
model <- make_model("X -> Y <- W -> X; X <-> Y")
```

3.1. Graphing

Once defined, a model can be graphed (we use the `dagitty` package for this):

```
model <- make_model("X -> Y <- W -> X; X <-> Y") |>
  plot()
```

This is useful to check that you have written the structure down correctly.

3.2. Model characterization

When a model is defined, a set of objects are generated. These are the key quantities that are used for all inference.

3.3. Nodal types

Two units have the same *nodal type* at node Y, θ^Y, if their outcome at Y responds in the same ways to parents of Y.

A node with k parents has 2^{2^k} nodal types. The reason is that with k parents, there are 2^k possible values of the parents and so 2^{2^k} ways to respond to these possible parental values. As a convention we say that a node with no parents has two nodal types (0 or 1).

When a model is created the full set of "nodal types" is identified. These are stored in the model. The subscripts become very long and hard to parse for more complex models so the model object also includes a guide to interpreting nodal type values. You can see them like this.

```
make_model("X -> Y")$nodal_types
```

```
$X
[1] "0" "1"

$Y
[1] "00" "10" "01" "11"

attr(,"interpret")
attr(,"interpret")$X
  node position display interpretation
1    X       NA      X0          X = 0
2    X       NA      X1          X = 1

attr(,"interpret")$Y
  node position display interpretation
1    Y        1   Y[*]*      Y | X = 0
2    Y        2   Y*[*]      Y | X = 1
```

Note that we use θ^j to indicate nodal types because for qualitative analysis the nodal types are often the parameters of interest.

3.4. Causal types

Causal types are collections of nodal types. Two units are of the same *causal type* if they have the same nodal type at every node. For example in a $X \rightarrow M \rightarrow Y$ model, $\theta = (\theta_0^X, \theta_{01}^M, \theta_{10}^Y)$ is a type that has $X = 0$, M responds positively to X, and Y responds positively to M.

When a model is created, the full set of causal types is identified. These are stored in the model object:

```
make_model("A -> B")$causal_types
```

```
       A  B
A0.B00 0 00
A1.B00 1 00
A0.B10 0 10
A1.B10 1 10
A0.B01 0 01
A1.B01 1 01
A0.B11 0 11
A1.B11 1 11
```

A model with n_j nodal types at node j has $\prod_j n_j$ causal types. Thus the set of causal types can be large. In the model ($X \rightarrow M \rightarrow Y \leftarrow X$) there are $2 \times 4 \times 16 = 128$ causal types.

Knowledge of a causal type tells you what values a unit would take, on all nodes, absent an intervention. For example for a model $X \rightarrow M \rightarrow Y$ a type $\theta = (\theta_0^X, \theta_{01}^M, \theta_{10}^Y)$ would imply data $(X = 0, M = 0, Y = 1)$. (The converse of this, of course, is the key to updating: observation of data $(X = 0, M = 0, Y = 1)$ result in more weight placed on θ_0^X, θ_{01}^M, and θ_{10}^Y).)

3.5. Parameters dataframe

When a model is created, `CausalQueries` attaches a "parameters dataframe" which keeps track of model parameters, which belong together in a family, and how they relate to causal types. This becomes especially important for more complex models with confounding that might involve more complicated mappings between parameters and nodal types. In the case with no confounding the nodal types *are* the parameters; in cases with confounding you generally have more parameters than nodal types.

For instance:

```
make_model("X->Y")$parameters_df %>%
  kable()
```

param_names	node	gen	param_set	nodal_type	given	param_value	priors
X.0	X	1	X	0		0.50	1
X.1	X	1	X	1		0.50	1
Y.00	Y	2	Y	00		0.25	1
Y.10	Y	2	Y	10		0.25	1
Y.01	Y	2	Y	01		0.25	1
Y.11	Y	2	Y	11		0.25	1

Each row in the dataframe corresponds to a single parameter.

The columns of the parameters data frame are understood as follows:

- `param_names` gives the name of the parameter, in shorthand. For instance the parameter $\lambda_0^X = \Pr(\theta^X = \theta_0^X)$ has `par_name X.0`. See section @ref(notation) for a summary of notation.

- `param_value` gives the (possibly default) parameter values. These are probabilities.

- `param_set` indicates which parameters group together to form a simplex. The parameters in a set have parameter values that sum to 1. In this example $\lambda_0^X + \lambda_1^X = 1$.
- `node` indicates the node associated with the parameter. For parameter `\lambda^X_0` this is X.
- `nodal_type` indicates the nodal types associated with the parameter.
- `gen` indicates the place in the partial causal ordering (generation) of the node associated with the parameter
- `priors` gives (possibly default) Dirichlet priors arguments for parameters in a set. Values of 1 (.5) for all parameters in a set implies uniform (Jeffrey's) priors over this set.

Below we will see examples where the parameter dataframe helps keep track of parameters that are created when confounding is added to a model.

3.6. Parameter matrix

The parameters dataframe keeps track of parameter values and priors for parameters but it does not provide a mapping between parameters and the probability of causal types.

The parameter matrix (P matrix) can be added to the model to provide this mapping. The P matrix has a row for each parameter and a column for each causal type. For instance:

