Probabilistic programming language

- Wikipedia “A probabilistic programming language (PPL) is a programming language designed to describe probabilistic models and then perform inference in those models”

Probabilistic programming language

- Wikipedia “A probabilistic programming language (PPL) is a programming language designed to describe probabilistic models and then perform inference in those models”
- To make probabilistic programming useful
  - inference has to be as automatic as possible
  - diagnostics for telling if the automatic inference doesn’t work
  - easy workflow (to reduce manual work)
  - fast enough (manual work replaced with automation)

Probabilistic programming

- Enables agile workflow for developing probabilistic models
  - language
  - automated inference
  - diagnostics
- Many frameworks Stan, PyMC, Pyro (Uber), TFP (Google), Turing.jl, JAGS, ELFI, ...
  - Short review of the landscape:
    Štrumbelj et al. (2023). Past, Present, and Future of Software for Bayesian Inference. *Statistical Science*, 39(1):46-61. https://doi.org/10.1214/23-STS907.

Stan - probabilistic programming framework

- Language, inference engine, user interfaces, documentation, case studies, diagnostics, packages, ...
  - autodiff to compute gradients of the log density

mc-stan.org

Stan - probabilistic programming framework

- Language, inference engine, user interfaces, documentation, case studies, diagnostics, packages, ...
  - autodiff to compute gradients of the log density
- Most popular, 200K+ users in social, biological, and physical sciences, medicine, engineering, and business

mc-stan.org

Stan - probabilistic programming framework

- Language, inference engine, user interfaces, documentation, case studies, diagnostics, packages, ...
  - autodiff to compute gradients of the log density
- Most popular, 200K+ users in social, biological, and physical sciences, medicine, engineering, and business
- Several full time developers, 40+ developers, 100+ contributors

mc-stan.org

Stan - probabilistic programming framework

- Language, inference engine, user interfaces, documentation, case studies, diagnostics, packages, ...
  - autodiff to compute gradients of the log density
- Most popular, 200K+ users in social, biological, and physical sciences, medicine, engineering, and business
- Several full time developers, 40+ developers, 100+ contributors
- R, Python, Julia, Scala, Stata, command line interfaces
- 300+ R packages using Stan

mc-stan.org

Stan

- Stanislaw Ulam (1909-1984)
  - Monte Carlo method
  - H-Bomb

Binomial model - Stan code

Domain-specific language for constructing models
with common *distributed as* $\sim$ notation

```
data {
  int<lower=0> N;          // number of experiments
  int<lower=0,upper=N> y; // number of successes
}

parameters {
  real<lower=0,upper=1> theta; // parameter of the binomial
}

model {
  theta ~ beta(1, 1);       // prior
  y ~ binomial(N, theta);  // observation / data model
}
```

Binomial model - Stan code

Domain-specific language for constructing models
with common *distributed as* $\sim$ notation

```
data {
  int<lower=0> N;          // number of experiments
  int<lower=0,upper=N> y; // number of successes
}

parameters {
  real<lower=0,upper=1> theta; // parameter of the binomial
}

model {
  theta ~ beta(1, 1);       // prior
  y ~ binomial(N, theta);  // observation / data model
}
```

Binomial model - Stan code

Domain-specific language for constructing models
with common *distributed as* $\sim$ notation

```
data {
  int<lower=0> N;          // number of experiments
  int<lower=0,upper=N> y; // number of successes
}

parameters {
  real<lower=0,upper=1> theta; // parameter of the binomial
}

model {
  theta ~ beta(1, 1);       // prior
  y ~ binomial(N, theta);  // observation / data model
}
```

Binomial model - Stan code

```
data {
  int<lower=0> N;          // number of experiments
  int<lower=0,upper=N> y; // number of successes
}
```

- Data type and size are declared
- Stan checks that given data matches type and constraints

Binomial model - Stan code

```
data {
  int<lower=0> N;          // number of experiments
  int<lower=0,upper=N> y; // number of successes
}
```

- Data type and size are declared
- Stan checks that given data matches type and constraints
  - If you are not used to strong typing, this may feel annoying, but it will reduce the probability of coding errors, which will reduce probability of data analysis errors

Binomial model - Stan code

```
parameters {
  real<lower=0,upper=1> theta; // parameter of the binomial
}
```

- Only continuous parameters allowed (discrete parameters can often be integerated out in the model block)
- Parameters may have constraints
- Stan makes transformation to unconstrained space and samples in unconstrained space
  - e.g. log transformation for `<lower=a>`
  - e.g. logit transformation for `<lower=a,upper=b>`

Binomial model - Stan code

```
parameters {
  real<lower=0,upper=1> theta; // parameter of the binomial
}
```

- Only continuous parameters allowed (discrete parameters can often be integerated out in the model block)
- Parameters may have constraints
- Stan makes transformation to unconstrained space and samples in unconstrained space
  - e.g. log transformation for `<lower=a>`
  - e.g. logit transformation for `<lower=a,upper=b>`
- For these declared transformation Stan automatically takes into account the Jacobian of the transformation (see BDA3 p. 21)

Binomial model - Stan code

```
model {
  theta ~ beta(1, 1);      // prior
  y ~ binomial(N, theta); // observation model
}
```

- ~ defines a *distribution statement*
  e.g. *y is distributed as* binomial(*N, theta*)

Binomial model - Stan code

```
model {
  theta ~ beta(1, 1);      // prior
  y ~ binomial(N, theta); // observation model
}
```

- ~ defines a *distribution statement*
  e.g. *y is distributed as* binomial(*N, theta*)
- these can be written also as *log density increment statements*
  left side of | denotes what is distributed as, e.g., binomial

```
model {
  target += beta_lpdf(theta | 1, 1);
  target += binomial_lpmf(y | N, theta);
}
```

Binomial model - Stan code

```
model {
  theta ~ beta(1, 1);      // prior
  y ~ binomial(N, theta); // observation model
}
```

- `~` defines a *distribution statement*
  e.g. *y is distributed as* binomial(*N, theta*)
- these can be written also as *log density increment statements*
  left side of | denotes what is distributed as, e.g., binomial

```
model {
  target += beta_lpdf(theta | 1, 1);
  target += binomial_lpmf(y | N, theta);
}
```

- `target` is the log posterior density (Lecture 4 discussed log)

Binomial model - Stan code

```
model {
  theta ~ beta(1, 1);      // prior
  y ~ binomial(N, theta); // observation model
}
```

- `~` defines a *distribution statement*
  e.g. *y is distributed as* binomial(*N, theta*)
- these can be written also as *log density increment statements*
  left side of | denotes what is distributed as, e.g., binomial

```
model {
  target += beta_lpdf(theta | 1, 1);
  target += binomial_lpmf(y | N, theta);
}
```

- `target` is the log posterior density (Lecture 4 discussed log)
- `_lpdf` for continuous, `_lpmf` for discrete distributions (left of `|`)

Binomial model - Stan code

```
model {
  theta ~ beta(1, 1);      // prior
  y ~ binomial(N, theta); // observation model
}
```

- `~` defines a *distribution statement*
  e.g. *y is distributed as* binomial(*N, theta*)
- these can be written also as *log density increment statements*
  left side of | denotes what is distributed as, e.g., binomial

```
model {
  target += beta_lpdf(theta | 1, 1);
  target += binomial_lpmf(y | N, theta);
}
```

- `target` is the log posterior density (Lecture 4 discussed log)
- `_lpdf` for continuous, `_lpmf` for discrete distributions (left of `|`)
- if `y` are data, and `theta` is a parameter, then that term defines log likelihood

Binomial model - Stan code

```
model {
  theta ~ beta(1, 1);      // prior
  y ~ binomial(N, theta); // observation model
}
```

- `~` defines a *distribution statement*
  e.g. *y is distributed as* binomial(*N, theta*)
- these can be written also as *log density increment statements*
  left side of | denotes what is distributed as, e.g., binomial

```
model {
  target += beta_lpdf(theta | 1, 1);
  target += binomial_lpmf(y | N, theta);
}
```

- `target` is the log posterior density (Lecture 4 discussed log)
- `_lpdf` for continuous, `_lpmf` for discrete distributions (left of `|`)
- if `y` are data, and `theta` is a parameter, then that term defines log likelihood
- for Stan sampler there is no difference between prior and likelihood, all that matters is the final `target`

Stan

- You can write in Stan language any program to compute the log density (Stan language is Turing complete)

Stan

- You can write in Stan language any program to compute the log density (Stan language is Turing complete)
- Stan compiles (transpiles) the model written in Stan language to C++
  - this makes the sampling for complex models and bigger data faster
  - also makes Stan models easily portable, you can use your own favorite interface and scripting language for manipulating data and inference results (e.g. R, Python, Julia, Stata, ...)

CmdStanR

CmdStanR is an R interface for Stan

```
# Load CmdStanR
library(cmdstanr)

# Compile Stan model
mod_bin <- cmdstan_model(stan_file = 'binom.stan')

# Sample from the posterior given the model and data
d_bin <- list(N = 10, y = 7)
fit_bin <- mod_bin$sample(data = d_bin)

# Show summary and access draws
fit_bin$summary()
draws <- fit_bin$draws(format = "df")
```

CmdStanR

CmdStanR is an R interface for Stan

```r
# Load CmdStanR
library(cmdstanr)

# Compile Stan model
mod_bin <- cmdstan_model(stan_file = 'binom.stan')

# Sample from the posterior given the model and data
d_bin <- list(N = 10, y = 7)
fit_bin <- mod_bin$sample(data = d_bin)

# Show summary and access draws
fit_bin$summary()
draws <- fit_bin$draws(format = "df")
```

CmdStanR

CmdStanR is an R interface for Stan

```
# Load CmdStanR
library(cmdstanr)

# Compile Stan model
mod_bin <- cmdstan_model(stan_file = 'binom.stan')

# Sample from the posterior given the model and data
d_bin <- list(N = 10, y = 7)
fit_bin <- mod_bin$sample(data = d_bin)

# Show summary and access draws
fit_bin$summary()
draws <- fit_bin$draws(format = "df")
```

CmdStanR

CmdStanR is an R interface for Stan

```
# Load CmdStanR
library(cmdstanr)

# Compile Stan model
mod_bin <- cmdstan_model(stan_file = 'binom.stan')

# Sample from the posterior given the model and data
d_bin <- list(N = 10, y = 7)
fit_bin <- mod_bin$sample(data = d_bin)

# Show summary and access draws
fit_bin$summary()
draws <- fit_bin$draws(format = "df")
```

Stan

- Compilation (unless previously compiled model available)
- Pick random initial values for MCMC chains
- Run warm-up iterations including adaptation of mass matrix and step-size
- Sampling
- Generated quantities
- Save posterior draws
- Report divergences, ESS, $\widehat{R}$

Vaccine efficacy

- Randomized trial for influenza vaccine
- Participants were randomly assigned to vaccine and placebo groups:
  - out of 338 patients receiving the placebo, 16 were infected
  - out of 867 receiving the vaccine, 19 were infected

Vaccine efficacy

```
data {
  int<lower=0> N1;
  int<lower=0> y1;
  int<lower=0> N2;
  int<lower=0> y2;
}
parameters {
  real<lower=0,upper=1> theta1;
  real<lower=0,upper=1> theta2;
}
model {
  theta1 ~ beta(1, 1);
  theta2 ~ beta(1, 1);
  y1 ~ binomial(N1, theta1);
  y2 ~ binomial(N2, theta2);
}
generated quantities {
  real ve;
  ve = 1 - theta2 / theta1;
}
```

Vaccine efficacy

```
data {
  int<lower=0> N1;
  int<lower=0> y1;
  int<lower=0> N2;
  int<lower=0> y2;
}
parameters {
  real<lower=0,upper=1> theta1;
  real<lower=0,upper=1> theta2;
}
model {
  theta1 ~ beta(1, 1);
  theta2 ~ beta(1, 1);
  y1 ~ binomial(N1, theta1);
  y2 ~ binomial(N2, theta2);
}
generated quantities {
  real ve;
  ve = 1 - theta2 / theta1;
}
```

