Introduction to the Stan Language

Ben Goodrich

May 11, 2020

Grading / Final Projects

- I have the graded Assignment 1s and Assignment 2 should be graded this afternoon
- Final Projects due by 11:59 PM on May 19th
- Can analyze data used in another class
- If you cannot share the data, let me know
- Can use rstanarm or brms or write your own Stan code
- I don’t care very much what the previous literature says
- Go through the process of laying out a generative model, drawing from the prior predictive distribution, conditioning on the observed data (and making sure Stan samples well), looking at posterior predictive plots, comparing it to an alternative model, etc.
- Should be around ten pages as a PDF

Workflow for Stan via R

- You write the program in a (text) .stan file with a R-like syntax
- Stan's parser, `stanc`, does three things:
  - checks that program is syntactically valid and tells you if not
  - writes a conceptually equivalent C++ source file to disk
  - C++ compiler creates a binary file from the C++ source
- When you have some C++ like `x = mu + sigma * z;`
  - C++ can automatically store $\frac{\partial x}{\partial \mu}$, $\frac{\partial x}{\partial \sigma}$, and $\frac{\partial x}{\partial z}$ by overloading arithmetic operators and handle the chain-rule for you
  - Called automatic differentiation (not numerical differentiation)
  - Unless μ, σ, or z is constant, in which case it doesn't bother
- You execute the binary from R (can be concurrent with parsing and compiling)
- You analyze the resulting samples from the posterior

Primitive Object Types in Stan

- In Stan / C++, variables must declared with types
- In Stan / C++, statements are terminated with semi-colons
- Primitive scalar types: `real x;` or `int K;`
  - Unknowns cannot be `int` because no derivatives and hence no HMC
  - Can condition on integer data because no derivatives are needed
- Implicitly real `vector[K] z;` or `row_vector[K] z;`
- Implicitly real `matrix[N,K] X;` can have 1 column / row
- Arrays are just holders of any other homogenous objects
  - `real x[N]` is similar to `vector[N] x;` but lacks linear algebra functions
  - `vector[N] X[K];` and `row_vector[K] X[N]` are similar to `matrix[N,K] X;` but lack linear algebra functionality, although they have uses in loops
- Vectors and matrices cannot store integers, so instead use possibly multidimensional integer arrays `int y[N];` or `int Y[N,P];`

The `lookup` Function in rstan

- Can input the name of an R function, in which case it will try to find an analagous Stan function
- Can input a regular expression, in which case it will find matching Stan functions that match

```
library(rstan)          # functions starting with inv
lookup("^inv.*[^gf]$") # but not ending with g or f
```

```
#       StanFunction  Arguments ReturnType
# 216 inv_chi_square          ~       real
# 219        inverse (matrix A)     matrix
# 220    inverse_spd (matrix A)     matrix
# 225      inv_gamma          ~       real
# 227      inv_logit      (T x)          R
# 228        inv_phi      (T x)          R
# 229       inv_sqrt      (T x)          R
# 230     inv_square      (T x)          R
# 233    inv_wishart          ~       real
```

Optional `functions` Block of .stan Programs

- Stan permits users to define and use their own functions
- If used, must be defined in a leading `functions` block
- Can only validate constraints inside user-defined functions
- Very useful for several reasons:
    - Easier to reuse across different .stan programs
    - Makes subsequent chunks of code more readable
    - Enables posteriors with Ordinary Differential Equations, algebraic equations, and integrals
    - Can be exported to R via `expose_stan_functions()`
- All functions, whether user-defined or build-in, must be called by argument position rather than by argument name, and there are no default arguments
- User-defined functions cannot have the same name as existing functions or keywords and are case-sensitive

Constrained Object Declarations in Stan

Outside of the `functions` block, any primitive object can have bounds:

- `int<lower = 1> K;` `real<lower = -1, upper = 1> rho;`
- `vector<lower = 0>[K] alpha;` and similarly for a `matrix`
- A `vector` (but not a `row_vector`) can be further specialized:
  - `unit_vector[K] x;` implies $\sum_{k=1}^{K} x_k^2 = 1$
  - `simplex[K] x;` implies $x_k \geq 0 \, \forall k$ and $\sum_{k=1}^{K} x_k = 1$
  - `ordered[K] x;` implies $x_j < x_k \, \forall j < k$
  - `positive_ordered[K] x;` implies $0 < x_j < x_k \, \forall j < k$
- A `matrix` can be specialized to enforce constraints:
  - `cov_matrix[K] Sigma;` or better `cholesky_factor_cov[K, K] L;`
  - `corr_matrix[K] Lambda;` or `cholesky_factor_corr[K] C;`

“Required” **data** Block of .stan Programs

- All knowns passed from R to Stan as a NAMED list, such as outcomes $(\mathbf{y})$, covariates $(\mathbf{X})$, constants (K), and / or hyperparameters $(\mathbf{a})$
- Basically, everything posterior distribution conditions on
- Can have comments in C++ style (`//` or `/* ... */`)
- Whitespace is essentially irrelevant, except after keywords

```
data {
  int<lower = 0> N;     // number of observations
  int<lower = 0> y[N];  // count outcome

  real<lower = 0> a;    // shape of gamma prior
  real<lower = 0> b;    // rate of gamma prior
}
```

“Required” **`parameters`** Block of .stan Programs

- Declare exogenous unknowns whose posterior distribution is sought
- Cannot declare any integer parameters currently, only reals
- Must specify the parameter space but lower and upper bounds are implicitly $\pm\infty$ if unspecified

```
parameters {
  real<lower = 0> mu;   // mean of DGP
}
```

- The change-of-variables adjustment due to the transformation from an unconstrained parameter space to the (in this case, positive) constrained space is handled automatically and added to `target`

“Required” `model` Block of .stan Programs

- Can declare endogenous unknowns, assign to them, and use them
- Constraints cannot be declared / validated and samples not stored
- The `model` block must define (something proportional to) $\text{target} = \log(f(\boldsymbol{\theta}) \times f(\mathbf{y}|\,\boldsymbol{\theta}, \cdot)) = \log f(\boldsymbol{\theta}) + \log f(\mathbf{y}|\,\boldsymbol{\theta}, \cdot)$
- There is an internal reserved symbol called `target` that is initialized to zero (before change-of-variable adjustments) you increment by `target += ...;`
- Functions ending `_lpdf` or `_lpmf` return scalars even if some of their arguments are vectors or one-dimensional arrays, in which case it sums the log density/mass over the presumed conditionally independent elements

```
model {
  target += gamma_lpdf(mu | a, b); // log prior PDF
  target += poisson_lpmf(y | mu);  // log likelihood
}
```