```
make_model("X->Y") %>% get_parameter_matrix %>%
  kable
```

	X0.Y00	X1.Y00	X0.Y10	X1.Y10	X0.Y01	X1.Y01	X0.Y11	X1.Y11
X.0	1	0	1	0	1	0	1	0
X.1	0	1	0	1	0	1	0	1
Y.00	1	1	0	0	0	0	0	0
Y.10	0	0	1	1	0	0	0	0
Y.01	0	0	0	0	1	1	0	0
Y.11	0	0	0	0	0	0	1	1

The probability of a causal type is given by the product of the parameters values for parameters whose row in the P matrix contains a 1.

Below we will see examples where the P matrix helps keep track of parameters that are created when confounding is added to a model.

3.7. Setting restrictions

When a model is defined, the complete set of possible causal relations are worked out. This set can be very large.

Sometimes for theoretical or practical reasons it is useful to constrain the set of types. In `CausalQueries` this is done at the level of nodal types, with restrictions on causal types following from restrictions on nodal types.

For instance to impose an assumption that Y is not decreasing in X we generate a restricted model as follows:

```
model <- make_model("X->Y") %>% set_restrictions("Y[X=1] < Y[X=0]")
```

or:

```
model <- make_model("X->Y") %>% set_restrictions(decreasing("X", "Y"))
```

Viewing the resulting parameter matrix we see that both the set of parameters and the set of causal types are now restricted:

```
get_parameter_matrix(model)
```

```
Rows are parameters, grouped in parameter sets

Columns are causal types

Cell entries indicate whether a parameter probability is used
in the calculation of causal type probability

     X0.Y00 X1.Y00 X0.Y01 X1.Y01 X0.Y11 X1.Y11
X.0       1      0      1      0      1      0
X.1       0      1      0      1      0      1
Y.00      1      1      0      0      0      0
Y.01      0      0      1      1      0      0
Y.11      0      0      0      0      1      1


 param_set  (P)
```

Here and in general, setting restrictions typically involves using causal syntax; see Section @ref(syntax) for a guide the syntax used by `CausalQueries`.

Note:

- Restrictions have to operate on nodal types: restrictions on *levels* of endogenous nodes aren't allowed. This, for example, will fail: `make_model("X->Y") %>% set_restrictions(statement = "(Y == 1)")`. The reason is that it requests a correlated restriction on nodal types for `X` and `Y` which involves undeclared confounding.
- Restrictions implicitly assume fixed values for *all* parents of a node. For instance: `make_model("A -> B <- C") %>% set_restrictions("(B[C=1]==1)")` is interpreted as shorthand for the restriction `"B[C = 1, A = 0]==1 | B[C = 1, A = 1]==1"`.
- To place restrictions on multiple nodes at the same time, provide these as a vector of restrictions. This is not permitted: `set_restrictions("Y[X=1]==1 & X==1")`, since it requests correlated restrictions. This however is allowed: `set_restrictions(c("Y[X=1]==1", "X==1"))`.

- Use the `keep` argument to indicate whether nodal types should be dropped (default) or retained.
- Restrictions can be set using nodal type labels. `make_model("S -> C -> Y <- R <- X; X -> C -> R") %>% set_restrictions(labels = list(C = "C1000", R = "R0001", Y = "Y0001"), keep = TRUE)`
- Wild cards can be used in nodal type labels: `make_model("X->Y") %>% set_restrictions(labels = list(Y = "Y?0"))`

3.8. Allowing confounding

(Unobserved) confounding between two nodes arises when the nodal types for the nodes are not independently distributed.

In the $X \rightarrow Y$ graph, for instance, there are 2 nodal types for X and 4 for Y. There are thus 8 joint nodal types (or causal types):

		θ^X		
		0	1	Sum
θ^Y	00	$\Pr(\theta^X_0, \theta^Y_{00})$	$\Pr(\theta^X_1, \theta^Y_{00})$	$\Pr(\theta^Y_{00})$
	10	$\Pr(\theta^X_0, \theta^Y_{10})$	$\Pr(\theta^X_1, \theta^Y_{10})$	$\Pr(\theta^Y_{10})$
	01	$\Pr(\theta^X_0, \theta^Y_{01})$	$\Pr(\theta^X_1, \theta^Y_{01})$	$\Pr(\theta^Y_{01})$
	11	$\Pr(\theta^X_0, \theta^Y_{11})$	$\Pr(\theta^X_1, \theta^Y_{11})$	$\Pr(\theta^Y_{11})$
	Sum	$\Pr(\theta^X_0)$	$\Pr(\theta^X_1)$	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 $\Pr(\theta^X|\theta^Y) = \Pr(\theta^X)$, or $\Pr(\theta^X, \theta^Y) = \Pr(\theta^X)\Pr(\theta^Y)$. In this case we just put a distribution on the marginals and there would be 3 degrees of freedom for Y and 1 for X, totaling 4 rather than 7.

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

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 descendant node has a distribution conditional on the value of the ancestor node. To wit:

```
confounded <- make_model("X->Y") %>%
  set_confound("X <-> Y")

confounded$parameters_df %>% kable
```

param_names	node	gen	param_set	nodal_type	given	param_value	priors
X.0	X	1	X	0		0.50	1
X.1	X	1	X	1		0.50	1
Y.00_X.0	Y	2	Y.X.0	00	X.0	0.25	1

param_names	node	gen	param_set	nodal_type	given	param_value	priors
Y.10_X.0	Y	2	Y.X.0	10	X.0	0.25	1
Y.01_X.0	Y	2	Y.X.0	01	X.0	0.25	1
Y.11_X.0	Y	2	Y.X.0	11	X.0	0.25	1
Y.00_X.1	Y	2	Y.X.1	00	X.1	0.25	1
Y.10_X.1	Y	2	Y.X.1	10	X.1	0.25	1
Y.01_X.1	Y	2	Y.X.1	01	X.1	0.25	1
Y.11_X.1	Y	2	Y.X.1	11	X.1	0.25	1

We see here that there are now two parameter families for parameters associated with the node Y. Each family captures the conditional distribution of Y's nodal types, given X. For instance the parameter `Y01_X.1` can be interpreted as $\Pr(\theta^Y = \theta^Y_{01}|X = 1)$.

To see exactly how the parameters map to causal types we can view the parameter matrix:

```
get_parameter_matrix(confounded) %>% kable
```

	X0.Y00	X1.Y00	X0.Y10	X1.Y10	X0.Y01	X1.Y01	X0.Y11	X1.Y11
X.0	1	0	1	0	1	0	1	0
X.1	0	1	0	1	0	1	0	1
Y.00_X.0	1	0	0	0	0	0	0	0
Y.10_X.0	0	0	1	0	0	0	0	0
Y.01_X.0	0	0	0	0	1	0	0	0
Y.11_X.0	0	0	0	0	0	0	1	0
Y.00_X.1	0	1	0	0	0	0	0	0
Y.10_X.1	0	0	0	1	0	0	0	0
Y.01_X.1	0	0	0	0	0	1	0	0
Y.11_X.1	0	0	0	0	0	0	0	1

Importantly, the P matrix works as before, despite confounding. We can assess the probability of causal types by multiplying the probabilities of the constituent parameters.

Note:

- 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 $Y(do(X = 1)) = 1$." 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 $\theta^Y = \theta^Y_{01}$ or $\theta^Y = \theta^Y_{11}$ and other parameters for residual cases. Thus 6 degrees of freedom in all.

  - 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.

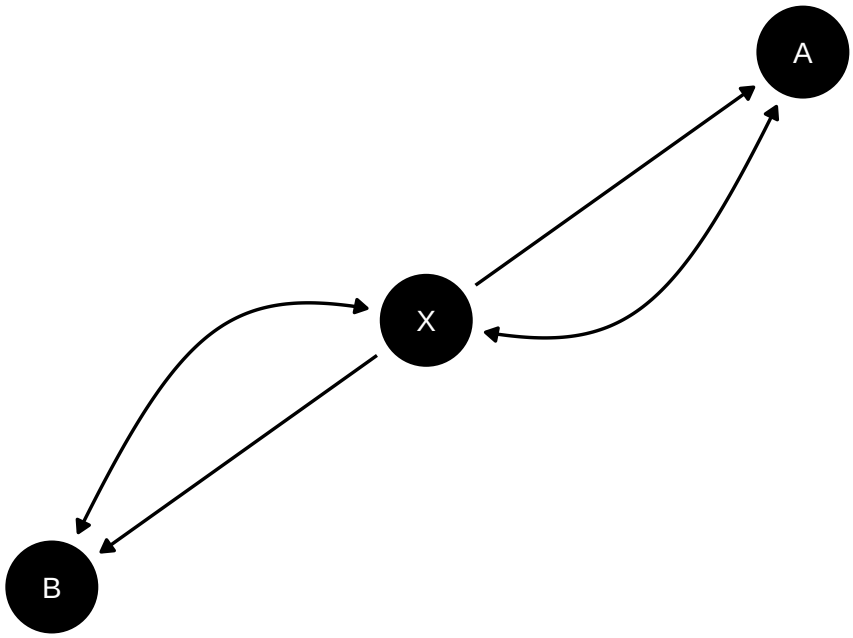
- Unlike nodal restrictions, a confounding relation can involve multiple nodal types simultaneously. For instance `make_model("X -> M -> Y") %>% set_confound(list(X = "Y[X=1] > Y[X=0]"))` allows for a parameter that renders X more or less likely depending on whether X has a positive effect on Y whether it runs through a positive or a negative effect on M.

- The parameters needed to capture confounding relations depend on the direction of causal arrows. For example compare:

  - `make_model("A -> W <- B ; A <-> W; B <-> W")$parameters_df %>% dim` In this case we can decompose shocks on A, B, W via: $\Pr(\theta^A, \theta^B, \theta^W) = \Pr(\theta^W|\theta^A, \theta^A)\Pr(\theta^A)\Pr(\theta^B)$, and we have 68 parameters.
  - `make_model("A <- W -> B ; A <-> W; B <-> W")$parameters_df %>% dim` In this case we have $\Pr(\theta^A, \theta^B, \theta^W) = \Pr(\theta^A|\theta^W)\Pr(\theta^B|\theta^W)\Pr(\theta^W)$ and just has just 18 parameters.

When confounding is added to a model, a dataframe, `confounds_df` is created and added to the model, recording which variables involve confounding. This is then used for plotting:

```
make_model("A <- X -> B; A <-> X; B <-> X") %>% plot()
```



3.9. Setting Priors

Priors on model parameters can be added to the parameters dataframe. The priors are interpreted as alpha arguments for a Dirichlet distribution. They can be seen using `get_priors`.