Vaccine efficacy

```
data {
  int<lower=0> N1;
  int<lower=0> y1;
  int<lower=0> N2;
  int<lower=0> y2;
}
parameters {
  real<lower=0,upper=1> theta1;
  real<lower=0,upper=1> theta2;
}
model {
  theta1 ~ beta(1, 1);
  theta2 ~ beta(1, 1);
  y1 ~ binomial(N1, theta1);
  y2 ~ binomial(N2, theta2);
}
generated quantities {
  real ve;
  ve = 1 - theta2 / theta1;
}
```

Vaccine efficacy

```
data {
  int<lower=0> N1;
  int<lower=0> y1;
  int<lower=0> N2;
  int<lower=0> y2;
}
parameters {
  real<lower=0,upper=1> theta1;
  real<lower=0,upper=1> theta2;
}
model {
  theta1 ~ beta(1, 1);
  theta2 ~ beta(1, 1);
  y1 ~ binomial(N1, theta1);
  y2 ~ binomial(N2, theta2);
}
generated quantities {
  real ve;
  ve = 1 - theta2 / theta1;
}
```

Vaccine efficacy

```
generated quantities {
  real ve;
  ve = 1 - theta2 / theta1;
}
```

- generated quantities is run after the sampling

Vaccine efficacy

```
d_bin2 <- list(N1 = 338, y1 = 16, N2 = 867, y2 = 19)
mod_bin2 <- cmdstan_model(stan_file = 'binom2.stan')
fit_bin2 <- mod_bin2$sample(data = d_bin2, refresh=1000)
```

```
> Running MCMC with 4 parallel chains...

Chain 1 Iteration:    1 / 2000 [  0%]  (Warmup)
Chain 1 Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain 1 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 1 Iteration: 2000 / 2000 [100%]  (Sampling)
...
All 4 chains finished successfully.
Mean chain execution time: 0.0 seconds.
Total execution time: 0.2 seconds.
```

Vaccine efficacy

```
options(posterior.num_args=list(sigfig=2))
fit_bin2$summary()
```

```
  variable     mean   median     sd    mad       q5      q95  rhat ess_bulk ess_tail
  <chr>       <dbl>    <dbl>  <dbl>  <dbl>    <dbl>    <dbl> <dbl>    <dbl>    <dbl>
1 lp__     -164.    -163.     1.0    0.71   -166.    -163.       1.0    2014.    2483.
2 theta1      0.050    0.049 0.012  0.011     0.032    0.071    1.0    3468.    2714.
3 theta2      0.023    0.023 0.0050 0.0050    0.015    0.032    1.0    4000.    3061.
4 ve          0.51     0.54  0.17   0.15      0.21     0.73     1.0    3572.    2990.
```

Vaccine efficacy

```
options(posterior.num_args=list(sigfig=2))
fit_bin2$summary()
```

```
  variable     mean   median     sd    mad       q5      q95  rhat ess_bulk ess_tail
  <chr>       <dbl>    <dbl>  <dbl>  <dbl>    <dbl>    <dbl> <dbl>    <dbl>    <dbl>
1 lp__     -164.    -163.     1.0    0.71   -166.    -163.       1.0    2014.    2483.
2 theta1      0.050    0.049 0.012  0.011     0.032    0.071    1.0    3468.    2714.
3 theta2      0.023    0.023 0.0050 0.0050    0.015    0.032    1.0    4000.    3061.
4 ve          0.51     0.54  0.17   0.15      0.21     0.73     1.0    3572.    2990.
```

- `lp__` is the log density, ie, same as `target`

HMC specific diagnostics

```
fit_bin2$diagnostic_summary(diagnostics = c("divergences",
                                            "treedepth"))
```

```
$num_divergent
[1] 0 0 0 0

$num_max_treedepth
[1] 0 0 0 0
```

HMC specific diagnostics

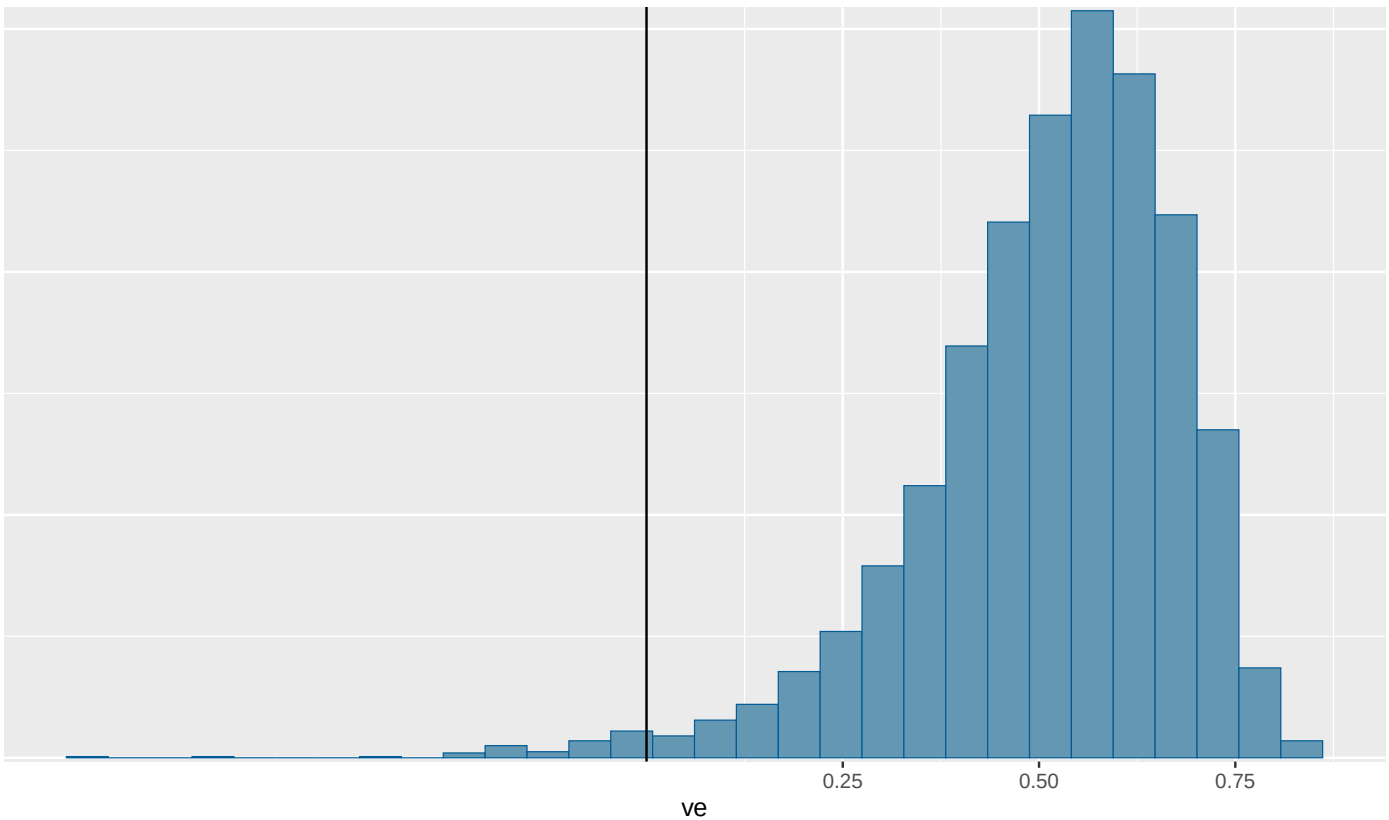
```
fit_bin2$diagnostic_summary(diagnostics = c("divergences",
                                            "treedepth"))
```

```
$num_divergent
[1] 0 0 0 0

$num_max_treedepth
[1] 0 0 0 0
```

`diagnostic_summary()` includes E-BFMI diagnostic, which I'll skip in this course

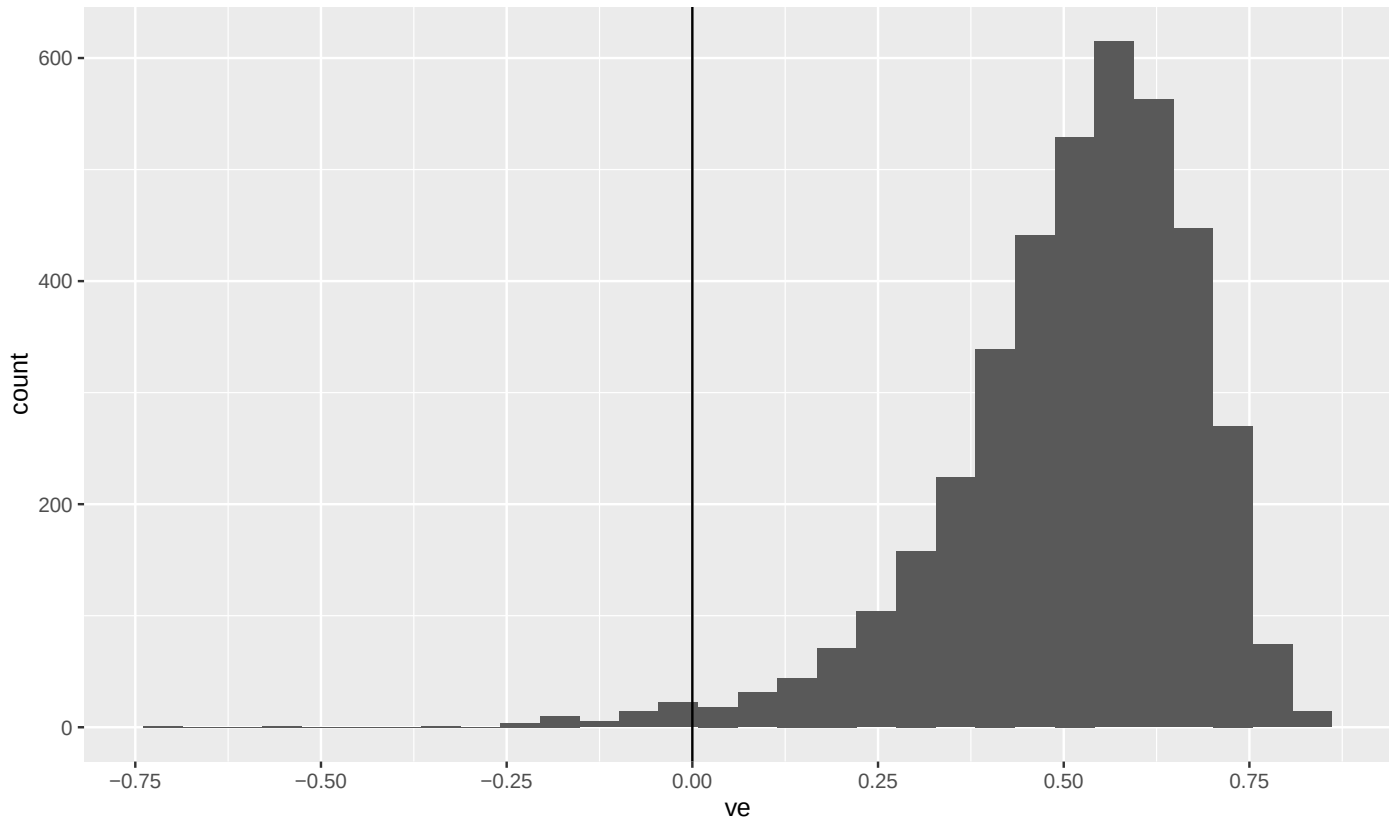
Vaccine efficacy (bayesplot)

```
draws <- fit_bin2$draws(format = "df")
mcmc_hist(draws, pars = 've') +
  geom_vline(xintercept = 0) +
  scale_x_continuous(breaks = c(seq(-0.75,1,by=0.25)))
```



Vaccine efficacy (ggplot2)

```
draws <- fit_bin2$draws(format = "df")
draws |> ggplot(aes(x=ve)) +
  geom_histogram() +
  geom_vline(xintercept = 0) +
  scale_x_continuous(breaks = c(seq(-0.75,1,by=0.25)))
```



Vaccine efficacy (probability and MCSE)