Entire Stan Program

```
data {
  int<lower = 0> N;     // number of observations
  int<lower = 0> y[N];  // count outcome

  real<lower = 0> a;    // shape of gamma prior
  real<lower = 0> b;    // rate of gamma prior
}
parameters {
  real<lower = 0> mu;   // mean of DGP
}
model {
  target += gamma_lpdf(mu | a, b); // log prior PDF
  target += poisson_lpmf(y | mu);  // log likelihood
}
```

Calling **stan** in the rstan Package

```
library(rstan)
options(mc.cores = parallel::detectCores())
fit_1 <- stan("poisson.stan",
              data = list(N = nrow(faithful), y = faithful$waiting,
                          a = 2, b = 0.03),
              # below are default values but you could change them
              control = list(adapt_delta = 0.8, max_treedepth = 10))
```

```
dim(fit_1) # 1000 draws on each of 4 chains for 1 unknown and 1 target
```

```
## [1] 1000    4    2
```

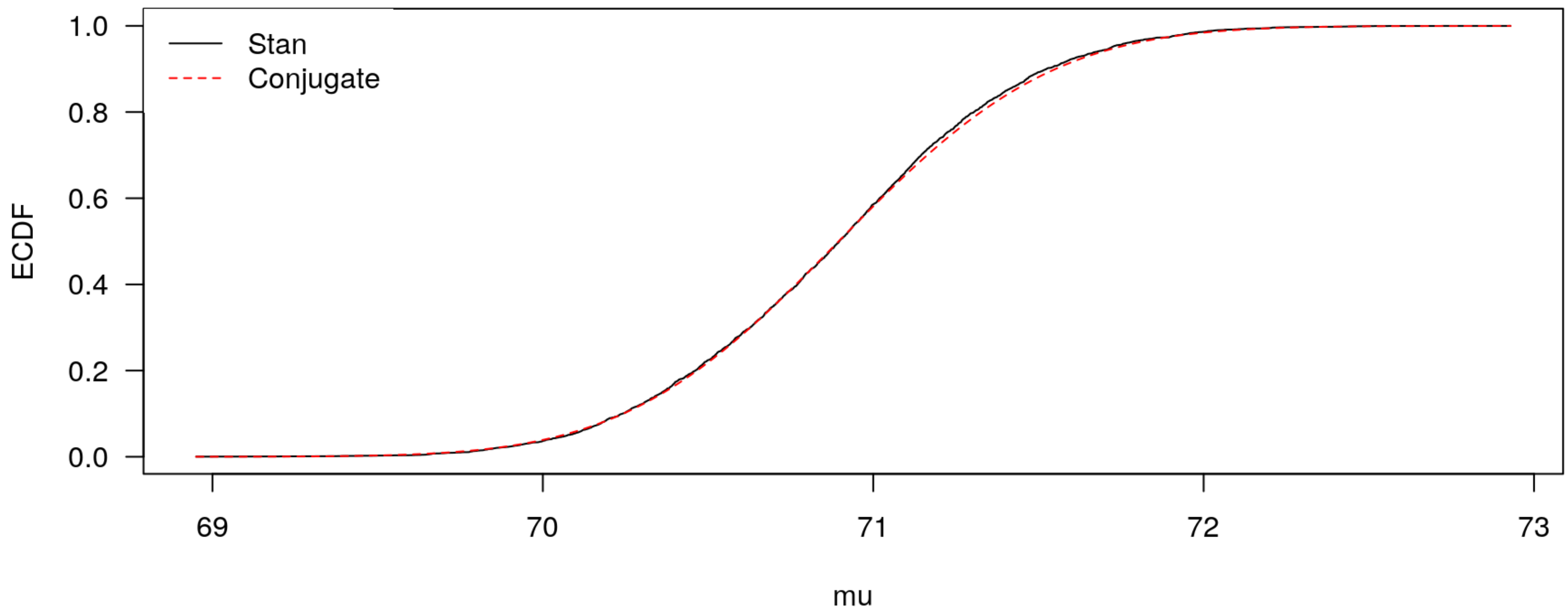
Posterior Summary

```
print(fit_1, digits = 2)
```

```
## Inference for Stan model: poisson.
## 4 chains, each with iter=2000; warmup=1000; thin=1;
## post-warmup draws per chain=1000, total post-warmup draws=4000.
##
##          mean se_mean   sd     2.5%      25%      50%      75%    97.5% n_eff Rhat
## mu      70.89    0.01 0.50    69.91    70.54    70.90    71.22    71.90  1465    1
## lp__ -1195.34    0.02 0.69 -1197.22 -1195.49 -1195.07 -1194.91 -1194.86  1905    1
##
## Samples were drawn using NUTS(diag_e) at Mon May 11 08:34:55 2020.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

Extracting Posterior Draws

```r
mu <- sort(as.data.frame(fit_1)$mu) # or use extract()
plot(mu, (1:length(mu)) / length(mu), type = "l", ylab = "ECDF")
lines(mu, pgamma(mu, shape = 2 + sum(faithful$waiting),
                 rate = 0.03 + length(faithful$waiting)), col = 2, lty = 2)
legend("topleft", legend = c("Stan", "Conjugate"), col = 1:2, lty = 1:2, box.lwd = NA)
```



Optional `transformed parameters` Block

- Comes after the `parameters` block but before the `model` block
- Need to declare objects before they are assigned
- Calculate endogenous unknowns that are deterministic functions of things declared in earlier blocks
- Used to create interesting intermediate inputs to the log-kernel
- Declared constraints are validated and samples are stored
- Often used in multilevel models to define group-specific unknowns

Stan Does not Care about Conjugacy

```
#include quantile_functions.stan
data {
  int<lower = 0> N;     // number of observations
  int<lower = 0> y[N];  // count outcome

  real<lower = 0> m;    // prior median
  real<lower = 0> r;    // prior IQR
  real<lower = -1, upper = 1> asymmetry;
  real<lower =  0, upper = 1> steepness;
}
parameters {
  real<lower = 0, upper = 1> p;
}
transformed parameters {
  real mu = GLD_icdf(p, m, r, asymmetry, steepness);
} // implicit: p has a standard uniform prior
model {
  target += poisson_lpmf(y | mu);           // log likelihood
}
```

Posterior from GLD Prior

```r
expose_stan_functions("quantile_functions.stan")
source("GLD_helpers.R")
a_s <- GLD_solver_LBFGS(lower_quartile = 25, median = 60, upper_quartile = 125,
                        other_quantile = 0, alpha = 0)
fit_2 <- stan("poisson_GLD.stan",
              data = list(N = nrow(faithful), y = faithful$waiting, m = 60,
                          r = 125 - 25, asymmetry = a_s[1], steepness = a_s[2]))
```