```
make_model("X->Y") %>% get_priors
```

```
 X.0  X.1 Y.00 Y.10 Y.01 Y.11
   1    1    1    1    1    1
```

Here the priors have not been specified and so they default to 1, which corresponds to uniform priors.

Alternatively you could set jeffreys priors like this:

```
make_model("X->Y") %>% set_priors(distribution = "jeffreys") %>% get_priors
```

```
no specific parameters to alter values for specified. Altering all parameters.
```

```
 X.0  X.1 Y.00 Y.10 Y.01 Y.11
 0.5  0.5  0.5  0.5  0.5  0.5
```

Custom priors

Custom priors are most simply specified by being added as a vector of numbers using `set_priors`. For instance:

```
make_model("X->Y") %>% set_priors(1:6) %>% get_priors
```

```
 X.0  X.1 Y.00 Y.10 Y.01 Y.11
   1    2    3    4    5    6
```

The priors here should be interpreted as indicating:

- $\alpha_X = (1, 2)$, which implies a distribution over $(\lambda_0^X, \lambda_1^X)$ centered on $(1/3, 2/3)$.
- $\alpha_Y = (3, 4, 5, 6)$, which implies a distribution over $(\lambda_{00}^Y, \lambda_{10}^Y, \lambda_{01}^Y \lambda_{11}^Y)$ centered on $(3/18, 4/18, 5/18, 6/18)$.

For larger models it can be hard to provide priors as a vector of numbers and so `set_priors` can allow for more targeted modifications of the parameter vector. For instance:

```
make_model("X->Y") %>%
  set_priors(statement = "Y[X=1] > Y[X=0]", alphas = 3) %>%
  get_priors
```

```
 X.0  X.1 Y.00 Y.10 Y.01 Y.11
   1    1    1    1    3    1
```

See `?set_priors` and `?make_priors` for many examples.

Prior warnings

"Flat" priors over parameters in a parameter family put equal weight on each nodal type, but this in turn can translate into strong assumptions on causal quantities of interest.

For instance in an $X \rightarrow Y$ model model in which negative effects are ruled out, the average causal effect implied by "flat" priors is 1/3. This can be seen by querying the model:

```
make_model("X -> Y") %>%
  set_restrictions(decreasing("X", "Y")) %>%
  query_model("Y[X=1] - Y[X=0]", n_draws = 10000) %>%
  kable
```

model	query	given	using	case_level	mean	sd	cred.low.2.5%	cred.high.97.5%
model_1	Y[X=1] - Y[X=0]	-	parameters	FALSE	0.3333333	NA	0.3333333	0.3333333

More subtly the *structure* of a model, coupled with flat priors, has substantive importance for priors on causal quantities. For instance with flat priors, priors on the probability that X has a positive effect on Y in the model $X \rightarrow Y$ is centered on 1/4. But priors on the probability that X has a positive effect on Y in the model $X \rightarrow M \rightarrow Y$ is centered on 1/8.

Again, you can use `query_model` to figure out what flat (or other) priors over parameters imply for priors over causal quantities:

```
make_model("X -> Y") %>%
  query_model("Y[X=1] > Y[X=0]", n_draws = 10000) %>%
  kable
```

model	query	given	using	case_level	mean	sd	cred.low.2.5%	cred.high.97.5%
model_1	Y[X=1] > Y[X=0]	-	parameters	FALSE	0.25	NA	0.25	0.25

```
make_model("X -> M -> Y") %>%
  query_model("Y[X=1] > Y[X=0]", n_draws = 10000) %>%
  kable
```

model	query	given	using	case_level	mean	sd	cred.low.2.5%	cred.high.97.5%
model_1	Y[X=1] > Y[X=0]	-	parameters	FALSE	0.125	NA	0.125	0.125

Caution regarding priors is particularly important when models are not identified, as is the case for many of the models considered here. In such cases, for some quantities, the marginal posterior distribution can be the same as the marginal prior distribution (**?**).

The key point here is to make sure you do not fall into a trap of thinking that "uninformative" priors make no commitments regarding the values of causal quantities of interest. They do, and the implications of flat priors for causal quantities can depend on the structure of the model. Moreover for some inferences from causal models the priors can matter a lot even if you have a lot of data. In such cases it can be helpful to know what priors on parameters imply for priors on causal quantities of interest (by using `query_model`) and to assess how

much conclusions depend on priors (by comparing results across models that vary in their priors).

3.10. Setting Parameters

By default, models have a vector of parameter values included in the `parameters_df` dataframe. These are useful for generating data, or for situations, such as process tracing, when one wants to make inferences about causal types (θ), given case level data, under the assumption that the model is known.

Consider the causal model below. It has two parameter sets, X and Y, with six nodal types, two corresponding to X and four corresponding to Y. The key feature of the parameters is that they must sum to 1 within each parameter set.

```
make_model("X->Y") |>
  get_parameters()
```

```
 X.0  X.1 Y.00 Y.10 Y.01 Y.11
0.50 0.50 0.25 0.25 0.25 0.25
```

Setting parameters can be done using a similar syntax as `set_priors`. The main difference is that when a given value is altered the entire set must still always sum to 1. The example below illustrates a change in the value of the parameter Y in the case it is increasing in X. Here nodal type `Y.Y01` is set to be .5, while the other nodal types of this parameter set were renormalized so that the parameters in the set still sum to one.