Probability (and corresponding MCSE) that VE<0

```
> draws |>
    mutate_variables(p_ve_lt_0 =
                       as.numeric(ve<0)) |>
    subset_draws("p_ve_lt_0") |>
    summarise_draws(prob=mean, MCSE=mcse_mean)

  variable   prob   MCSE
  p_ve_lt_0 0.013 0.0021
```

posterior object formats

Default is draws_array

```
> fit_bin2$draws()
# A draws_array: 1000 iterations, 4 chains, and 4 variables
, , variable = lp__

         chain
iteration    1    2    3    4
        1 -163 -164 -164 -163
        2 -163 -164 -165 -163
        3 -163 -166 -163 -163
        4 -164 -165 -164 -163
        5 -163 -167 -164 -164

, , variable = theta1

         chain
iteration     1     2     3     4
        1 0.061 0.048 0.068 0.055
        2 0.037 0.074 0.049 0.047
...
```

posterior object formats

draws_df looks prettier and works with ggplot()

```
> fit_bin2$draws(format ="df")
```

```
# A draws_df: 1000 iterations, 4 chains, and 4 variables
   lp__ theta1 theta2   ve
1  -163  0.061  0.025 0.59
2  -163  0.037  0.024 0.36
3  -163  0.044  0.020 0.55
4  -164  0.073  0.023 0.68
5  -163  0.040  0.022 0.46
6  -163  0.062  0.023 0.62
7  -165  0.072  0.018 0.75
8  -166  0.076  0.017 0.78
9  -165  0.069  0.017 0.76
10 -164  0.035  0.027 0.21
# ... with 3990 more draws
# ... hidden reserved variables {'.chain', '.iteration', '.draw'}
```

posterior object formats

draws_rvar makes it easy to compute derived quantities

```
> as_draws_rvars(fit_bin2$draws())
# A draws_rvars: 1000 iterations, 4 chains, and 4 variables
$lp__: rvar<1000,4>[1] mean ± sd:
[1] -164 ± 1

$theta1: rvar<1000,4>[1] mean ± sd:
[1] 0.05 ± 0.012

$theta2: rvar<1000,4>[1] mean ± sd:
[1] 0.023 ± 0.005

$ve: rvar<1000,4>[1] mean ± sd:
[1] 0.51 ± 0.17
```

posterior object formats

draws_rvar makes it easy to compute derived quantities

```
> as_draws_rvars(fit_bin2$draws())
# A draws_rvars: 1000 iterations, 4 chains, and 4 variables
$lp__: rvar<1000,4>[1] mean ± sd:
[1] -164 ± 1

$theta1: rvar<1000,4>[1] mean ± sd:
[1] 0.05 ± 0.012

$theta2: rvar<1000,4>[1] mean ± sd:
[1] 0.023 ± 0.005

$ve: rvar<1000,4>[1] mean ± sd:
[1] 0.51 ± 0.17
```

```
> with(as_draws_rvars(draws), 1 - theta2 / theta1)
rvar<1000,4>[1] mean ± sd:
[1] 0.51 ± 0.17
```

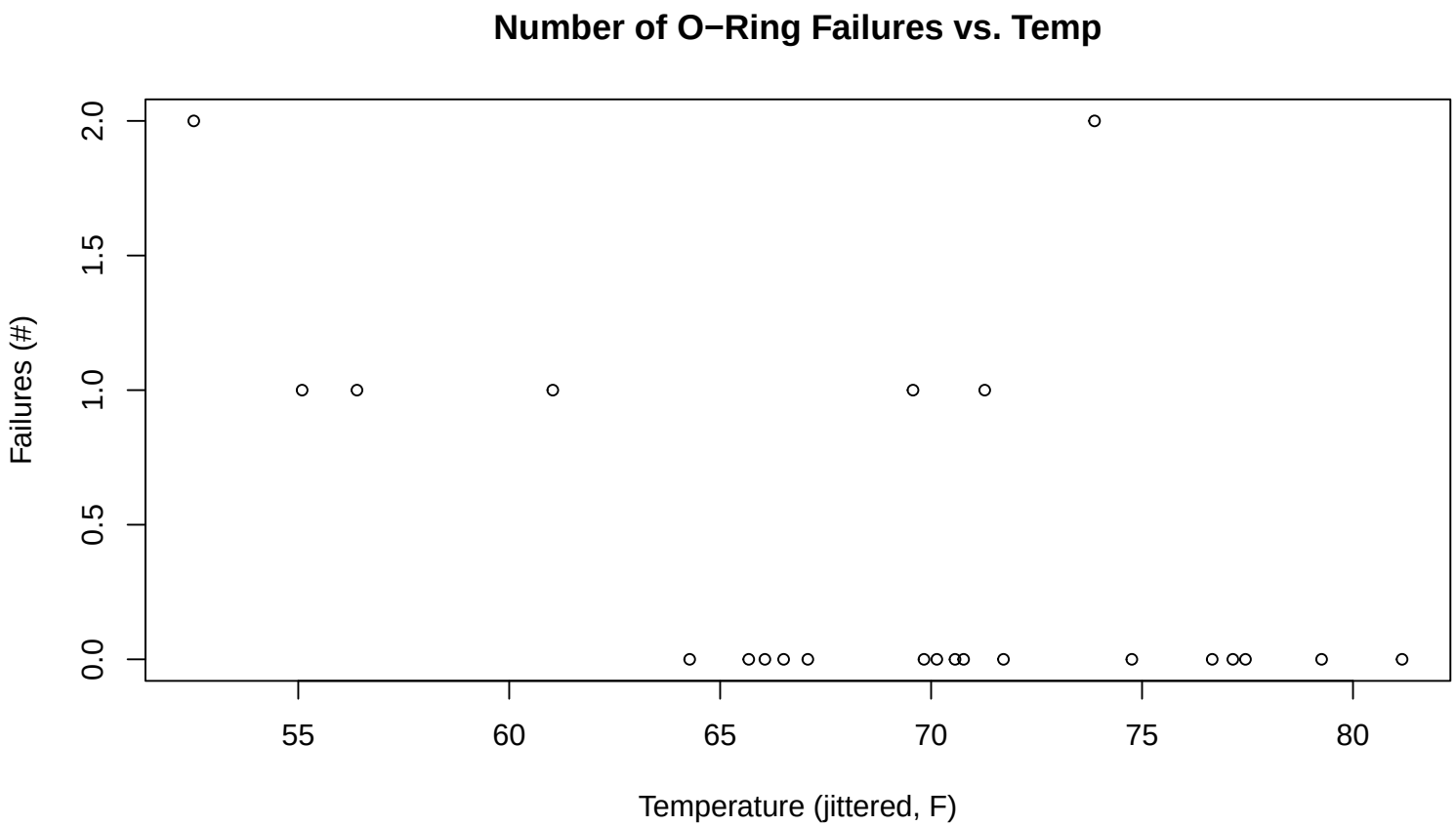
posterior object formats

draws_rvar makes it easy to compute derived quantities

```
> as_draws_rvars(fit_bin2$draws())
# A draws_rvars: 1000 iterations, 4 chains, and 4 variables
$lp__: rvar<1000,4>[1] mean ± sd:
[1] -164 ± 1

$theta1: rvar<1000,4>[1] mean ± sd:
[1] 0.05 ± 0.012

$theta2: rvar<1000,4>[1] mean ± sd:
[1] 0.023 ± 0.005

$ve: rvar<1000,4>[1] mean ± sd:
[1] 0.51 ± 0.17

> with(as_draws_rvars(draws), 1 - theta2 / theta1)
rvar<1000,4>[1] mean ± sd:
[1] 0.51 ± 0.17

> as_draws_rvars(draws)$ve<0
rvar<1000,4>[1] mean ± sd:
[1] 0.013 ± 0.11
```

Challenger dataset

- Number of failed o-rings from 23 Challenger launches vs. temperature
- Does temperature predict failure?



Logistic regression model

```
data {
    int<lower=0> N;          // number of observations
    vector[N] x;
    array[N] int<lower=1> K;
    array[N] int<lower=0> y;
}
parameters {
    real alpha;              // intercept
    real beta;               // slope
}
transformed parameters {
    vector[N] logit_p;
    logit_p = alpha + beta * x;  // logit-linear model
}
model {
    y ~ binomial_logit(K, logit_p); // observation model
}
```

Logistic regression model

```
data {
    int<lower=0> N;         // number of observations
    vector[N] x;
    array[N] int<lower=1> K;
    array[N] int<lower=0> y;
}
parameters {
    real alpha;             // intercept
    real beta;              // slope
}
transformed parameters {
    vector[N] logit_p;
    logit_p = alpha + beta * x;  // logit-linear model
}
model {
    y ~ binomial_logit(K, logit_p); // observation model
}
```

Logistic regression model

```
data {
    int<lower=0> N;         // number of observations
    vector[N] x;
    array[N] int<lower=1> K;
    array[N] int<lower=0> y;
}
parameters {
    real alpha;             // intercept
    real beta;              // slope
}
transformed parameters {
    vector[N] logit_p;
    logit_p = alpha + beta * x;  // logit-linear model
}
model {
    y ~ binomial_logit(K, logit_p); // observation model
}
```

Logistic regression model

```
data {
    int<lower=0> N;         // number of observations
    vector[N] x;
    array[N] int<lower=1> K;
    array[N] int<lower=0> y;
}
parameters {
    real alpha;             // intercept
    real beta;              // slope
}
transformed parameters {
    vector[N] logit_p;
    logit_p = alpha + beta * x;  // logit-linear model
}
model {
    y ~ binomial_logit(K, logit_p); // observation model
}
```

Logistic regression model

```
data {
    int<lower=0> N;          // number of observations
    vector[N] x;
    array[N] int<lower=1> K;
    array[N] int<lower=0> y;
}
```

- difference between `vector[N] x` and `array[N] real x`

Logistic regression model

```
data {
    int<lower=0> N;          // number of observations
    vector[N] x;
    array[N] int<lower=1> K;
    array[N] int<lower=0> y;
}
```

- difference between `vector[N] x` and `array[N] real x`
- only integer arrays: `array[N] int x`

Logistic regression model

```
parameters {
    real alpha;           // intercept
    real beta;            // slope
}
transformed parameters {
    vector[N] logit_p;
    logit_p = alpha + beta*x;  // logit-linear model
}
```

- transformed parameters are deterministic transformations of parameters and data

Priors for logistic regression

```
data {
    int<lower=0> N;          // number of observations
    vector[N] x;
    array[N] int<lower=1> K;
    array[N] int<lower=0> y;
    real pmualpha; // prior mean for alpha
    real psalpha;  // prior std for alpha
    real pmubeta;  // prior mean for beta
    real psbeta;   // prior std for beta
}
...
transformed parameters {
    vector[N] logit_p;
    logit_p = alpha + beta*x;
}
model {
    alpha ~ normal(pmualpha, psalpha); // prior for alpha
    beta ~ normal(pmubeta, psbeta);    // prior for beta
    y ~ binomial_logit(K, logit_p);                // observation mode
}
```

Priors

- Prior for temperature effect?