```r
fit_2
```

```
## Inference for Stan model: poisson_GLD.
## 4 chains, each with iter=2000; warmup=1000; thin=1;
## post-warmup draws per chain=1000, total post-warmup draws=4000.
##
##          mean se_mean   sd     2.5%      25%      50%      75%    97.5% n_eff Rhat
## p        0.56    0.00 0.00     0.55     0.56     0.56     0.56     0.56  1652    1
## mu      70.91    0.01 0.51    69.89    70.55    70.92    71.26    71.89  1652    1
## lp__ -1196.14    0.02 0.71 -1198.10 -1196.29 -1195.88 -1195.69 -1195.64  1910    1
##
## Samples were drawn using NUTS(diag_e) at Mon May 11 01:45:10 2020.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

Breakout Rooms:

- Let $\mu = e^{\eta}$ in the Poisson log-PMF:

$$y_n \log \mu - \mu + \sum_{k=2}^{y_n} \log k$$

- Write a Stan program with a normal prior on η

Optional `generated quantities` Block

- Can declare more endogenous knowns, assign to them, and use them
- Samples are stored
- Can reference anything except stuff in the `model` block
- Can also do this in R afterward, but primarily used for
    - Interesting functions of posterior that don't involve likelihood
    - Posterior predictive distributions and / or functions thereof
    - The log-likelihood for each observation to pass to `loo`

Reparameterizing the Likelihood

```
data {
  int<lower = 0> N;     // number of observations
  int<lower = 0> y[N];  // count outcome

  real<lower = 0> loc;  // location of normal prior
  real<lower = 0> scal; // scale of normal prior
}
parameters {
  real eta; // log of mean of DGP
}
model {
  target += normal_lpdf(eta | loc, scal); // log prior PDF
  target += poisson_log_lpmf(y | eta);    // log likelihood
}
generated quantities {
  real mu = exp(eta);
}
```

Posterior with Inverse Link Function

```r
fit_3 <- stan("poisson_N.stan",
              data = list(N = nrow(faithful), y = faithful$waiting,
                          loc = log(60), scal = 5))
```

```r
fit_3
```

```
## Inference for Stan model: poisson_N.
## 4 chains, each with iter=2000; warmup=1000; thin=1;
## post-warmup draws per chain=1000, total post-warmup draws=4000.
##
##          mean se_mean   sd     2.5%      25%      50%      75%    97.5% n_eff Rhat
## eta      4.26    0.00 0.01     4.25     4.26     4.26     4.27     4.28  1589    1
## mu      70.91    0.01 0.53    69.87    70.56    70.90    71.25    71.95  1588    1
## lp__ -1197.30    0.02 0.74 -1199.45 -1197.47 -1197.00 -1196.82 -1196.77  1748    1
##
## Samples were drawn using NUTS(diag_e) at Mon May 11 08:15:33 2020.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

Mixture Model

```
data {
  int<lower = 0> N;     // number of observations
  int<lower = 0> y[N];  // count outcomes

  real<lower = 0> loc;  // location of normal prior
  real<lower = 0> scal; // scal of normal prior
}
parameters {
  vector[2] eta;                 // log of means
  real<lower = 0, upper = 1> pi; // mixture probability
}
model {
  target += normal_lpdf(eta | loc, scal); // log prior PDF
  target += log_mix(pi, poisson_log_lpmf(y | eta[1]),
                        poisson_log_lpmf(y | eta[2]));
} // mixture log-likelihood ^^^
generated quantities {
  vector[2] mu = exp(eta);
}
```

Posterior from Mixture Model

```
fit_4 <- stan("poisson_mix.stan",
              data = list(N = nrow(faithful), y = faithful$waiting,
                          loc = log(60), scal = 5))
```

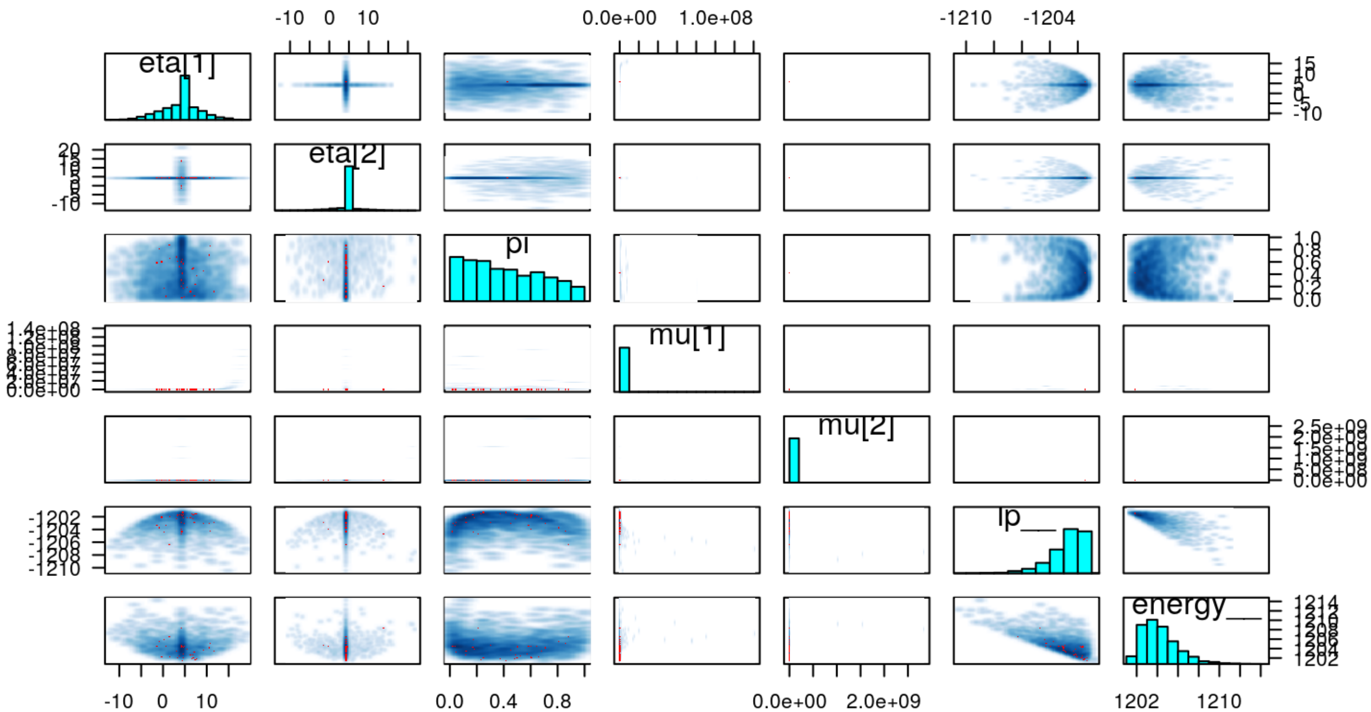
```
fit_4
```

