Lecture: Applied Bayesian Statistics II

Overview

What is brms?

The brms package provides an interface to fit Bayesian generalized (non-)linear multivariate multilevel models using Stan. The formula syntax is very similar to that of the package lme4 to provide a familiar and simple interface for performing regression analyses.

paul-buerkner.github.io/brms

It was created and is being maintained by Paul Bürkner. It is extensively documented on its own website.

Comparison: Pre-implemented model types

	brms	rstanarm	rethinking
Supported model types:			
Linear models	yes	yes	yes
Robust linear models	yes	yes[1]	no
Binomial models	yes	yes	yes
Categorical models	yes	no	no
Multinomial models	no	no	no
Count data models	yes	yes	yes
Survival models	yes[2]	yes	yes
Ordinal models	various	cumulative[3]	no
Zero-inflated and hurdle models	yes	no	no
Generalized additive models	yes	yes	no
Non-linear models	yes	no	limited[4]
Additional modeling options:			
Variable link functions	various	various	various
Weights	yes	yes	no
Offset	yes	yes	yes
Multivariate responses	limited	no	no
Autocorrelation effects	yes	no	no
Category specific effects	yes	no	no
Standard errors for meta-analysis	yes	no	no
Censored data	yes	no	no
Truncated data	yes	no	yes
Customized covariances	yes	no	no
Missing value imputation	no	no	yes
Bayesian specifics:			
parallelization	yes	yes	yes
population-level priors	flexible	normal, Student-t	flexible
group-level priors	normal	normal	normal
covariance priors	flexible	restricted[5]	flexible
Other:			
Estimator	HMC, NUTS	HMC, NUTS	HMC, NUTS
Information criterion	WAIC, LOO	AIC, LOO	AIC, LOO
C++ compiler required	yes	no	yes
Modularized	no	no	no

Source: Bürkner, Paul-Christian (2022). brms: An R Package for Bayesian Multilevel Models using Stan.

Note: This table is from a 2022 publication. Newer versions of `brms` handle missing values.

brms vignettes as an indication of its versatility

General

- General Introduction to brms

Model types

- Advanced Multilevel Modeling with brms
- Bayesian Item Response Modeling with brms
- Estimating Distributional Models with brms
- Estimating Multivariate Models with brms
- Estimating Non-Linear Models with brms
- Estimating Phylogenetic Multilevel Models with brms

Auxiliary

- Define Custom Response Distributions with brms
- Parameterization of Response Distributions in brms
- Handle Missing Values with brms
- Estimating Monotonic Effects with brms
- Running brms models with within-chain parallelization

A function call to brms

```
lm_brms <- brms::brm(
  sup_afd ~                                 # outcome
    la_self,                                # predictor
  data = gles,                              # data
  family = gaussian(link = "identity"),     # family and link
  chains = 4L,                              # number of chains
  iter = 2000L,                             # number of iterations per chain
  warmup = 1000L,                           # number of warm-up samples per chain
  algorithm = "sampling",                   # algorithm (HMC/NUTS)
  backend = "rstan",                        # backend (rstan)
  seed = 20231123L                          # seed
)
```

What happens under the hood

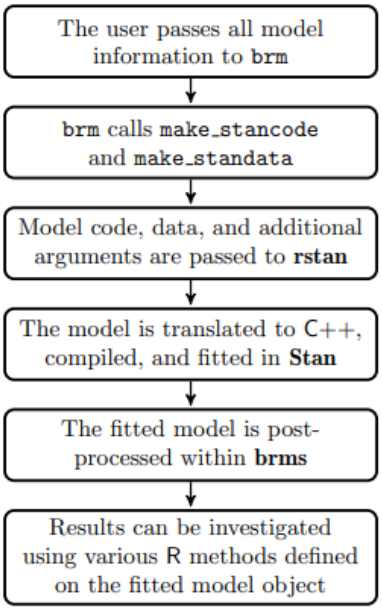


Figure 1: High-level description of the model fitting procedure used in **brms**.

Source: Bürkner, Paul-Christian (2022). brms: An R Package for Bayesian Multilevel Models using Stan.