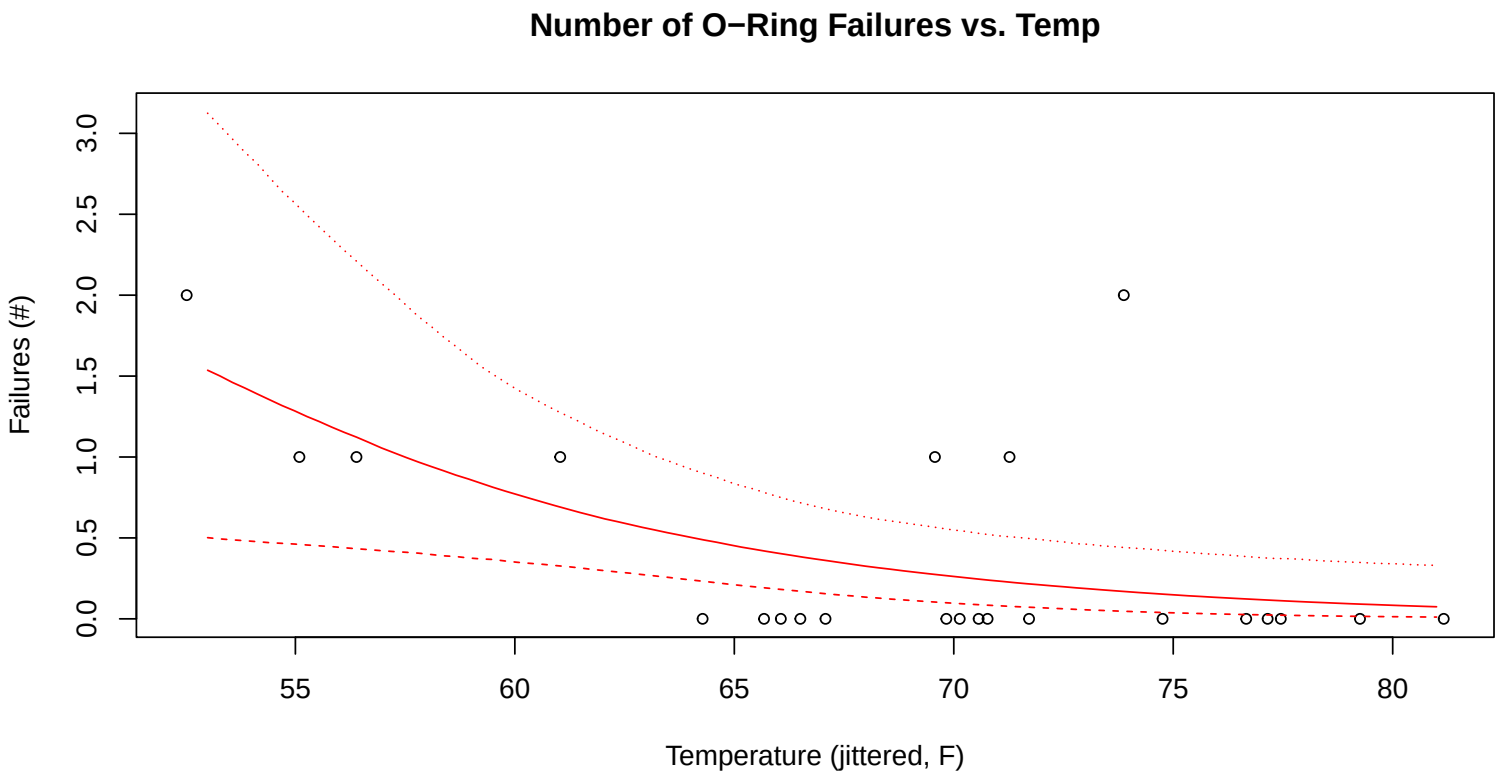
Challenger fit

```
chal <- read.csv("challenger.csv")
stan_chal <-
  list(
    N = nrow(chal),
    y = chal$fail,
    x = chal$temp,
    K = rep(6, nrow(chal))
  )
mod_logistic <- cmdstan_model(stan_file = 'logistic.stan')
fit_logistic <- mod_logistic$sample(data = stan_chal, refresh=1000)
```

```
> Running MCMC with 4 parallel chains...

Chain 1 Iteration:    1 / 2000 [  0%]  (Warmup)
Chain 1 Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain 1 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 1 Iteration: 2000 / 2000 [100%]  (Sampling)
...
All 4 chains finished successfully.
Mean chain execution time: 0.0 seconds.
```

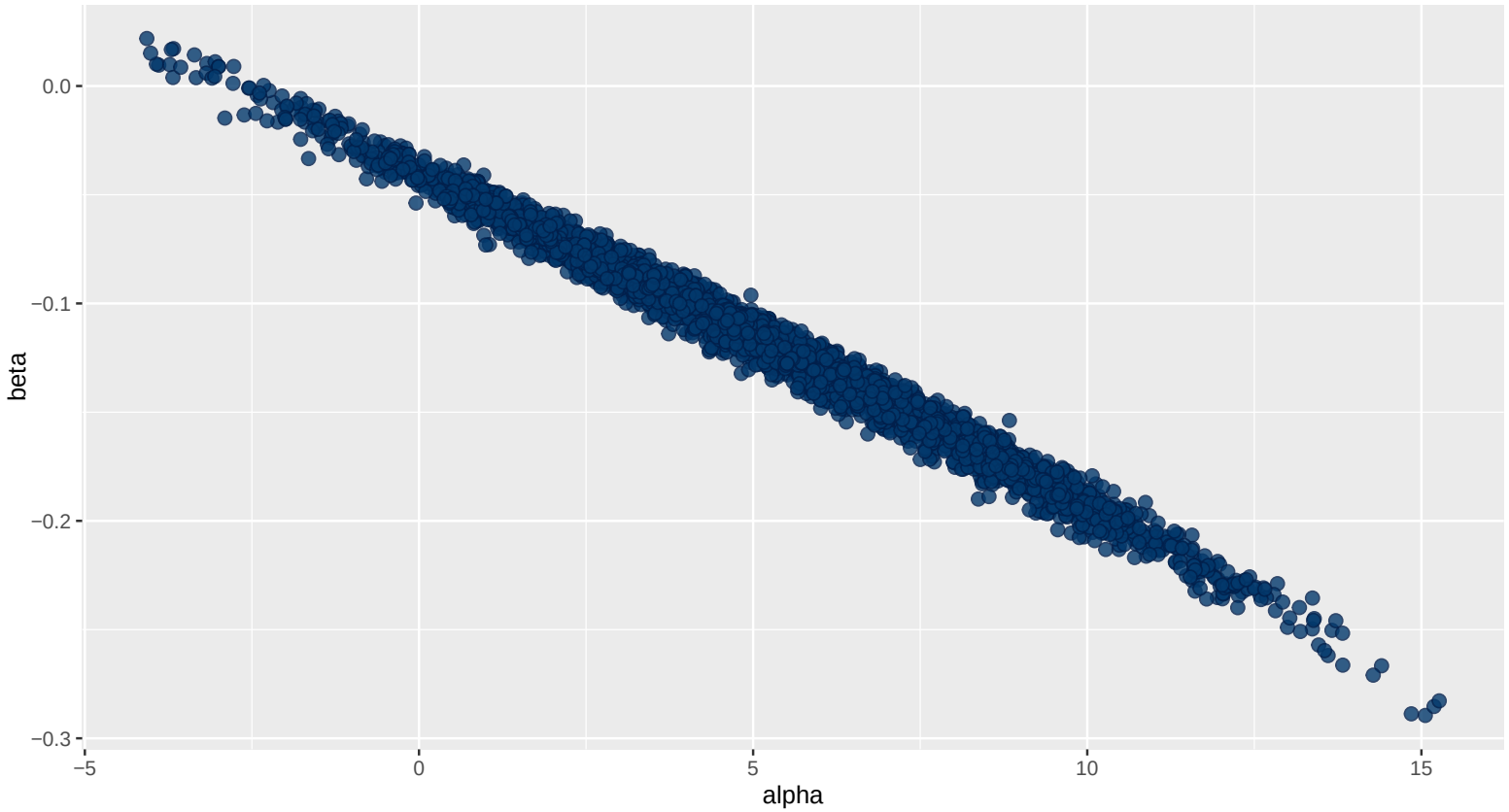
Challenger fit

Posterior fit (with flat priors)