```
...
##                mean   se_mean          sd     2.5%      25%      50%      75%     97.5%
## eta[1]         4.00      0.07        4.20    -4.93     1.89     4.26     6.08     13.03
## eta[2]         4.15      0.05        2.54    -2.65     4.25     4.26     4.27     10.50
## pi             0.41      0.11        0.28     0.02     0.18     0.38     0.64      0.94
## mu[1]     201167.36  57254.17  3451435.10     0.01     6.59    70.83   435.85 456952.18
## mu[2]    1371800.36 846237.63 53560927.94     0.07    70.36    70.88    71.36  36489.43
## lp__       -1202.76      0.03        1.26 -1206.07 -1203.34 -1202.46 -1201.82  -1201.31
...
```

- See also https://mc-stan.org/users/documentation/case-studies/identifying_mixture_models.html

Pairs Plot

```
pairs(fit_4)
```



Correct Mixture Model

```
data {
  int<lower = 0> N;     // number of observations
  int<lower = 0> y[N];  // count outcomes
  vector<lower = 0>[2] loc;  // location of normal prior
  vector<lower = 0>[2] scal; // scal of normal prior
}
parameters {
  ordered[2] eta;                // log of means
  real<lower = 0, upper = 1> pi; // mixture probability
}
model {
  target += normal_lpdf(eta | loc, scal);
  target += normal_lccdf(eta[1] | loc[2], scal[2]); // truncation of PDF for eta[2]
  target += log_mix(pi, poisson_log_lpmf(y | eta[1]),
                        poisson_log_lpmf(y | eta[2]));
}
generated quantities {
  vector[2] mu = exp(eta);
}
```

Breakout Rooms: Probit Model

- Suppose y_n indicates whether a person is in the labor force
- Use Bernoulli likelihood in a Stan program with the normal CDF as the inverse link function

```
data {
  // all knowns you condition on, including prior stuff
}
parameters {
  // unknowns
}
model {
  // numerator of Bayes Rule in log units
  // hint: Phi() evaluates the standard normal CDF
}
```

Optional `transformed data` Block

- Is executed only once before the iterations start
- Comes after the `data` block and used to calculate needed functions
- Not necessary if calling Stan from R with everything in `data`
- Can use it to check that data was passed correctly from R
- Need to declare objects before they can be assigned (=) but can be on the same line
- All declarations must come directly after the opening `{`

Using the brms Package to Generate Stan Code

- You do not need to start writing with a blank Stan program; you can use the `make_stancode` function in the *brms* package to look at or modify the code `brm` generaes
- Also, you can use `make_standata` to generate a named list of R objects that need to be passed to the `data` block of the Stan program

```
library(brms)
code <- make_stancode(count ~ log(Age) + Trt  + (1 | patient),
                      data = epilepsy, family = poisson(),
                      prior = c(prior(student_t(5, 0, 10), class = b),
                                prior(cauchy(0, 2), class = sd)))
dat <- make_standata(count ~ log(Age) + Trt + (1 | patient),
                     data = epilepsy, family = poisson(),
                     prior = c(prior(student_t(5, 0, 10), class = b),
                               prior(cauchy(0, 2), class = sd)))
```

Generated Data List

```
## List of 10
##  $ N         : int 236
##  $ Y         : num [1:236(1d)] 5 3 2 4 7 5 6 40 5 14 ...
##  $ K         : int 3
##  $ X         : num [1:236, 1:3] 1 1 1 1 1 1 1 1 1 1 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:236] "1" "2" "3" "4" ...
##   .. ..$ : chr [1:3] "Intercept" "logAge" "Trt1"
##   ..- attr(*, "assign")= int [1:3] 0 1 2
##   ..- attr(*, "contrasts")=List of 1
##   .. ..$ Trt: num [1:2, 1] 0 1
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. ..$ : chr [1:2] "0" "1"
##   .. .. .. ..$ : chr "1"
##  $ Z_1_1     : num [1:236(1d)] 1 1 1 1 1 1 1 1 1 1 ...
##   ..- attr(*, "dimnames")=List of 1
##   .. ..$ : chr [1:236] "1" "2" "3" "4" ...
##  $ J_1       : int [1:236(1d)] 1 2 3 4 5 6 7 8 9 10 ...
##  $ N_1       : int 59
##  $ M_1       : int 1
##  $ NC_1      : int 0
##  $ prior_only: int 0
##  - attr(*, "class")= chr "standata"
```

Generated Stan Code

```
data {
  int<lower=1> N;  // number of observations
  int Y[N];  // response variable
  int<lower=1> K;  // number of population-level effects
  matrix[N, K] X;  // population-level design matrix
  // data for group-level effects of ID 1
  int<lower=1> N_1;  // number of grouping levels
  int<lower=1> M_1;  // number of coefficients per level
  int<lower=1> J_1[N];  // grouping indicator per observation
  // group-level predictor values
  vector[N] Z_1_1;
  int prior_only;  // should the likelihood be ignored?
}
transformed data {
  int Kc = K - 1;
  matrix[N, Kc] Xc;  // centered version of X without an intercept
  vector[Kc] means_X;  // column means of X before centering
  for (i in 2:K) {
    means_X[i - 1] = mean(X[, i]);
    Xc[, i - 1] = X[, i] - means_X[i - 1];
  }
}
parameters {
  vector[Kc] b;  // population-level effects
  real Intercept;  // temporary intercept for centered predictors
  vector<lower=0>[M_1] sd_1;  // group-level standard deviations
  vector[N_1] z_1[M_1];  // standardized group-level effects
}
transformed parameters {
  vector[N_1] r_1_1;  // actual group-level effects
  r_1_1 = (sd_1[1] * (z_1[1]));
}
model {
  // initialize linear predictor term
  vector[N] mu = Intercept + Xc * b;
  for (n in 1:N) {
    // add more terms to the linear predictor
    mu[n] += r_1_1[J_1[n]] * Z_1_1[n];
  }
  // priors including all constants
  target += student_t_lpdf(b | 5, 0, 10);
  target += student_t_lpdf(Intercept | 3, 1, 10);
  target += cauchy_lpdf(sd_1 | 0, 2)
    - 1 * cauchy_lccdf(0 | 0, 2);
  target += normal_lpdf(z_1[1] | 0, 1);
  // likelihood including all constants
  if (!prior_only) {
    target += poisson_log_lpmf(Y | mu);
  }
}
generated quantities {
  // actual population-level intercept
  real b_Intercept = Intercept - dot_product(means_X, b);
}
```

Data for Hierarchical Model of Bowling

```r
ROOT <- "https://www.cs.rpi.edu/academics/courses/fall14/csci1200/"
US_Open2010 <- readLines(paste0(ROOT, "hw/02_bowling_classes/2010_US_Open.txt"))
x1_x2 <- lapply(US_Open2010, FUN = function(x) {
  pins <- scan(what = integer(), sep = " ", quiet = TRUE,
               text = sub("^[a-zA-Z_ \']+(.*$)", "\\1", x))
  results <- matrix(NA_integer_, 10, 2)
  pos <- 1
  for (f in 1:10) {
    x1 <- pins[pos]
    if (x1 == 10) results[f, ] <- c(x1, 0L)
    else {
      pos <- pos + 1
      x2 <- pins[pos]
      results[f, ] <- c(x1, x2)
    }
    pos <- pos + 1
  }
  return(results)
}) # 30 element list each with a 10x2 integer array of pins knocked down
```

Illustrative Data