Linear model

Likelihood

The linear model stipulates that the observed outcomes y_i for every unit i can be expressed as realizations from a normal distribution with unit-specific *mean or location parameter* μ_i and a constant (i.e., general) *variance or scale parameter* σ^2.

$$y_i \sim \mathrm{N}(\mu_i, \sigma^2) \text{ for all } i = 1, ...N$$

or, alternatively,

$$y_i = \mu_i + \epsilon_i \text{ for all } i = 1, ...N \epsilon_i \sim \mathrm{N}(0, \sigma^2)$$

The latter notation makes explicit that each observed y_i can be thought of as a combination of a *systematic component*, μ_i, and a *stochastic error component*, ϵ_i, which follows a zero-mean normal distribution with constant variance σ^2.

The systematic component

The systematic component is represented by the mean parameter μ_i. In fact, μ_i is merely a *transformed parameter*: It is a linear function of unit-specific data $\mathbf{x}_i$ and coefficients β.

The formula below illustrates this, using the row vector notation $\mathbf{x}_i'\beta$ as shorthand for the scalar notation $\beta_1 + \beta_2 x_{i,2} + ... + \beta_k x_{i,k}$.

$$\mu_i = \underbrace{\mathbf{x}_i'\beta}_{=\beta_1+\beta_2 x_{i,2}+...+\beta_k x_{i,k}} \quad \text{for all } i = 1, ...N$$

Parameters and priors

In the linear model, all coefficients β as well as the variance σ^2 are model parameters.

In Bayesian analysis, we must assign them priors (though `brms`, like Stan, will assign default uniform priors if we do not explicitly specify priors).

Data

We model respondents' support for the AfD (`sup_afd`, measured on an 11-point scale ranging from -5 to 5) as a function of respondents' pro-redistribution preferences (`se_self`) and anti-immigration preferences (`la_self`), a multiplicative interaction term between the two, and some controls: Gender (`fem`), age (`age`), and East/West residence (`east`).

Both `se_self` and `la_self` are measured on 11-point scales:

- `se_self` ranges from values (0) "less taxes and deductions, even if that means less social spending" to (10) "more social spending, even if that means more taxes and deductions".
- `la_self` ranges from values (0) "facilitate immigration" to (10) "restrict immigration".

The model formula is given by

$$\texttt{sup\_afd}_i = \beta_1 + \beta_2 \texttt{se\_self}_i + \beta_3 \texttt{la\_self}_i + \beta_4 \texttt{fem}_i + \beta_5 \texttt{east}_i + \beta_6 \texttt{age}_i + \beta_7 \texttt{se\_self}_i \times \texttt{la\_self}_i + \epsilon$$

Fitting

Choosing priors

`brms` uses default priors for certain "classes" of parameters. To check these defaults, we need to supply the model formula, data, and generative model (i.e., family and link function) to `brms::get_prior()`.

```
# Get default priors
default_priors <- brms::get_prior(
  sup_afd ~                                 # outcome
    la_self *                               # immigration preferences
    se_self +                               # redistribution preferences
    fem +                                   # gender
    east +                                  # east/west residence
    age,                                    # age
  data = gles,                              # data
  family = gaussian(link = "identity")      # family and link
)
default_priors
```

Note: Missing entries in the `prior` column denote flat/uniform priors.

Define custom priors

If we don't like the default priors, we can create a `brmsprior` object by specifying the desired distributional properties of parameters of various classes:

```r
custom_priors <- c(
  brms::prior(normal(0, 5), class = b),          # normal slopes
  brms::prior(normal(0, 5), class = Intercept),  # normal intercept
  brms::prior(cauchy(0, 5), class = sigma)       # half-cauchy SD
)
custom_priors
```

Let's think about these values intuitively. How informative/vague are our priors?