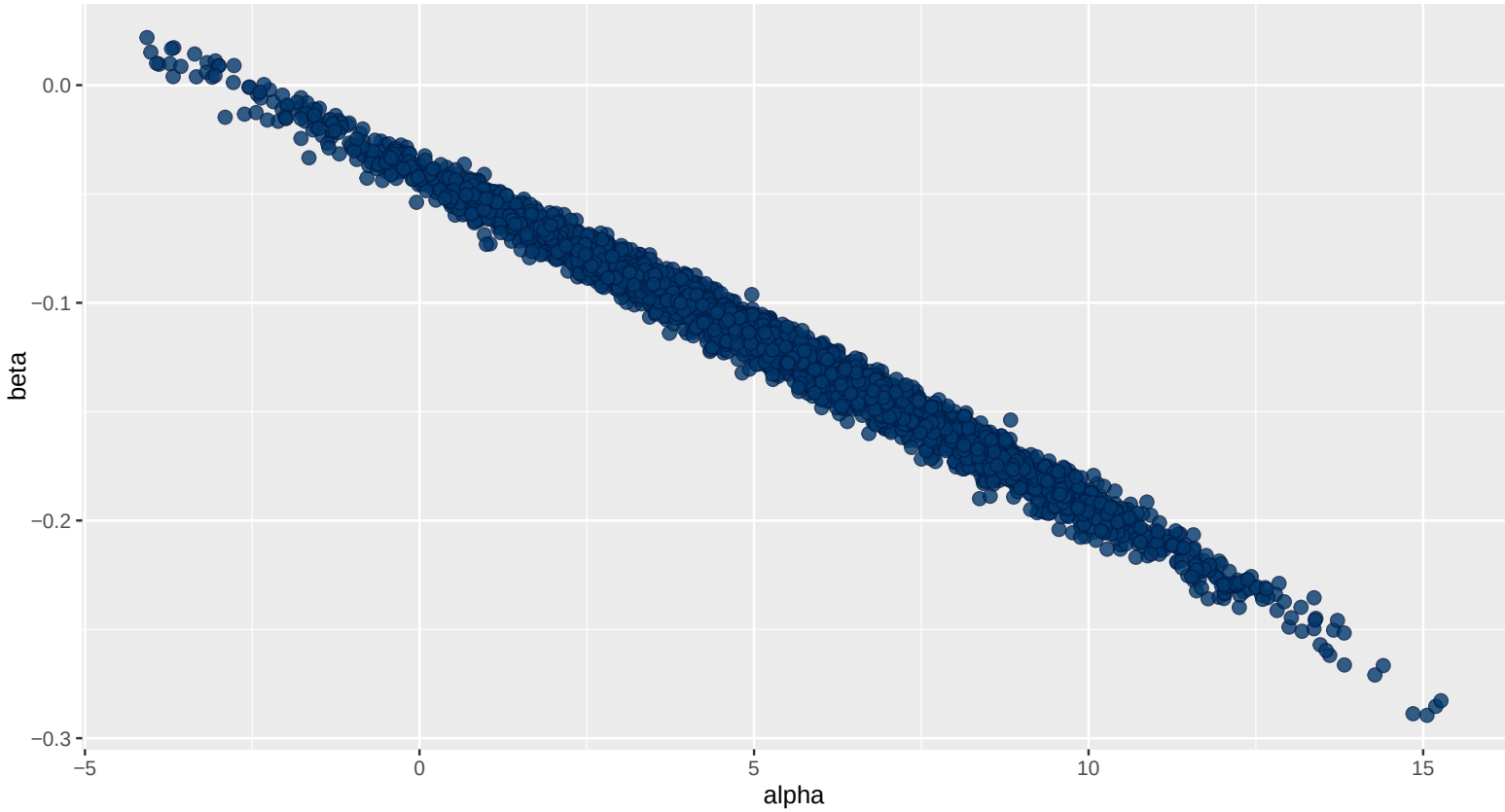
Challenger fit

Posterior draws of alpha and beta



Challenger fit

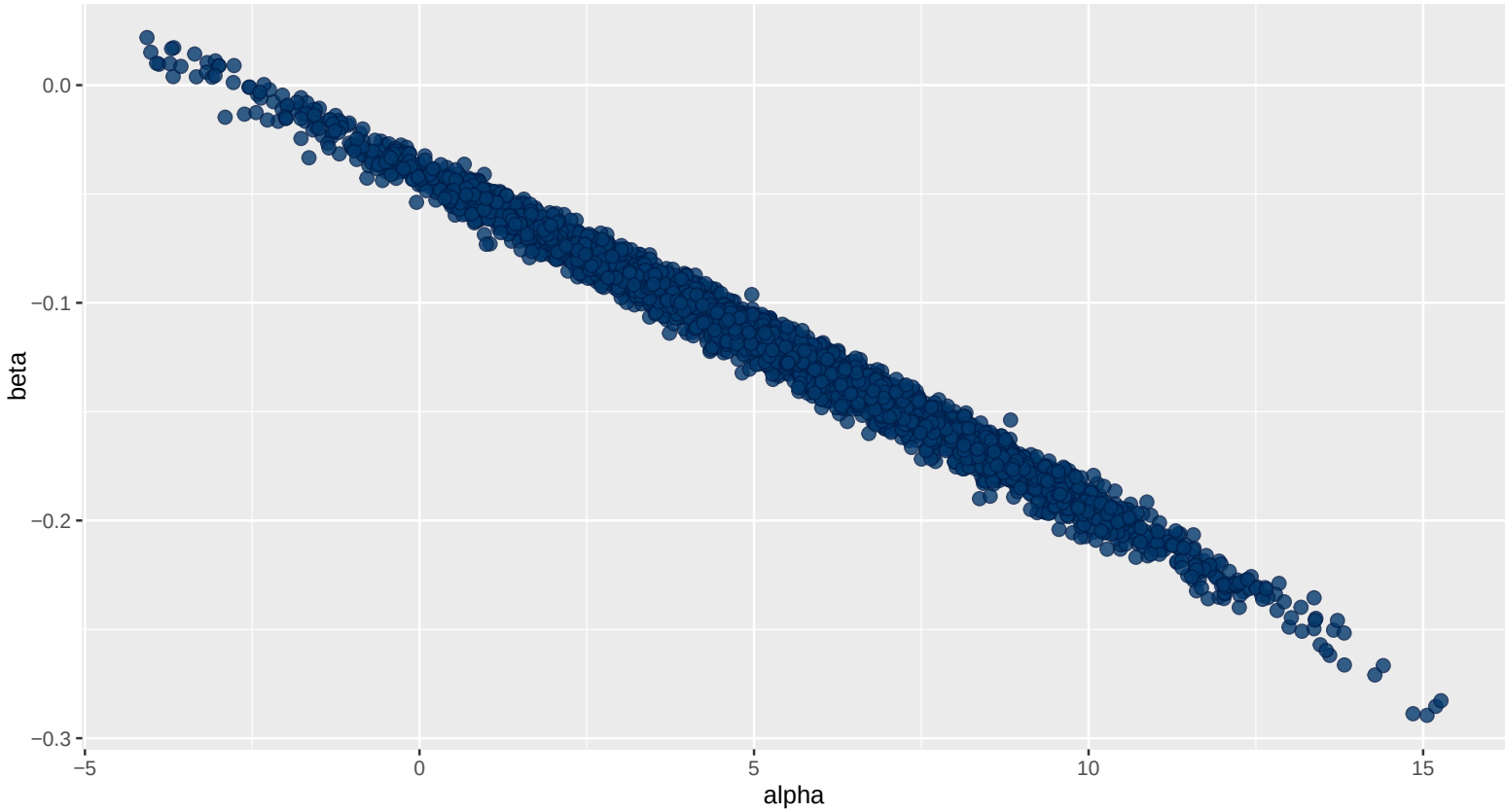
Posterior draws of alpha and beta




```
# A tibble: 3 × 10
  variable   mean median    sd   mad     q5     q95  rhat ess_bulk ess_tail
  <chr>     <dbl>  <dbl> <dbl> <dbl>  <dbl>   <dbl> <dbl>    <dbl>    <dbl>
2 alpha      5.2    5.2  3.1   3.0     0.21  10.      1.0     571.     606.
3 beta      -0.12  -0.12 0.047 0.047  -0.20  -0.042   1.0     572      597
```

Challenger fit

Posterior draws of alpha and beta




```
# A tibble: 3 × 10
  variable   mean median    sd   mad     q5     q95  rhat ess_bulk ess_tail
  <chr>     <dbl>  <dbl> <dbl> <dbl>  <dbl>   <dbl> <dbl>    <dbl>    <dbl>
2 alpha      5.2    5.2  3.1   3.0     0.21  10.      1.0     571.     606.
3 beta      -0.12  -0.12 0.047 0.047  -0.20  -0.042   1.0     572      597
```

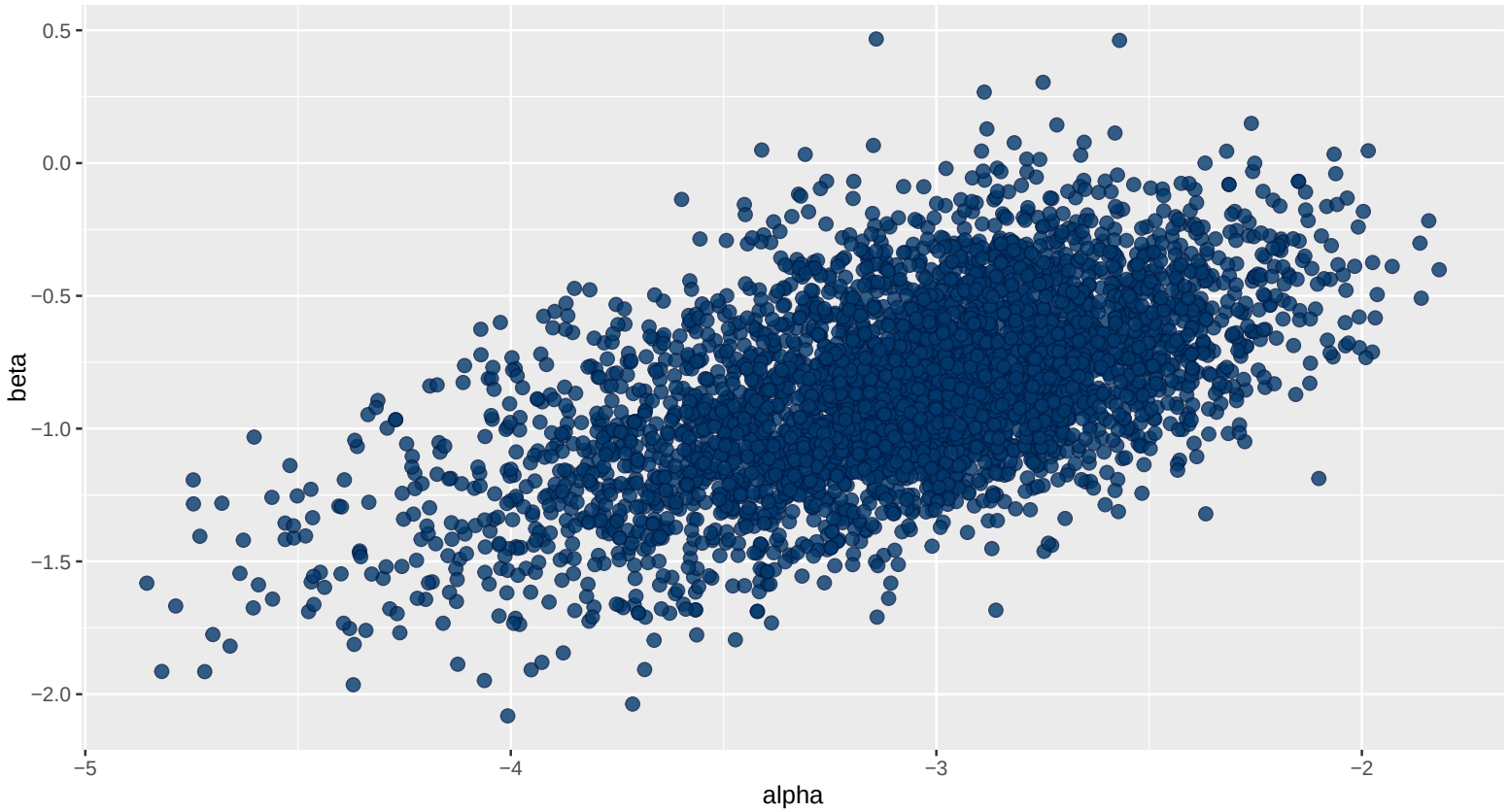
Logistic regression model

Center the data inside the model code

```
data {
  int<lower=0> N;         // number of observations
  vector[N] x;
  array[N] int<lower=1> K;
  array[N] int<lower=0> y;
  real xpred; // covariate value for prediction
}

transformed data {
  vector[N] x_std;
  real xpred_std;
  x_std = (x - mean(x)) / sd(x);
  xpred_std = (xpred - mean(x)) / sd(x);
}

transformed parameters {
    vector[N] logit_p;
    logit_p = alpha + beta * x_std;
}
```

Challenger fit

Posterior draws of alpha and beta when data is centered



Challenger fit

Without centering

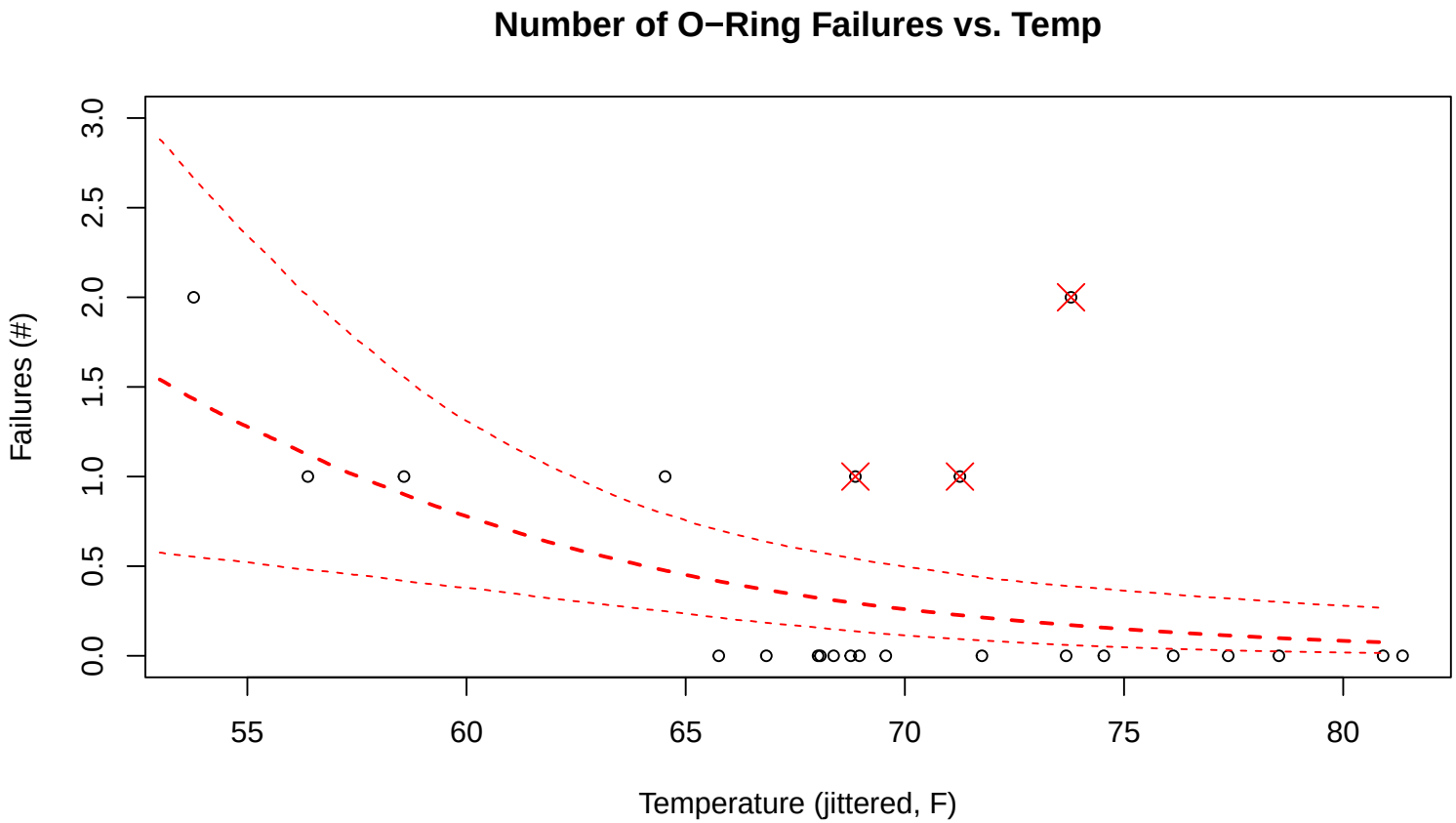
```
> fit$summary(variables=c("alpha","beta"),
                  default_convergence_measures())

 variable  rhat ess_bulk ess_tail
 alpha      1.0     581.     607.
 beta       1.0     587.     650.
```

With centering

```
> fit_std$summary(variables=c("alpha","beta"),
                      default_convergence_measures())

 variable  rhat ess_bulk ess_tail
 alpha      1.0    1660.    1818.
 beta       1.0    1568.    1739.
```

Challenger dataset

- Is the model fit sensitive to these outliers?