```
make_model("X->Y") %>%
  set_parameters(statement = "Y[X=1] > Y[X=0]", parameters = .5) %>%
  get_parameters
```

```
      X.0       X.1      Y.00      Y.10      Y.01      Y.11
0.5000000 0.5000000 0.1666667 0.1666667 0.5000000 0.1666667
```

3.11. Using models

Drawing data

```
model |> make_data(n = 5) |> kable()
```

X	Y
0	0
0	1
0	1
1	1
1	1

4. Updating models

The approach used by the `CausalQueries` package to updating parameter values given observed data uses `stan` and involves the following elements:

- Dirichlet priors over parameters, λ (which, in cases without confounding, correspond to nodal types)
- A mapping from parameters to event probabilities, w
- A likelihood function that assumes events are distributed according to a multinomial distribution given event probabilities.

We provide further details below.

4.1. Data for stan

We use a generic `stan` model that works for all binary causal models. Rather than writing a new `stan` model for each causal model we send `stan` details of each particular causal model as data inputs.

In particular we provide a set of matrices that `stan` tailor itself to particular models: the parameter matrix (P) tells `stan` how many parameters there are, and how they map into causal types; an ambiguity matrix A tells `stan` how causal types map into data types; and an event matrix E relates data types into patterns of observed data (in cases where there are incomplete observations).

The internal function `prep_stan_data` prepares data for `stan`. You generally don't need to use this manually, but we show here a sample of what it produces as input for `stan`.

We provide `prep_stan_data` with data in compact form (listing "data events").

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

data  <- data.frame(X = c(0, 1, 1, NA), Y = c(0, 1, 0, 1))

compact_data <-  collapse_data(data, model)

kable(compact_data)
```

event	strategy	count
X0Y0	XY	1
X1Y0	XY	1
X0Y1	XY	0
X1Y1	XY	1
Y0	Y	0
Y1	Y	1

Note that NAs are interpreted as data not having been sought. So in this case the interpretation is that there are two data strategies: data on Y and X was sought in three cases; data on Y only was sought in just one case.

`prep_stan_data` then returns a list of objects that `stan` expects to receive. These include indicators to figure out where a parameter set starts (`l_starts`, `l_ends`) and ends and where a data strategy starts and ends (`strategy_starts`, `strategy_ends`), as well as the matrices described above.

4.2. stan code

Below we show the `stan` code. This starts off with a block saying what input data is to be expected. Then there is a characterization of parameters and the transformed parameters. Then the likelihoods and priors are provided. `stan` takes it from there and generates a posterior distribution.

```
functions{
  row_vector col_sums(matrix X) {
    row_vector[cols(X)] s ;
    s = rep_row_vector(1, rows(X)) * X ;
    return s ;
  }
}
data {
int<lower=1> n_params;
int<lower=1> n_paths;
int<lower=1> n_types;
int<lower=1> n_param_sets;
int<lower=1> n_nodes;
int<lower=1> n_param_each[n_param_sets];
int<lower=1> n_data;
int<lower=1> n_events;
int<lower=1> n_strategies;
int<lower=0, upper=1> keep_transformed;
vector<lower=0>[n_params] lambdas_prior;
int<lower=1> l_starts[n_param_sets];
int<lower=1> l_ends[n_param_sets];
int<lower=1> node_starts[n_nodes];
int<lower=1> node_ends[n_nodes];
int<lower=1> strategy_starts[n_strategies];
int<lower=1> strategy_ends[n_strategies];
matrix[n_params, n_types] P;
matrix[n_params, n_paths] parmap;
matrix[n_paths, n_data] map;
matrix<lower=0,upper=1>[n_events,n_data] E;
int<lower=0> Y[n_events];
}
parameters {
vector<lower=0>[n_params - n_param_sets] gamma;
}
transformed parameters {
```

```
vector<lower=0>[n_params] lambdas;
vector<lower=1>[n_param_sets] sum_gammas;
matrix[n_params, n_paths] parlam;
matrix[n_nodes, n_paths] parlam2;
vector<lower=0, upper=1>[n_paths] w_0;
vector<lower=0, upper=1>[n_data] w;
vector[n_events] w_full;
// Cases in which a parameter set has only one value need special handling
// they have no gamma components and sum_gamma needs to be made manually
for (i in 1:n_param_sets) {
if (l_starts[i] >= l_ends[i]) {
  sum_gammas[i] = 1;
  // syntax here to return unity as a vector
  lambdas[l_starts[i]] = lambdas_prior[1]/lambdas_prior[1];
  }
else if (l_starts[i] < l_ends[i]) {
  sum_gammas[i] =
  1 + sum(gamma[(l_starts[i] - (i-1)):(l_ends[i] - i)]);
  lambdas[l_starts[i]:l_ends[i]] =
  append_row(1, gamma[(l_starts[i] - (i-1)):(l_ends[i] - i)]) / sum_gammas[i];
  }
  }
// Mapping from parameters to data types
parlam  = rep_matrix(lambdas, n_paths) .* parmap; // (usual case): [n_par * n_data] * [n_p
// Sum probability over nodes on each path
for (i in 1:n_nodes) {
 parlam2[i,] = col_sums(parlam[(node_starts[i]):(node_ends[i]),]);
 }
// then take product  to get probability of data type on path
for (i in 1:n_paths) {
  w_0[i] = prod(parlam2[,i]);
 }
 // last (if confounding): map to n_data columns instead of n_paths
 w = map'*w_0;
 w = w / sum(w);
 w_full = E * w;
}
model {
// Dirichlet distributions (earlier versions used gamma)
for (i in 1:n_param_sets) {
  target += dirichlet_lpdf(lambdas[l_starts[i]:l_ends[i]]  | lambdas_prior[l_starts[i] :l_
  target += -n_param_each[i] * log(sum_gammas[i]);
 }
// Multinomials
for (i in 1:n_strategies) {
  target += multinomial_lpmf(
  Y[strategy_starts[i]:strategy_ends[i]] | w_full[strategy_starts[i]:strategy_ends[i]]);
```

```
 }
}
// Option to export distribution of causal types
// Note if clause used here to effectively turn off this block if not required
generated quantities{
vector[n_types] prob_of_types;
if (keep_transformed == 1){
for (i in 1:n_types) {
   prob_of_types[i] = prod(P[, i].*lambdas + 1 - P[,i]);
}}
 if (keep_transformed == 0){
    prob_of_types = rep_vector(1, n_types);
 }
}
```