```r
names(x1_x2) <- sub("^([a-zA-Z_ \']+)( .*$)", "\\1", US_Open2010)
x1_x2[1]
```

```
## $`Mike Scroggins`
##       [,1] [,2]
##  [1,]    9    1
##  [2,]   10    0
##  [3,]    8    2
##  [4,]   10    0
##  [5,]    8    2
##  [6,]    9    1
##  [7,]   10    0
##  [8,]    8    1
##  [9,]   10    0
## [10,]   10    0
```

Multilevel Stan Program for Bowling

```
#include bowling_kernel.stan
data {
  int<lower = 0> J;                                // number of bowlers
  int<lower = 0, upper = 10> x1_x2[J, 10, 2]; // results of each bowler's frames
  vector<lower = 0>[11] a;                         // shapes for Dirichlet prior on mu
  real<lower = 0> s;                               // scale factor on top of theta
}
parameters {
  simplex[11] mu;        // overall probability of knocking down 0:10 pins
  real<lower = 0> theta; // concentration parameter across bowlers
  simplex[11] pi[J];     // bowler's probability of knocking down 0:10 pins
}
model { // target becomes the log-numerator of Bayes Rule
  vector[11] mu_theta = mu * theta * s;                  // not saved in results
  for (j in 1:J) // bowling_kernel() is defined in the functions block
    target += bowling_kernel(pi[j], mu_theta, x1_x2[j]); // note indexing
  target += dirichlet_lpdf(mu | a);                      // prior on mu
  target += exponential_lpdf(theta | 1);                 // prior on theta
}
```

What Was the **bowling_kernel** Function?

```
functions { /* bowling_kernel.stan */
  real bowling_kernel(vector pi, vector a,
                      int [ , ] x1_x2) {
    real log_like = 0; // categorical
    real log_prior = dirichlet_lpdf(pi | a);
    for (n in 1:dims(x1_x2)[1]) {
      int x1 = x1_x2[n, 1];
      log_like += log(pi[x1 + 1]);
      if (x1 < 10) {  // not a strike
        int np1 = 10 - x1 + 1;
        vector[np1] pi_ = pi[1:np1]
                        / sum(pi[1:np1]);
        int x2 = x1_x2[n, 2];
        log_like += log(pi_[x2 + 1]);
      }
    }
    return log_prior + log_like;
  }
}
```

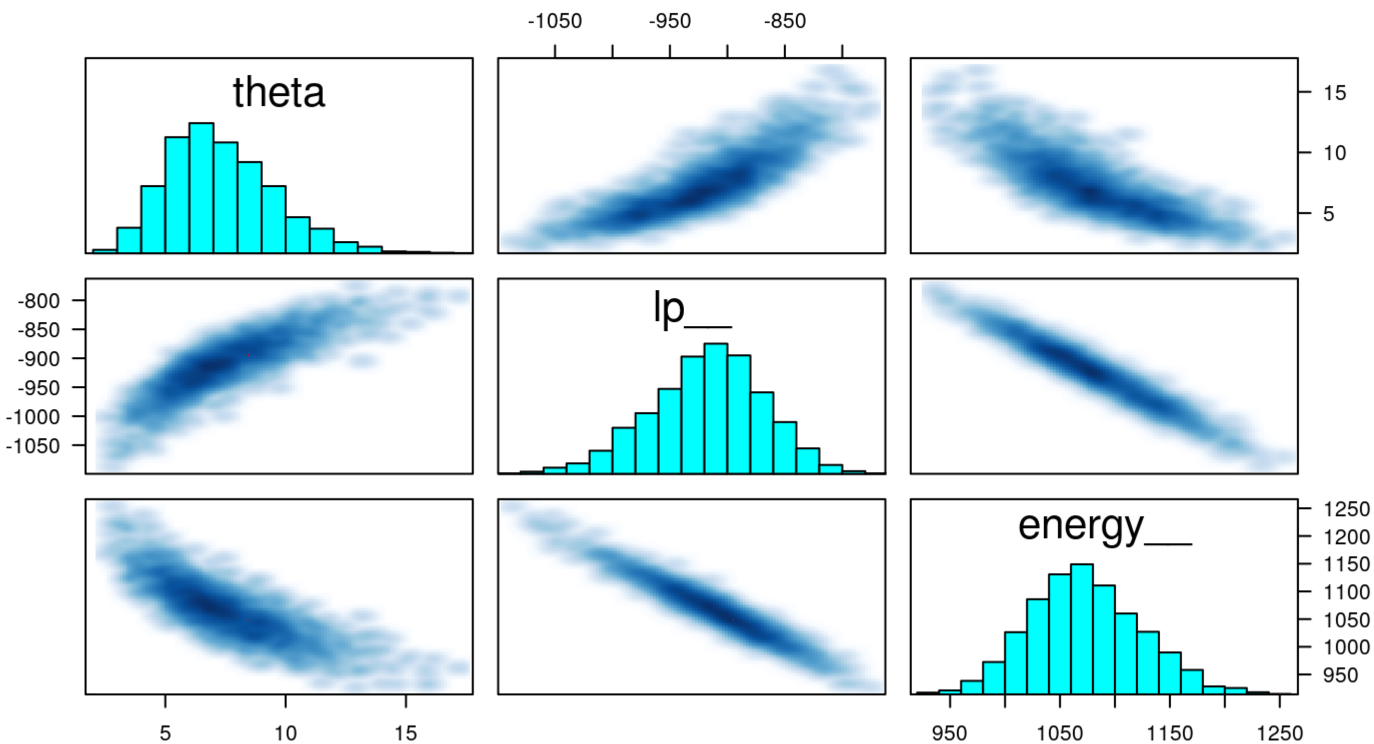
Multilevel Posterior Distribution

```
post_mlm <- stan("bowling_mlm.stan", control = list(adapt_delta = 0.85), refresh = 0,
                 data = list(J = length(x1_x2), x1_x2 = x1_x2, a = 1:11, s = 10))
print(post_mlm, pars = "pi", include = FALSE, digits = 2)
```