Logistic regression as latent variable model

Standard formulation:

$$Y_i \sim \text{Bernoulli}(p_i)$$

$$p_i = \frac{1}{1 + e^{-\beta^T X_i}}$$

Logistic regression as latent variable model

Standard formulation:

$$Y_i \sim \text{Bernoulli}(p_i)$$

$$p_i = \frac{1}{1 + e^{-\beta^T X_i}}$$

Latent variable formulation:

$$Z_i = \beta^T X_i + \epsilon_i$$

Logistic regression as latent variable model

Standard formulation:

$$Y_i \sim \text{Bernoulli}(p_i)$$

$$p_i = \frac{1}{1 + e^{-\beta^T X_i}}$$

Latent variable formulation:

$$Z_i = \beta^T X_i + \epsilon_i$$

$$P(\epsilon_i \leq x) = \frac{1}{1 + e^{-x}} \qquad \text{(Logistic RV)}$$

Logistic regression as latent variable model

Standard formulation:

$$Y_i \sim \text{Bernoulli}(p_i)$$

$$p_i = \frac{1}{1 + e^{-\beta^T X_i}}$$

Latent variable formulation:

$$Z_i = \beta^T X_i + \epsilon_i$$

$$P(\epsilon_i \leq x) = \frac{1}{1 + e^{-x}} \qquad \text{(Logistic RV)}$$

$$Y_i = \mathbb{1}\left(Z_i > 0\right)$$

Logistic regression as latent variable model

Standard formulation:

$$Y_i \sim \text{Bernoulli}(p_i)$$

$$p_i = \frac{1}{1 + e^{-\beta^T X_i}}$$

Latent variable formulation:

$$Z_i = \beta^T X_i + \epsilon_i$$

$$P(\epsilon_i \leq x) = \frac{1}{1 + e^{-x}} \qquad \text{(Logistic RV)}$$

$$Y_i = \mathbb{1}\left(Z_i > 0\right)$$

$$P(Y_i = 1) = P(\epsilon_i > -\beta^T X_i)$$

Logistic regression as latent variable model

Standard formulation:

$$Y_i \sim \text{Bernoulli}(p_i)$$

$$p_i = \frac{1}{1 + e^{-\beta^T X_i}}$$

Latent variable formulation:

$$Z_i = \beta^T X_i + \epsilon_i$$

$$P(\epsilon_i \leq x) = \frac{1}{1 + e^{-x}} \qquad \text{(Logistic RV)}$$

$$Y_i = \mathbb{1}\left(Z_i > 0\right)$$

$$P(Y_i = 1) = P(\epsilon_i > -\beta^T X_i)$$

$$= 1 - \frac{1}{1 + e^{\beta^T X_i}}$$

Logistic regression as latent variable model

Standard formulation:

$$Y_i \sim \text{Bernoulli}(p_i)$$

$$p_i = \frac{1}{1 + e^{-\beta^T X_i}}$$

Latent variable formulation:

$$Z_i = \beta^T X_i + \epsilon_i$$

$$P(\epsilon_i \leq x) = \frac{1}{1 + e^{-x}} \qquad \text{(Logistic RV)}$$

$$Y_i = \mathbb{1}\left(Z_i > 0\right)$$

$$\begin{aligned} P(Y_i = 1) &= P(\epsilon_i > -\beta^T X_i) \\ &= 1 - \frac{1}{1 + e^{\beta^T X_i}} \\ &= \frac{e^{\beta^T X_i}}{1 + e^{\beta^T X_i}} \end{aligned}$$

Logistic regression as latent variable model

Standard formulation:

$$Y_i \sim \text{Bernoulli}(p_i)$$

$$p_i = \frac{1}{1 + e^{-\beta^T X_i}}$$

Latent variable formulation:

$$Z_i = \beta^T X_i + \epsilon_i$$

$$P(\epsilon_i \leq x) = \frac{1}{1 + e^{-x}} \qquad \text{(Logistic RV)}$$

$$Y_i = \mathbb{1}\left(Z_i > 0\right)$$

$$\begin{aligned} P(Y_i = 1) &= P(\epsilon_i > -\beta^T X_i) \\ &= 1 - \frac{1}{1 + e^{\beta^T X_i}} \\ &= \frac{1}{1 + e^{-\beta^T X_i}} \end{aligned}$$

Robust binary regression

Latent variable formulation:

$$Z_i = \beta^T X_i + \epsilon_i$$

Robust binary regression

Latent variable formulation:

$$Z_i = \beta^T X_i + \epsilon_i$$
$$Y_i = \mathbb{1}\left(Z_i > 0\right)$$

Robust binary regression

Latent variable formulation:

$$Z_i = \beta^T X_i + \epsilon_i$$
$$Y_i = \mathbb{1}\left(Z_i > 0\right)$$
$$\epsilon_i \overset{\text{iid}}{\sim} t_\nu(0, 1)$$

Robust binary regression

Latent variable formulation:

$$
\begin{aligned}
Z_i &= \beta^T X_i + \epsilon_i \\
Y_i &= \mathbb{1}\left(Z_i > 0\right) \\
\epsilon_i &\overset{\text{iid}}{\sim} t_\nu(0, 1) \\
P(Y_i = 1) &= F_{\epsilon_i}(\beta^T X_i)
\end{aligned}
$$

Robust binary regression

Latent variable formulation:

$$
\begin{aligned}
Z_i &= \beta^T X_i + \epsilon_i \\
Y_i &= \mathbb{1}\left(Z_i > 0\right) \\
\epsilon_i &\overset{\text{iid}}{\sim} t_\nu(0, 1) \\
P(Y_i = 1) &= F_{\epsilon_i}(\beta^T X_i)
\end{aligned}
$$

Robust binary regression

Latent variable formulation:

$$
\begin{aligned}
Z_i &= \beta^T X_i + \epsilon_i \\
Y_i &= \mathbb{1}\left(Z_i > 0\right) \\
\epsilon_i &\overset{\text{iid}}{\sim} t_\nu(0, 1) \\
P(Y_i = 1) &= F_{\epsilon_i}(\beta^T X_i)
\end{aligned}
$$

- $\nu = 1$: $\epsilon_i \sim \text{Cauchy}(0, 1)$

Robust binary regression

Latent variable formulation:

$$
\begin{aligned}
Z_i &= \beta^T X_i + \epsilon_i \\
Y_i &= \mathbb{1}\left(Z_i > 0\right) \\
\epsilon_i &\overset{\text{iid}}{\sim} t_\nu(0, 1) \\
P(Y_i = 1) &= F_{\epsilon_i}(\beta^T X_i)
\end{aligned}
$$

- $\nu = 1$: $\epsilon_i \sim \text{Cauchy}(0, 1)$
- $\nu \to \infty$: $\epsilon_i \sim \text{Normal}(0, 1)$

Robust binary regression

Latent variable formulation:

$$
\begin{aligned}
Z_i &= \beta^T X_i + \epsilon_i \\
Y_i &= \mathbb{1}\left(Z_i > 0\right) \\
\epsilon_i &\overset{\text{iid}}{\sim} t_\nu(0, 1) \\
P(Y_i = 1) &= F_{\epsilon_i}(\beta^T X_i)
\end{aligned}
$$

- $\nu = 1$: $\epsilon_i \sim \text{Cauchy}(0, 1)$
- $\nu \to \infty$: $\epsilon_i \sim \text{Normal}(0, 1)$
- $\nu = 9$: $F_{\epsilon_i}(s(\nu)x) \approx \frac{1}{1+e^{-x}}$

Prior for ν

- Limit $\nu \in [1, \infty)$

Prior for ν

- Limit $\nu \in [1, \infty)$
- If we reparameterize to $\eta = 1/\nu$, $\eta \in (0, 1]$

Prior for ν

- Limit $\nu \in [1, \infty)$
- If we reparameterize to $\eta = 1/\nu$, $\eta \in (0, 1]$
- A uniform prior on η implies a priori $P(\nu \in [1, 2]) = 0.5$

Robit model

```
...
parameters {
  real alpha;
  real beta;
  real<lower=0, upper=1> inv_nu;
}
transformed parameters {
  vector[N] p;
  {
    vector[N] q_p = alpha + beta * x_std;
    for (n in 1:N)
      p[n] = student_t_cdf(q_p[n] | 1 / inv_nu, 0, 1);
  }
}
model {
  // Omitting a prior implies that inv_nu ~ Unif(0,1)
  y ~ binomial(K, p);  // observation model
}
```

Robit model

```
transformed parameters {
  vector[N] p;
  {
    vector[N] q_p = alpha + beta * x_std;
    for (n in 1:N)
      p[n] = student_t_cdf(q_p[n] | 1 / inv_nu, 0, 1);
  }
}
```

- Variables declared in will be local to that block

Robit model

```
transformed parameters {
  vector[N] p;
  {
    vector[N] q_p = alpha + beta * x_std;
    for (n in 1:N)
      p[n] = student_t_cdf(q_p[n] | 1 / inv_nu, 0, 1);
  }
}
```

- Variables declared in will be local to that block
- Thus draws of q_p aren't saved

Challenger fit

```
mod_robit <- cmdstan_model(stan_file = 'robit.stan')
fit_robit <- mod_robit$sample(data = stan_chal, refresh=1000)
```

```
> Running MCMC with 4 parallel chains...

Chain 1 Iteration:    1 / 2000 [  0%]  (Warmup)
Chain 1 Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain 1 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 1 Iteration: 2000 / 2000 [100%]  (Sampling)
...
All 4 chains finished successfully.
Mean chain execution time: 0.7 seconds.
Total execution time: 0.9 seconds.

Warning: 2 of 4000 (0.0%) transitions ended with a divergence.
See https://mc-stan.org/misc/warnings for details.
```

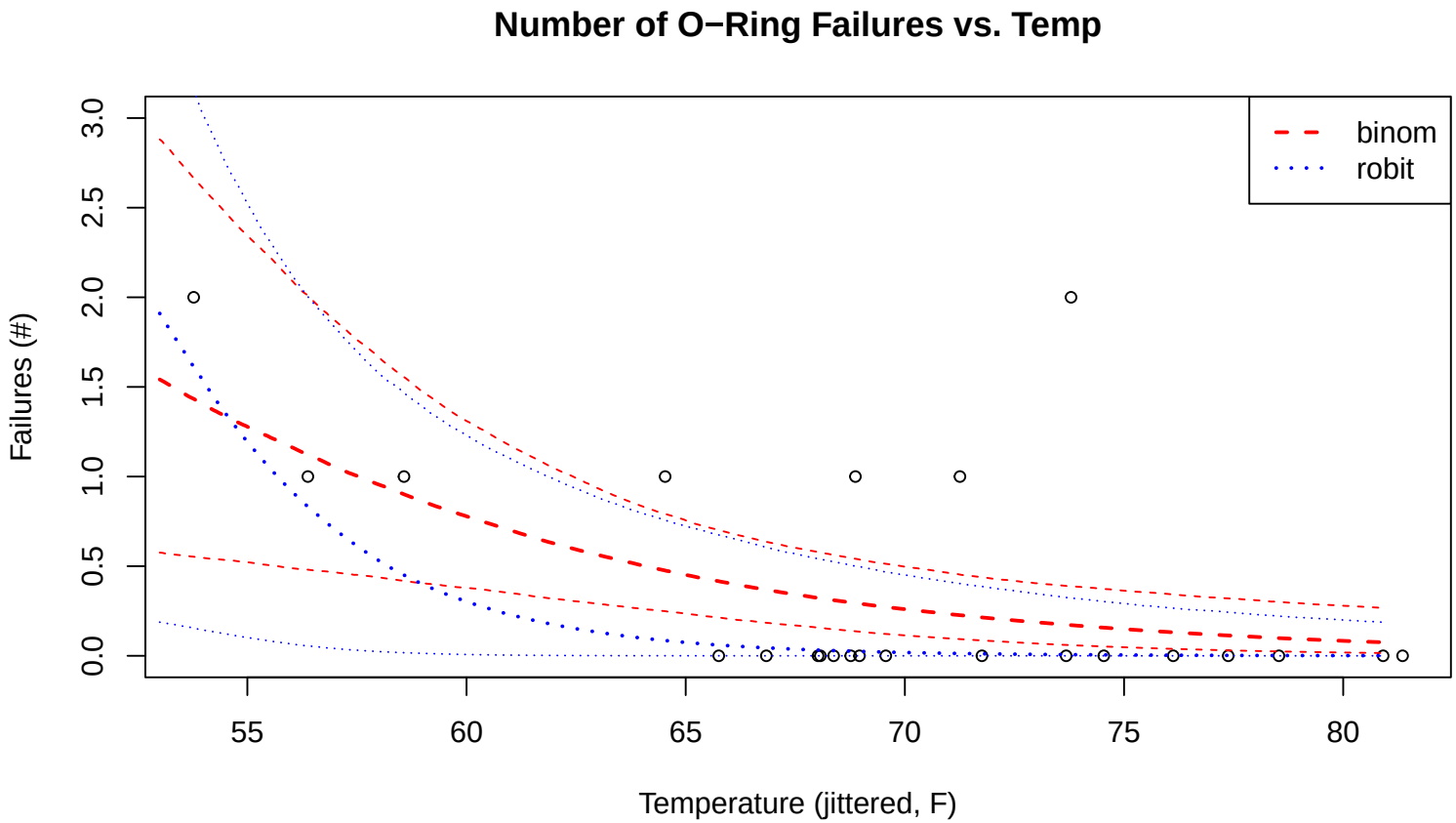
Adapt delta

```
fit_robit <- mod_robit$sample(data = stan_chal, refresh=1000,
                              adapt_delta = 0.9)
```

```
> Running MCMC with 4 parallel chains...

Chain 1 Iteration:    1 / 2000 [  0%]  (Warmup)
Chain 1 Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain 1 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 1 Iteration: 2000 / 2000 [100%]  (Sampling)
...
All 4 chains finished successfully.
Mean chain execution time: 0.8 seconds.
Total execution time: 1.1 seconds.
```