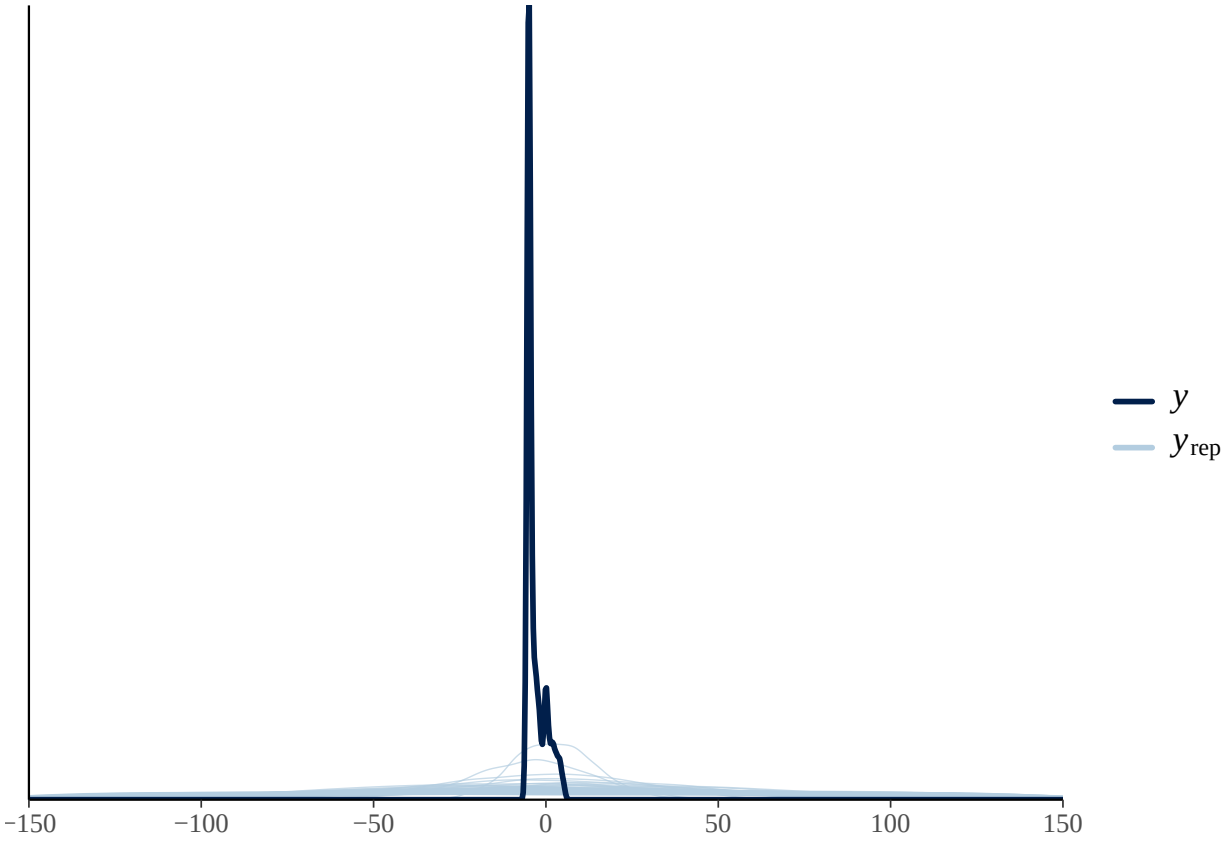
Prior predictive checks: Fitting

```r
lm_brms_prior_only <- brms::brm(
  sup_afd ~                                  # outcome
    la_self *                                # immigration preferences
    se_self +                                # redistribution preferences
    fem +                                    # gender
    east +                                   # east/west residence
    age,                                     # age
  data = gles,                               # data
  family = gaussian(link = "identity"),      # family and link
  prior = custom_priors,                     # priors
  sample_prior = "only",                     # samply only from prior
  chains = 2L,                               # number of chains
  iter = 1000L,                              # number of iterations per chain
  warmup = 0L,                               # number of warm-up samples per chain
  algorithm = "sampling",                    # algorithm (HMC/NUTS)
  backend = "rstan",                         # backend (rstan)
  seed = 20231123L                           # seed
)
```

Check model predictions vis-à-vis empirical distribution of outcome

```
brms::pp_check(lm_brms_prior_only, ndraws = 100, type = "dens_overlay") +
  xlim(-150, 150)
```

```
## Warning: Removed 33045 rows containing non-finite values (`stat_density()`).
```



Fitting the model

Lastly, we can fit the model using `brms::brm()`.

Note: Model compilation and estimation may take a while.

```
lm_brms <- brms::brm(
  sup_afd ~                                  # outcome
    la_self *                                # immigration preferences
```

```
    se_self +                                # redistribution preferences
    fem +                                    # gender
    east +                                   # east/west residence
    age,                                     # age
  data = gles,                               # data
  family = gaussian(link = "identity"),      # family and link
  prior = custom_priors,                     # priors
  chains = 4L,                               # number of chains
  iter = 2000L,                              # number of iterations per chain
  warmup = 1000L,                            # number of warm-up samples per chain
  algorithm = "sampling",                    # algorithm (HMC/NUTS)
  backend = "rstan",                         # backend (rstan)
  seed = 20231123L                           # seed
)
```

Summarize and diagnose

Model summary and generic diagnostics

First, we print the model summary. We can check `Rhat` for any signs of non-convergence.