The `stan` model works as follows:

- We are interested in "sets" of parameters. For example in the $X \rightarrow Y$ model we have two parameter sets (`param_sets`). The first is $\lambda^X \in \{\lambda^X_0, \lambda^X_1\}$ whose elements give the probability that X is 0 or 1. These two probabilities sum to one. The second parameter set is $\lambda^Y \in \{\lambda^Y_{00}, \lambda^Y_{10}, \lambda^Y_{01}\lambda^Y_{11}\}$. These are also probabilities and their values sum to one. Note in all that we have 6 parameters but just $1 + 3 = 4$ degrees of freedom.

- We would like to express priors over these parameters using multiple Dirichlet distributions (two in this case). In practice because we are dealing with multiple simplices of varying length, it is easier to express priors over gamma distributions with a unit scale parameter and shape parameter corresponding to the Dirichlet priors, α. We make use of the fact that $\lambda^X_0 \sim Gamma(\alpha^X_0, 1)$ and $\lambda^X_1 \sim Gamma(\alpha^X_1, 1)$ then $\frac{1}{\lambda^X_0+\lambda^X_1}(\lambda^X_0, \lambda^X_1) \sim Dirichlet(\alpha^X_0, \alpha^X_1)$. For a discussion of implementation of this approach in `stan` see https://discourse.mc-stan.org/t/ragged-array-of-simplexes/1382.

- For any candidate parameter vector λ we calculate the probability of *causal* types (`prob_of_types`) by taking, for each type i, the product of the probabilities of all parameters (λ_j) that appear in column i of the parameter matrix P. Thus the probability of a (X_0, Y_{00}) case is just $\lambda^X_0 \times \lambda^Y_{00}$. The implementations in `stan` uses `prob_of_types_[i]` $= \prod_j \left(P_{j,i}\lambda_j + (1 - P_{j,i})\right)$: this multiplies the probability of all parameters involved in the causal type (and substitutes 1s for parameters that are not). (`P` and `not_P` (1-P) are provided as data to `stan`).

- The probability of data types, `w`, is given by summing up the probabilities of all causal types that produce a given data type. For example, the probability of a $X = 0, Y = 0$ case, w_{00} is $\lambda^X_0 \times \lambda^Y_{00} + \lambda^X_0 \times \lambda^Y_{01}$. The ambiguity matrix A is provided to `stan` to indicate which probabilities need to be summed.

- In the case of incomplete data we first identify the set of "data strategies", where a collection of a data strategy might be of the form "gather data on X and M, but not Y, for n_1 cases and gather data on X and Y, but not M, for n_2 cases. The probability of an observed event, within a data strategy, is given by summing the probabilities of the types that could give rise to the incomplete data. For example X is observed, but

Y is not, then the probability of $X = 0, Y = \text{NA}$ is $w_{00} + w_{01}$. The matrix E is passed to `stan` to figure out which event probabilities need to be combined for events with missing data.

- The probability of a dataset is then given by a multinomial distribution with these event probabilities (or, in the case of incomplete data, the product of multinomials, one for each data strategy). Justification for this approach relies on the likelihood principle and is discussed in Chapter 6.

4.3. Implementation

To update a CausalQueries model with data use:

```
update_model(model, data)
```

where the data argument is a dataset containing some or all of the nodes in the model.

As `update_model()` calls `rstan::sampling` one can pass along all arguments in `...` to `rstan::sampling`. For instance:

- `iter` sets the number of iterations and ultimately the number of draws in the posterior
- `chains` sets the number of chains; doing multiple chains in parallel speeds things up

If you have multiple cores you can do parallel processing by including this line before running `CausalQueries`:

```
options(mc.cores = parallel::detectCores())
```

Note the parameters estimated by `stan` include the gamma parameters plus transformed parameters, λ, which are our parameters of interest and which `CausalQueries` then interprets as possible row probabilities for the P matrix.

4.4. censored data

4.5. Output

The primary output from `update_model()` is a posterior distribution over model parameters, stored as a dataframe in `model$posterior_distribution`. However another of other objects are also optionally stored:

5. Querying models

Models can be queried using the `query_distribution` and `query_model` functions. The difference between these functions is that `query_distribution` examines a single query and returns a full distribution of draws from the distribution of the estimand (prior or posterior); `query_model` takes a collection of queries and returns a dataframe with summary statistics on the queries.

The simplest queries ask about causal estimands given particular parameter values and case level data. Here is one surprising result of this form:

5.1. Case level queries

The `query_model` function takes causal queries and conditions (`given`) and specifies the parameters to be used. The result is a dataframe which can be displayed as a table.