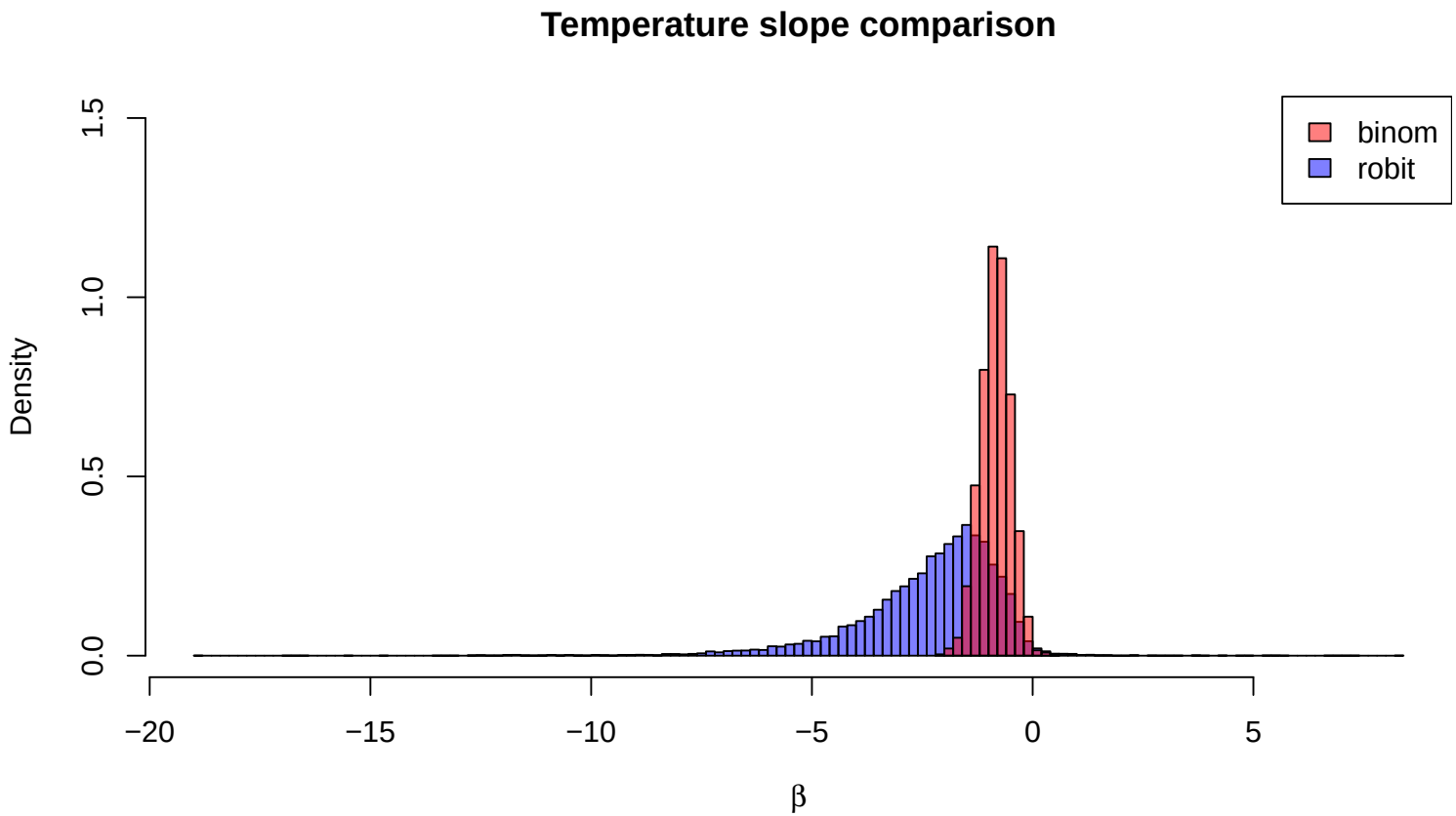
Challenger dataset

- Inferences are sensitive to the outliers



Temperature coefficient comparison

- Inferences are sensitive to the outliers



From binomial to beta-binomial regression

Binomial model for failures assumes independence, which may not hold.

$$X_{ij} \mid p_i \overset{\text{iid}}{\sim} \text{Bernoulli}(p_i),\ j = 1, \ldots, 6$$
$$\text{logit}(p_i) = \alpha + \beta \text{temp}_i$$

From binomial to beta-binomial regression

Binomial model for failures assumes independence, which may not hold.

$$X_{ij} \mid p_i \overset{\text{iid}}{\sim} \text{Bernoulli}(p_i),\ j = 1, \ldots, 6$$
$$\text{logit}(p_i) = \alpha + \beta \text{temp}_i$$

Incorporating dependence among failures j for a given launch i

$$X_{ij} \mid p_i \overset{\text{iid}}{\sim} \text{Bernoulli}(p_i),\ j = 1, \ldots, 6$$

From binomial to beta-binomial regression

Binomial model for failures assumes independence, which may not hold.

$$X_{ij} \mid p_i \overset{\text{iid}}{\sim} \text{Bernoulli}(p_i),\ j = 1, \ldots, 6$$
$$\text{logit}(p_i) = \alpha + \beta \text{temp}_i$$

Incorporating dependence among failures j for a given launch i

$$X_{ij} \mid p_i \overset{\text{iid}}{\sim} \text{Bernoulli}(p_i),\ j = 1, \ldots, 6$$
$$Y_i \mid p_i \sim \text{Binomial}(6, p_i)$$

From binomial to beta-binomial regression

Binomial model for failures assumes independence, which may not hold.

$$X_{ij} \mid p_i \overset{\text{iid}}{\sim} \text{Bernoulli}(p_i),\ j = 1, \ldots, 6$$
$$\text{logit}(p_i) = \alpha + \beta \text{temp}_i$$

Incorporating dependence among failures j for a given launch i

$$X_{ij} \mid p_i \overset{\text{iid}}{\sim} \text{Bernoulli}(p_i),\ j = 1, \ldots, 6$$
$$Y_i \mid p_i \sim \text{Binomial}(6, p_i)$$
$$p_i \sim \text{Beta}(\mu_i, \kappa)$$

From binomial to beta-binomial regression

Binomial model for failures assumes independence, which may not hold.

$$X_{ij} \mid p_i \overset{\text{iid}}{\sim} \text{Bernoulli}(p_i),\ j = 1, \ldots, 6$$
$$\text{logit}(p_i) = \alpha + \beta \text{temp}_i$$

Incorporating dependence among failures j for a given launch i

$$X_{ij} \mid p_i \overset{\text{iid}}{\sim} \text{Bernoulli}(p_i),\ j = 1, \ldots, 6$$
$$Y_i \mid p_i \sim \text{Binomial}(6, p_i)$$
$$p_i \sim \text{Beta}(\mu_i, \kappa)$$
$$\text{logit}(\mu_i) = \alpha + \beta \text{temp}_i$$

From binomial to beta-binomial regression

Binomial model for failures assumes independence, which may not hold.

$$
\begin{aligned}
X_{ij} \mid p_i &\overset{\text{iid}}{\sim} \text{Bernoulli}(p_i),\ j = 1, \ldots, 6 \\
\text{logit}(p_i) &= \alpha + \beta \text{temp}_i
\end{aligned}
$$

Incorporating dependence among failures j for a given launch i

$$
\begin{aligned}
X_{ij} \mid p_i &\overset{\text{iid}}{\sim} \text{Bernoulli}(p_i),\ j = 1, \ldots, 6 \\
Y_i \mid p_i &\sim \text{Binomial}(6, p_i) \\
p_i &\sim \text{Beta}(\mu_i, \kappa) \\
\text{logit}(\mu_i) &= \alpha + \beta \text{temp}_i \\
\mathbb{E}[Y_i] = \mu_i, \mathbb{V}[Y_i] &= n\mu_i(1 - \mu_i)\frac{\kappa + n}{\kappa + 1}
\end{aligned}
$$

Beta-binomial model

```
data {
  int<lower=0> N;
  array[N] int K;
  array[N] int<lower=0, upper=max(K)> y;
  vector[N] x;
  int<lower=0> N_pred_grid;
}
transformed data {
  vector[N] x_std = (x - mean(x)) / sd(x);
  real x_pred_std = (x_pred - mean(x)) / sd(x);
}
parameters {
  real alpha, beta;
  real<lower=0> kappa;
}
transformed parameters {
  vector[N] mu = inv_logit(alpha + beta * x_std);
}
model {
  alpha ~ normal(pmualpha, psalpha);
  beta ~ normal(pmubeta, psbeta);
  for (n in 1:N)
    y[n] ~ beta_binomial_proportion(K[n], mu[n], kappa);     // user defined distrib
}
generated quantities {
  array[N] y_rep = beta_binomial_proportion_rng(K, mu, kappa);// user defined rng
}
```

Beta-binomial model

```
data {
  int<lower=0> N;
  array[N] int K;
  array[N] int<lower=0, upper=max(K)> y;
  vector[N] x;
  int<lower=0> N_pred_grid;
}
transformed data {
  vector[N] x_std = (x - mean(x)) / sd(x);
  real x_pred_std = (x_pred - mean(x)) / sd(x);
}
parameters {
  real alpha, beta;
  real<lower=0> kappa;
}
transformed parameters {
  vector[N] mu = inv_logit(alpha + beta * x_std);
}
model {
  alpha ~ normal(pmualpha, psalpha);
  beta ~ normal(pmubeta, psbeta);
  for (n in 1:N)
    y[n] ~ beta_binomial_proportion(K[n], mu[n], kappa);     // user defined distrib
}
generated quantities {
  array[N] y_rep = beta_binomial_proportion_rng(K, mu, kappa);// user defined rng
}
```

Beta-binomial model

```
data {
  int<lower=0> N;
  array[N] int K;
  array[N] int<lower=0, upper=max(K)> y;
  vector[N] x;
  int<lower=0> N_pred_grid;
}
transformed data {
  vector[N] x_std = (x - mean(x)) / sd(x);
  real x_pred_std = (x_pred - mean(x)) / sd(x);
}
parameters {
  real alpha, beta;
  real<lower=0> kappa;
}
transformed parameters {
  vector[N] mu = inv_logit(alpha + beta * x_std);
}
model {
  alpha ~ normal(pmualpha, psalpha);
  beta ~ normal(pmubeta, psbeta);
  for (n in 1:N)
    y[n] ~ beta_binomial_proportion(K[n], mu[n], kappa);     // user defined distrib
}
generated quantities {
  array[N] y_rep = beta_binomial_proportion_rng(K, mu, kappa);// user defined rng
}
```