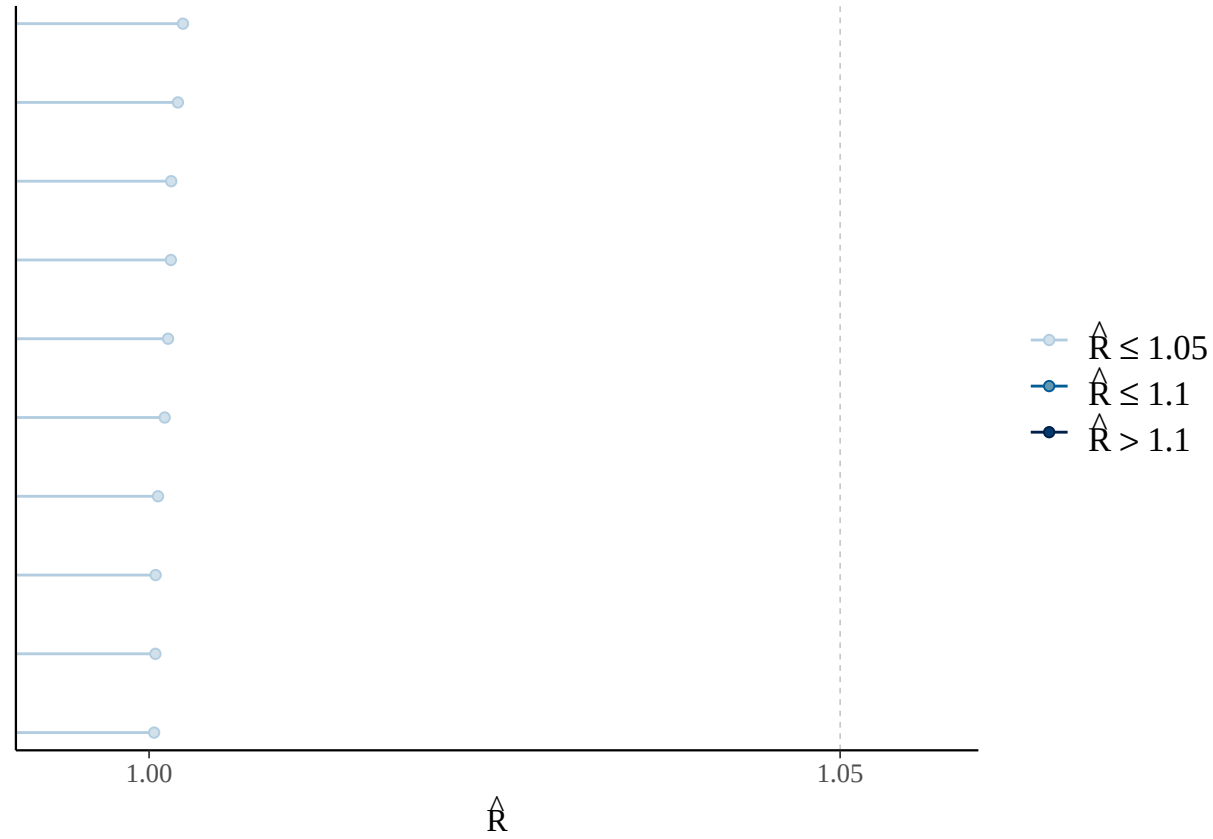
```
##  Family: gaussian
##   Links: mu = identity; sigma = identity
## Formula: sup_afd ~ la_self * se_self + fem + east + age
##    Data: gles (Number of observations: 1321)
##   Draws: 4 chains, each with iter = 2000; warmup = 1000; thin = 1;
##          total post-warmup draws = 4000
##
## Population-Level Effects:
##                Estimate Est.Error l-95% CI u-95% CI Rhat Bulk_ESS Tail_ESS
## Intercept         -3.81      0.43    -4.63    -2.95 1.00     2719     2473
## la_self            0.30      0.05     0.20     0.41 1.00     2474     2331
## se_self           -0.12      0.06    -0.25     0.01 1.00     2575     2151
```

```