For a case level query we can make the query *given* a particular parameter vector, as below:

```
make_model("X-> M -> Y <- X") %>%

  set_restrictions(c(decreasing("X", "M"),
                     decreasing("M", "Y"),
                     decreasing("X", "Y"))) %>%

  query_model(queries = "Y[X=1]> Y[X=0]",
              given = c("X==1 & Y==1",
                        "X==1 & Y==1 & M==1",
                        "X==1 & Y==1 & M==0"),
              using = c("parameters")) %>%

  kable(
    caption = "In a monotonic model with flat priors, knowledge
    that $M=1$ *reduces* confidence that $X=1$ caused $Y=1$")
```

model	query	given	using	case_level	mean	sd	cred.low.2.5%	cred.high.97.5%
model_1	Y[X=1]> Y[X=0]	X==1 & Y==1	parameters	FALSE	0.6153846	NA	0.6153846	0.6153846
model_1	Y[X=1]> Y[X=0]	X==1 & Y==1 & M==1	parameters	FALSE	0.6000000	NA	0.6000000	0.6000000
model_1	Y[X=1]> Y[X=0]	X==1 & Y==1 & M==0	parameters	FALSE	0.6666667	NA	0.6666667	0.6666667

Table 11: In a monotonic model with flat priors, knowledge that $M = 1$ *reduces* confidence that $X = 1$ caused $Y = 1$

This example shows how inferences change given additional data on M in a monotonic $X \rightarrow M \rightarrow Y \leftarrow X$ model. Surprisingly observing $M = 1$ *reduces* beliefs that X caused Y, the reason being that perhaps M and not X was responsible for $Y = 1$.

5.2. Posterior queries

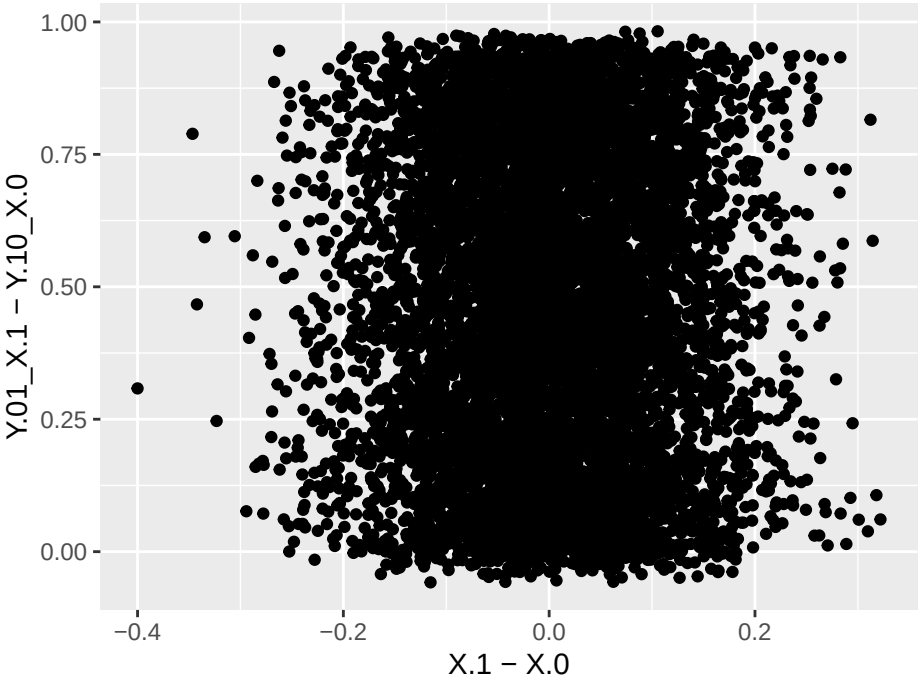
Queries can also draw directly from the posterior distribution provided by `stan`. In this next example we illustrate the joint distribution of the posterior over causal effects, drawing directly from the posterior dataframe generated by `update_model`:

```
library(DeclareDesign)

data  <- fabricate(N = 100, X = complete_ra(N), Y = X)

model <- make_model("X -> Y; X <-> Y") %>%
  update_model(data, iter  = 4000)

model$posterior_distribution %>%
  data.frame() %>%
  ggplot(aes(X.1 - X.0, Y.01_X.1 - Y.10_X.0)) +
  geom_point()
```



We see that beliefs about the size of the overall effect are related to beliefs that X is assigned differently when there is a positive effect.