```
...
##             mean se_mean    sd     2.5%     25%     50%     75%   97.5% n_eff Rhat
## mu[1]       0.00    0.00  0.00     0.00    0.00    0.00    0.00    0.00   391 1.01
## mu[2]       0.01    0.00  0.00     0.01    0.01    0.01    0.01    0.02   713 1.00
## mu[3]       0.02    0.00  0.00     0.01    0.01    0.02    0.02    0.03   728 1.01
## mu[4]       0.02    0.00  0.00     0.01    0.01    0.02    0.02    0.03   703 1.01
## mu[5]       0.02    0.00  0.01     0.01    0.01    0.02    0.02    0.03   602 1.00
## mu[6]       0.03    0.00  0.01     0.01    0.02    0.02    0.03    0.04   556 1.00
## mu[7]       0.04    0.00  0.01     0.02    0.03    0.04    0.05    0.07   634 1.01
## mu[8]       0.08    0.00  0.01     0.06    0.07    0.08    0.09    0.11   663 1.01
## mu[9]       0.13    0.00  0.02     0.10    0.12    0.13    0.14    0.17   824 1.00
## mu[10]      0.23    0.00  0.02     0.19    0.22    0.23    0.25    0.28   879 1.00
## mu[11]      0.42    0.00  0.03     0.37    0.40    0.42    0.44    0.47   867 1.00
## theta       7.33    0.16  2.23     3.71    5.72    7.05    8.71   12.32   198 1.02
## lp__     -917.64    4.80 49.05 -1019.03 -948.83 -915.00 -883.89 -828.43   104 1.04
...
```

Pairs Plot

```
pairs(post_mlm, pars = c("mu", "pi"), include = FALSE)
```



Meta-Analysis

- “Meta-analysis” of previous studies is popular in some fields such as education and medicine
- Can be written as a multi-level model where each study is its own “group” with its own intercept that captures the difference between what each study is estimating and what it wants to estimate
- Outcome is the point estimate for each Frequentist study
- Estimated standard error from each Frequentist study is treated as an exogenous known

Simulation Based Callibration (SBC)

- Talts et al. (2018) proposes SBC
- The posterior distribution conditional on data drawn from the prior predictive distribution cannot be systematically different from the prior
- Appearances to the contrary are due to failure of the software
- Provides a way to limit the fourth source of uncertainty by repeatedly

  1. Drawing $\tilde{\boldsymbol{\theta}}$ the prior of $\boldsymbol{\theta}$
  2. Drawing from the prior predictive distribution of $\widetilde{\mathbf{y}} \mid \tilde{\boldsymbol{\theta}}$
  3. Drawing from the posterior distribution of $\boldsymbol{\theta} \mid \widehat{\mathbf{y}}$
  4. Evaluating whether $\boldsymbol{\theta} > \tilde{\boldsymbol{\theta}}$

- See also this blog post

The **data** and **transformed data** Blocks

```
data {
  int<lower = 1> N; // number of studies
  vector<lower = 0>[N] se; // std. errors
}
transformed data { // for SBC
  vector[N] se2 = square(se);
  real mu_ = normal_rng(0, 1); // truth
  real tau_ = exponential_rng(1);
  vector[N] y; // estimates of mu
  for (n in 1:N) { // prior predictions
    real epsilon = normal_rng(0, 1);
    real delta = mu_ + tau_ * epsilon;
    y[n] = normal_rng(delta, se[n]);
  }
}
```

- Other blocks follow on the next slide

```
parameters {
  real mu;
  real<lower = 0> tau;
}
model {
  target += normal_lpdf(mu | 0, 1);
  target += exponential_lpdf(tau | 1);
  target += normal_lpdf(y | mu, sqrt(square(tau) + se2));
}
generated quantities {
  vector[N] y_ = y;  // copy of prior predictions
  vector[N] log_lik; // for loo()
  vector[2] pars_;   // copy of prior realizations
  int ranks_[2] = {mu > mu_, tau > tau_}; // for plot
  pars_[1] = mu_;
  pars_[2] = tau_;
  for (n in 1:N) {
    real s = sqrt(square(tau) + se2[n]);
    log_lik[n] = normal_lpdf(y[n] | mu, s);
  }
}
```

Doing Simulation Based Calibration

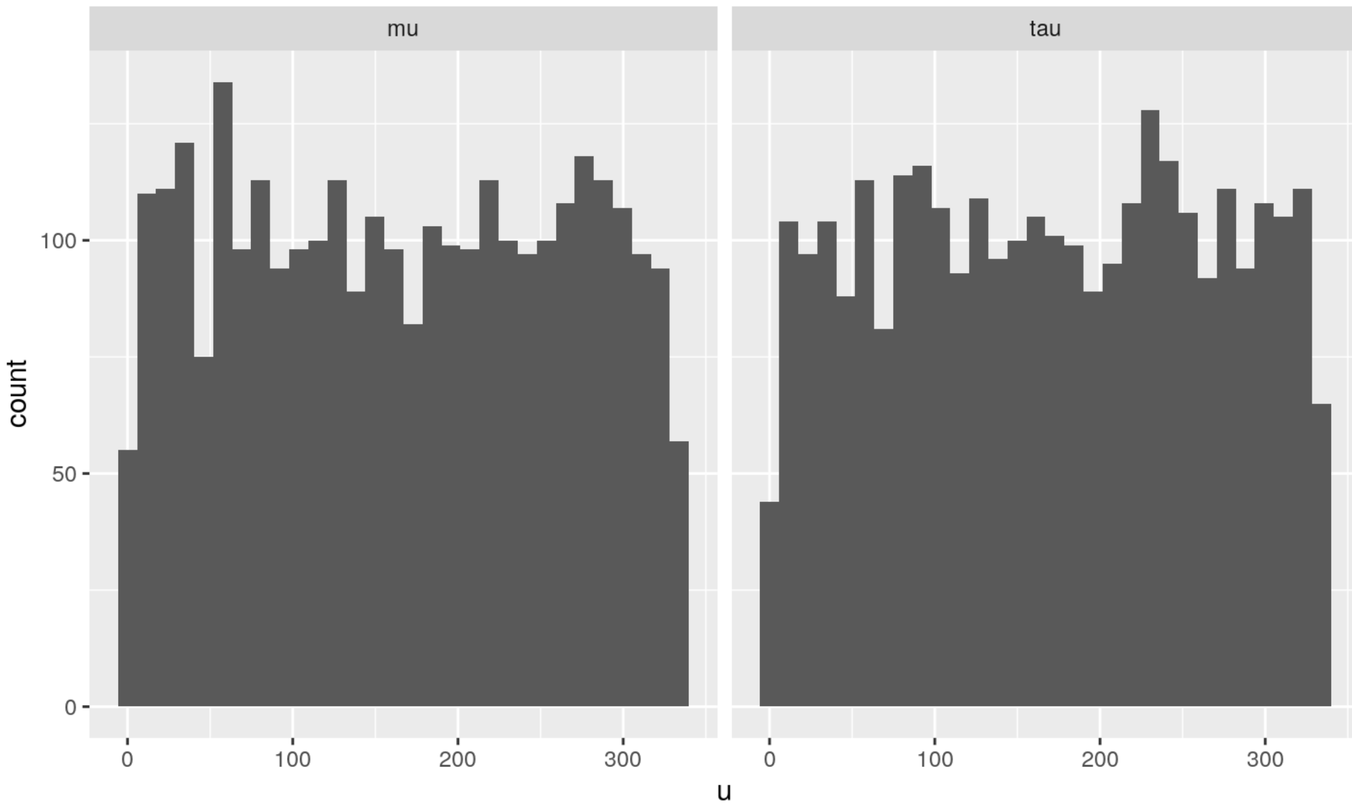
```
sm <- stan_model("meta_analysis.stan")
data("towels", package = "metaBMA")
dat <- list(N = nrow(towels), se = towels$SE)
results <- sbc(sm, data = dat, M = 3000, refresh = 0, control = list(adapt_delta = 0.85))
```