Beta-binomial model

```
data {
  int<lower=0> N;
  array[N] int K;
  array[N] int<lower=0, upper=max(K)> y;
  vector[N] x;
  int<lower=0> N_pred_grid;
}
transformed data {
  vector[N] x_std = (x - mean(x)) / sd(x);
  real x_pred_std = (x_pred - mean(x)) / sd(x);
}
parameters {
  real alpha, beta;
  real<lower=0> kappa;
}
transformed parameters {
  vector[N] mu = inv_logit(alpha + beta * x_std);
}
model {
  alpha ~ normal(pmualpha, psalpha);
  beta ~ normal(pmubeta, psbeta);
  for (n in 1:N)
    y[n] ~ beta_binomial_proportion(K[n], mu[n], kappa);     // user defined distrib
}
generated quantities {
  array[N] y_rep = beta_binomial_proportion_rng(K, mu, kappa);// user defined rng
}
```

User defined functions

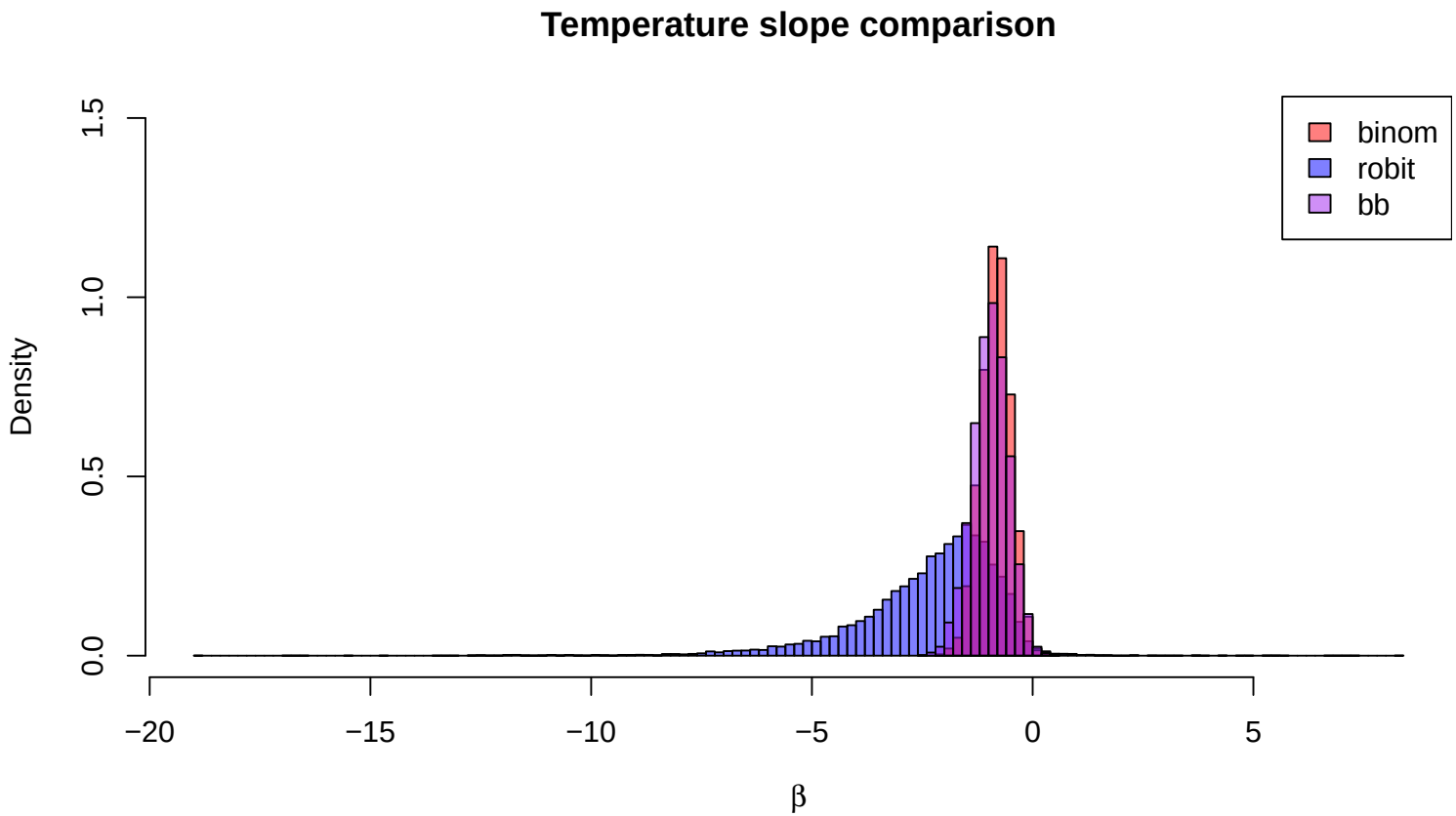
```
functions {
  real beta_binomial_proportion_lpmf(int y, int K,
                                     real mu, real kappa) {
    real alpha = mu * kappa;
    real beta = (1 - mu) * kappa;
    return beta_binomial_lpmf(y | K, alpha, beta);
  }
  array[] int beta_binomial_proportion_rng(array[] int K,
                                           vector mu,
                                           real kappa) {
    int N = size(K);
    vector[N] alphas = mu * kappa;
    vector[N] betas = (1 - mu) * kappa;
    return beta_binomial_rng(K, alphas, betas);
  }
}
```

User defined functions

```
functions {
  real beta_binomial_proportion_lpmf(int y, int K,
                                     real mu, real kappa) {
    real alpha = mu * kappa;
    real beta = (1 - mu) * kappa;
    return beta_binomial_lpmf(y | K, alpha, beta);
  }
  array[] int beta_binomial_proportion_rng(array[] int K,
                                           vector mu,
                                           real kappa) {
    int N = size(K);
    vector[N] alphas = mu * kappa;
    vector[N] betas = (1 - mu) * kappa;
    return beta_binomial_rng(K, alphas, betas);
  }
}
```

Temperature coefficient comparison

- Inferences are sensitive to the outliers

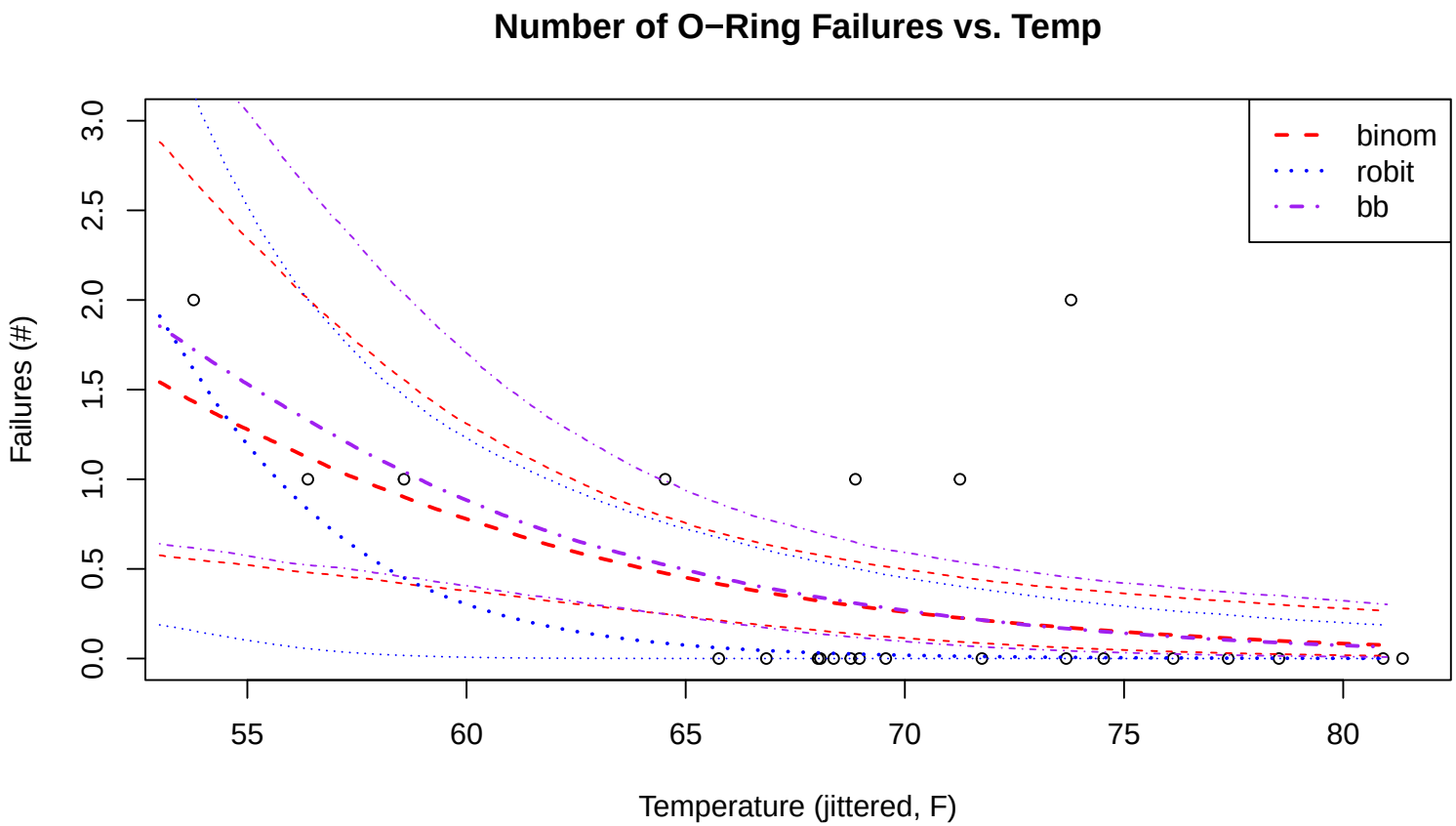


Predictive mean function

```
generated quantities {
    ...
    vector[N_pred_grid] fail_pred;
  {
    vector[N_pred_grid] x_grid_pred =
         (linspaced_vector(N_pred_grid, 53, 81) - mean(x))/sd(x);
    fail_pred = inv_logit(alpha + beta * x_grid_pred) * 6;
  }
}
```

Challenger dataset

- These lines are posterior pointwise quantiles for `fail_pred` from all three models



RStanARM

- `RStanARM` provides simplified model description with pre-compiled models
  - no need to wait for compilation
  - a restricted set of models

Two group Binomial model:

```
d_bin2 <- data.frame(N = c(338, 867), y = c(16,19), grp2 = c(0,1))
fit_bin2 <- stan_glm(y/N ~ grp2,
                     weights = N,
                     family = binomial(),
                     data = d_bin2)
```

RStanARM

- RStanARM provides simplified model description with pre-compiled models
  - no need to wait for compilation
  - a restricted set of models

Two group Binomial model:

```
d_bin2 <- data.frame(N = c(338, 867), y = c(16,19), grp2 = c(0,1))
fit_bin2 <- stan_glm(y/N ~ grp2,
                     weights = N,
                     family = binomial(),
                     data = d_bin2)
```

Normal linear model

```
fit_lin <- stan_glm(fail ~ temp,
                    data = chl)
```

brms

- `brms` provides simplified model description
  - \+ a larger set of models than `RStanARM`, but still restricted
  - \- need to wait for the compilation

```
fit_bin2 <- brm(y | trials(N) ~ grp2,
                family = binomial(),
                data = d_bin2)

fit_lin_t <- brm(fail ~ temp,
                 family = student(),
                 data = d_lin)
```

Different interfaces

- `CmdStanR` / `CmdStanPy`
  - Interface on top of command-line program CmdStan
- `RStan` / `PyStan`
  - C++ functions of Stan are called directly from R / Python
  - Higher integration between R/Python and Stan, but maybe more difficult to install due to more requirements of compatible C++ compilers and libraries

Other packages

- R
  - `posterior` — posterior handling and diagnostics (Lectures 5 and 6)
  - `bayesplot` — visualization and model checking (Lectures 5, 6, and 8)
  - `tidybayes` and `ggdist` — more posterior and prediction visualization (Lecture 6)
  - `marginaleffects` — prediction and comparison visualization
  - `loo` — cross-validation model assessment and comparison (Lecture 9)
  - `projpred` — projection predictive variable selection (Lecture 12)
  - `priorsense` — prior and likelihood sensitivity diagnostics (Lecture 12)

- Python
  - `ArviZ` — visualization, and model checking and assessment