5.3. Query distribution

`query_distribution` works similarly except that the query is over an estimand. For instance:

```
make_model("X -> Y") %>%
  query_distribution(
    query = list(increasing = "(Y[X=1] > Y[X=0])"),
    using = "priors") |>
```

```
  ggplot(aes(increasing)) +
  geom_histogram()
```

```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

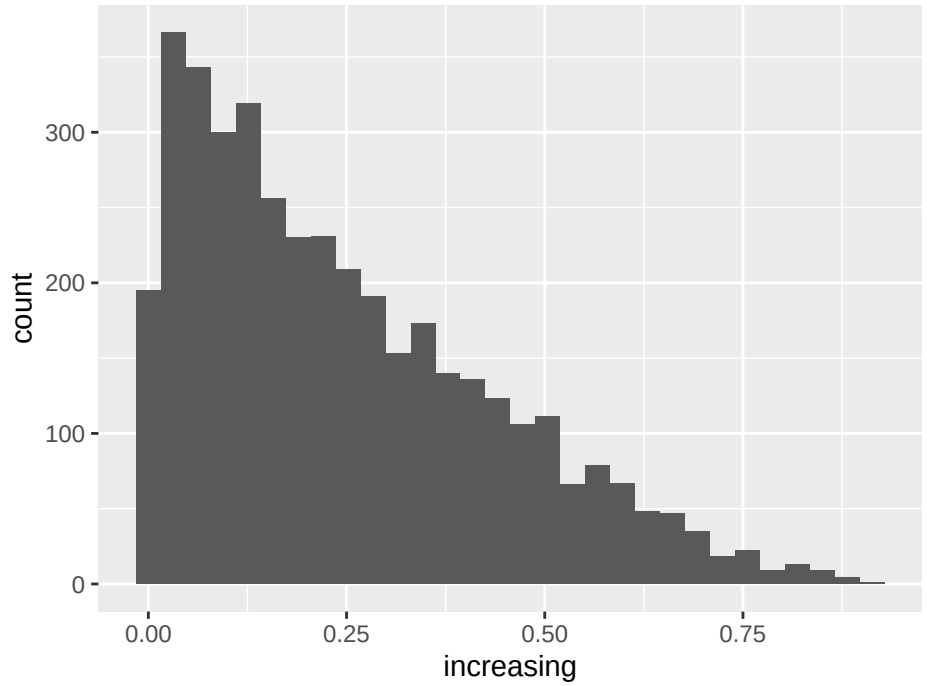


Figure 1: Prior on 'Probability Y is increasing in X'

5.4. Token and general causation

Note that in all these cases we use the same technology to make case level and population inferences. Indeed the case level query is just a conditional population query. As an illustration of this imagine we have a model of the form $X \rightarrow M \rightarrow Y$ and are interested in whether X caused Y in a case in which $M = 1$. We answer the question by asking "what would be the probability that X caused Y in a case in which $X = M = Y = 1$?" (line 3 below). This speculative answer is the same answer as we would get were we to ask the same question having updated our model with knowledge that in a particular case, indeed, $X = M = Y = 1$. See below:

```
model <- make_model("X->M->Y") %>%
  set_restrictions(c(decreasing("X", "M"), decreasing("M", "Y"))) %>%
  update_model(data = data.frame(X = 1, M = 1, Y = 1), iter = 8000)

query_model(
            model,
            query = "Y[X=1]> Y[X=0]",
            given = c("X==1 & Y==1", "X==1 & Y==1 & M==1"),
```

```
using = c("priors", "posteriors"),
expand_grid = TRUE)
```

model	query	given	using	case_level	mean	sd	cred.low.2.5%	cred.high.97.5%
model_1	Y[X=1]> Y[X=0]	X==1 & Y==1	priors	FALSE	0.21	0.21	0	0.75
model_1	Y[X=1]> Y[X=0]	X==1 & Y==1	posteriors	FALSE	0.22	0.21	0	0.76
model_1	Y[X=1]> Y[X=0]	X==1 & Y==1 & M==1	priors	FALSE	0.25	0.22	0	0.78
model_1	Y[X=1]> Y[X=0]	X==1 & Y==1 & M==1	posteriors	FALSE	0.25	0.22	0	0.78

Table 12: Posteriors equal priors for a query that conditions on data used to form the posterior

We see the conditional inference is the same using the prior and the posterior distributions.

6. Illustrations

6.1. Identification with CausalQueries

Computational details

- information about certain computational details such as version numbers, operating systems, or compilers could be included in an unnumbered section. Also, auxiliary packages (say, for visualizations, maps, tables, …) that are not cited in the main text can be credited here.

:::

The results in this paper were obtained using R~3.4.1 with the **MASS**~7.3.47 package. R itself and all packages used are available from the Comprehensive R Archive Network (CRAN) at [https://CRAN.R-project.org/].

Acknowledgments

All acknowledgments (note the AE spelling) should be collected in this unnumbered section before the references. It may contain the usual information about funding and feedback from colleagues/reviewers/etc. Furthermore, information such as relative contributions of the authors may be added here (if any).

References

More technical details

Appendices can be included after the bibliography (with a page break). Each section within the appendix should have a proper section title (rather than just *Appendix*).
For more technical style details, please check out JSS's style FAQ at [https://www.jstatsoft.org/pages/view/style#frequently-asked-questions] which includes the following topics:

- Title vs. sentence case.
- Graphics formatting.
- Naming conventions.
- Turning JSS manuscripts into R package vignettes.
- Trouble shooting.
- Many other potentially helpful details…

Using BibTeX

References need to be provided in a BibTeX file (`.bib`). All references should be made with `@cite` syntax. This commands yield different formats of author-year citations and allow to include additional details (e.g.,pages, chapters, …) in brackets. In case you are not familiar with these commands see the JSS style FAQ for details.
Cleaning up BibTeX files is a somewhat tedious task – especially when acquiring the entries automatically from mixed online sources. However, it is important that informations are complete and presented in a consistent style to avoid confusions. JSS requires the following format.

- item JSS-specific markup (`\proglang`, `\pkg`, `\code`) should be used in the references.
- item Titles should be in title case.
- item Journal titles should not be abbreviated and in title case.
- item DOIs should be included where available.
- item Software should be properly cited as well. For R packages `citation("pkgname")` typically provides a good starting point.

Affiliation:

Till Tietz
IPI
Reichpietschufer 50
Berlin Germany
E-mail: ttietz2014@gmail.com

Lily Medina

Macartan Humphreys
E-mail: macartan.humphreys@wzb.eu
URL: https://macartan.github.io/

Journal of Statistical Software
published by the Foundation for Open Access Statistics
MMMMMM YYYY, Volume VV, Issue II
doi:10.18637/jss.v000.i00

http://www.jstatsoft.org/
http://www.foastat.org/
Submitted: yyyy-mm-dd
Accepted: yyyy-mm-dd