```
results
```

```
## 32 chains had divergent transitions after warmup
## there were a total of 53 divergent transitions across all chains
## Aggregate Pareto k estimates:
## cut_pareto_k
##  (-Inf,0.5]   (0.5,0.7]     (0.7,1]    (1, Inf]
## 0.854285714 0.089761905 0.048095238 0.007857143
```

SBC Plot

```
plot(results) # use to visualize uniformity of order statistics
```



Oregon Medicaid Experiment Data

```
library(haven); library(dplyr)
oregon <- as_factor(read_dta("individual_voting_data.dta"))
(collapsed <- group_by(oregon, t = treatment, s = numhh_list, x = ohp_all_ever_nov2008) %>%
              summarize(y = sum(vote_presidential_2008_1), nmy = n() - y)  %>% as.data.frame)
```

```
##    t                                      s            x     y   nmy
## 1  0                         signed self up NOT enrolled 11833 22560
## 2  0                         signed self up     Enrolled   954  2352
## 3  0 signed self up + 1 additional person NOT enrolled  2290  4469
## 4  0 signed self up + 1 additional person     Enrolled   168   444
## 5  0 signed self up + 2 additional people NOT enrolled     1    15
## 6  0 signed self up + 2 additional people     Enrolled     0     2
## 7  1                         signed self up NOT enrolled  4141  8245
## 8  1                         signed self up     Enrolled  2661  4782
## 9  1 signed self up + 1 additional person NOT enrolled  2398  4471
## 10 1 signed self up + 1 additional person     Enrolled  1043  1953
## 11 1 signed self up + 2 additional people NOT enrolled    32    72
## 12 1 signed self up + 2 additional people     Enrolled    14    22
```

Oregon Medicaid Experiment in Symbols

Let $s_n \in \{1, 2, 3\}$ be the number of adults in n's household. Let t_n indicate whether any of them wins the Medicaid lottery. Let x_n indicate whether n enrolls in Medicaid and y_n indicate whether n votes.

$$
\begin{aligned}
\alpha_1 &\sim GLD(\mathbf{q}_\alpha) & \beta_1 &\sim GLD(\mathbf{q}_\beta) \\
\alpha_2 &\sim GLD(\mathbf{q}_\alpha) & \beta_2 &\sim GLD(\mathbf{q}_\beta) \\
\alpha_3 &\sim GLD(\mathbf{q}_\alpha) & \beta_3 &\sim GLD(\mathbf{q}_\beta) \\
\lambda &\sim GLD(\mathbf{q}_\lambda) & \Delta &\sim GLD(\mathbf{q}_\Delta) \\
& & \rho &\sim GLD(\mathbf{q}_\rho) \\
\forall n : \epsilon_n &\sim \mathcal{N}(0, 1) & \forall n : \nu_n &\sim \mathcal{N}\left(0 + \rho(\epsilon_n - 0), \sqrt{1 - \rho^2}\right) \\
\forall n : x_n^* &= \alpha_{s_n} + \lambda \times t_n + \epsilon_n & \forall n : y_n^* &= \beta_{s_n} + \Delta \times x_n + \nu_n \\
\forall n : x_n &= \mathcal{I}\{x_n^* > 0\} & \forall n : y_n &= \mathcal{I}\{y_n^* > 0\}
\end{aligned}
$$

- If $\rho \neq 0$, x_n is NOT independent of ν_n so $\mathbb{E} y_n^* \mid s_n, x_n = \beta_{s_n} + \Delta \times x_n + \mathbb{E}\nu_n \mid s_n, x_n$
- $\mathbb{E}[\nu_n \mid s_n, x_n = 0] = \mathbb{E}[\nu_n \mid s_n, x_n^* < 0] = \mathbb{E}[\nu_n \mid -\alpha_{s_n} - \lambda \times t_n > \epsilon_n] = \rho \frac{\phi(-\alpha_{s_n} - \lambda \times t_n)}{\Phi(\alpha_{s_n} + \lambda \times T_n)}$
- $\mathbb{E}[\nu_n \mid s_n, x_n = 1] = \mathbb{E}[\nu_n \mid s_n, x_n^* > 0] = \mathbb{E}[\nu_n \mid -\alpha_{s_n} - \lambda \times t_n < \epsilon_n] = -\rho \frac{\phi(-\alpha_{s_n} - \lambda \times t_n)}{\Phi(-\alpha_{s_n} - \lambda \times T_n)}$

Breakout Rooms

Draw from that prior predictive distribution within the transformed data block

```
data {
  int<lower = 1> N;
  int<lower = 0, upper = 1> t[N]; // win Medicaid lottery?
  int<lower = 1, upper = 3> s[N]; // number of adults in household
  // more stuff
}
transformed data {
  int x[N]; // enrolls in Medicaid?
  int y[N]; // votes in election?
  // draw parameters from the prior distributions
  for (n in 1:N) {
    // fill in x[n] and y[n]
  }
}
```

Posterior PDF for Oregon Medicaid Experiment

$$
\begin{gathered}
f\left(\boldsymbol{\alpha}, \lambda, \boldsymbol{\beta}, \Delta, \rho \mid \mathbf{s}, \mathbf{t}, \mathbf{x}, \mathbf{y}\right) \propto f\left(\boldsymbol{\alpha}, \lambda, \boldsymbol{\beta}, \Delta, \rho\right) \times \\
\prod_{j=1}^{3} \Pr\left(\epsilon < -\alpha_j \bigcap \nu < -\beta_j\right)^{c_j} \times \prod_{j=1}^{3} \Pr\left(\epsilon < -\alpha_j \bigcap \nu < \beta_j\right)^{c_{j+3}} \times \\
\prod_{j=1}^{3} \Pr\left(\epsilon < \alpha_j \bigcap \nu < -\beta_j - \Delta\right)^{c_{j+6}} \times \prod_{j=1}^{3} \Pr\left(\epsilon < \alpha_j \bigcap \nu < \beta_j + \Delta\right)^{c_{j+9}} \times \\
\prod_{j=1}^{3} \Pr\left(\epsilon < -\alpha_j - \lambda \bigcap \nu < -\beta_j\right)^{c_{j+12}} \times \prod_{j=1}^{3} \Pr\left(\epsilon < -\alpha_j - \lambda \bigcap \nu < \beta_j\right)^{c_{j+15}} \times \\
\prod_{j=1}^{3} \Pr\left(\epsilon < \alpha_j + \lambda \bigcap \nu < -\beta_j - \Delta\right)^{c_{j+18}} \times \prod_{j=1}^{3} \Pr\left(\epsilon < \alpha_j + \lambda \bigcap \nu < \beta_j + \Delta\right)^{c_{j+21}}
\end{gathered}
$$

where c_i indicates the count of people in the i-th stratum. You also need a function to evaluate to the bivariate normal CDF.

Directed Acyclic Graphs (DAGs)

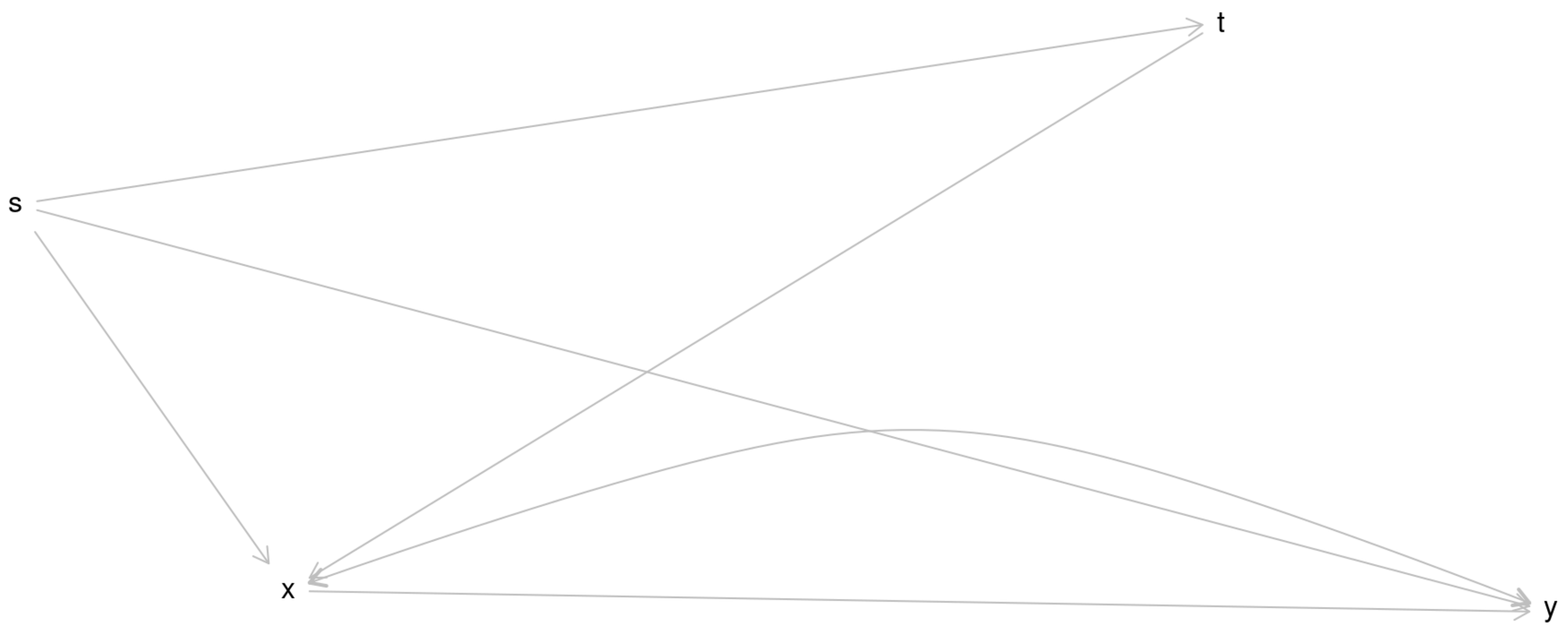
- DAGs are a popular tool for describing a theoretical data-generating process
  - Typically do not depict parameters or distributional assumptions
  - Most often used to algorithmically conclude whether a causal effect could be solved for given infinite data
- Most useful survey paper is Elwert (2013)
- Most references are to some work of Pearl
- Most useful website is http://dagitty.net/ which also has an R package

CausalQueries

- Any DAG with only observed nodes being binary variables can be reprented as a Stan program
- Primitive parameters are simplex variables of “causal types” like $\boldsymbol{\lambda}^\top = \begin{bmatrix} \lambda_a & \lambda_b & \lambda_c & \lambda_d \end{bmatrix}$ except in general there are 2^K “causal types” where K is the number of parents of a node
- Likelihood is multinomial with a potentially huge number of categories
- Can query a model either before or after updating your beliefs about the parameters with data to answer various causal counterfactual questions
- Computationally difficult and difficult to specify informative priors

CausalQueries: DAG Specification

```
library(CausalQueries)
model <- make_model("t -> x -> y; t <- s -> x; s -> y") %>%
  set_confound(confound = list(x = "y[x = 1] == 1"))
plot(model)
```



CausalQueries: Data Compacting

```
dataset <- transmute(oregon, t = treatment, y = vote_presidential_2008_1,
                     x = (ohp_all_ever_nov2008 == "Enrolled"),
                     s = numhh_list != "signed self up")
```

```
(compact_data <- collapse_data(dataset, model))
```

```
##       event strategy count
## 1  s0t0x0y0     stxy 22560
## 2  s1t0x0y0     stxy  4484
## 3  s0t1x0y0     stxy  8245
## 4  s1t1x0y0     stxy  4543
## 5  s0t0x1y0     stxy  2352
## 6  s1t0x1y0     stxy   446
## 7  s0t1x1y0     stxy  4782
## 8  s1t1x1y0     stxy  1975
## 9  s0t0x0y1     stxy 11833
## 10 s1t0x0y1     stxy  2291
## 11 s0t1x0y1     stxy  4141
## 12 s1t1x0y1     stxy  2430
## 13 s0t0x1y1     stxy   954
## 14 s1t0x1y1     stxy   168
## 15 s0t1x1y1     stxy  2661
## 16 s1t1x1y1     stxy  1057
```

CausalQueries: Drawing from the Posterior

```r
post <- update_model(model, dataset, iter = 1000, chains = 2) # can pass other arguments
result <- query_model(post, using = "posteriors", queries = list(ATE = "c(y[x=1] - y[x=0])"),
                      given = list(TRUE, "x[t=1] > x[t=0]",  "x==0",  "x==1"))
```

```r
result
```

```
##   Query           Given      Using   mean    sd
## 1   ATE               - posteriors  0.039 0.114
## 2   ATE x[t=1] > x[t=0] posteriors  0.011 0.017
## 3   ATE           x==0 posteriors  0.052 0.142
## 4   ATE           x==1 posteriors -0.015 0.013
```