## fem1               -0.59      0.14    -0.87    -0.32 1.00     4804     2719
## east1               0.42      0.15     0.14     0.71 1.00     4394     2740
## age                -0.01      0.00    -0.02    -0.01 1.00     5887     3038
## la_self:se_self     0.01      0.01    -0.01     0.03 1.00     2483     2183
##
## Family Specific Parameters:
##       Estimate Est.Error l-95% CI u-95% CI Rhat Bulk_ESS Tail_ESS
## sigma     2.44      0.05     2.36     2.53 1.00     5010     3136
##
## Draws were sampled using sampling(NUTS). For each parameter, Bulk_ESS
## and Tail_ESS are effective sample size measures, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

Visual diagnostics

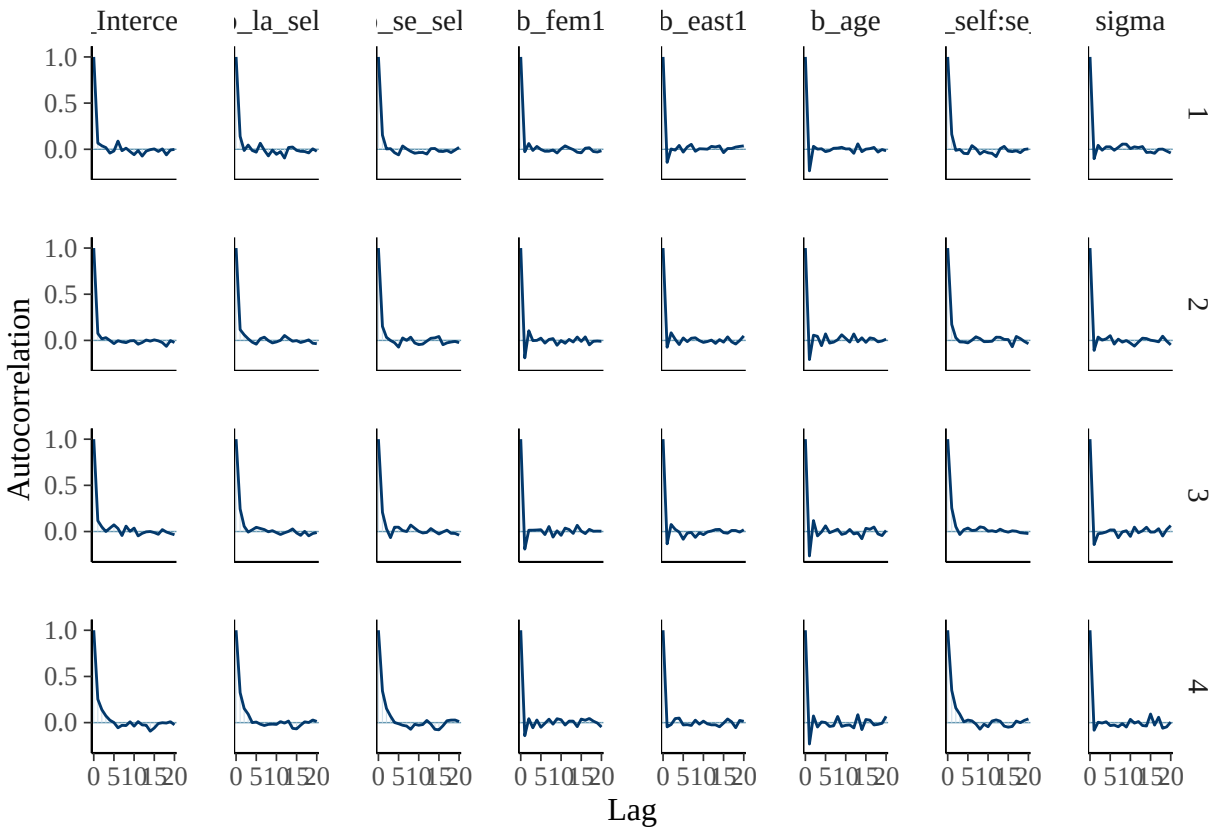
Let's explore the following visualizations of common generic diagnostics:

```
brms::mcmc_plot(lm_brms, type = "rhat")  # Gelman-Rubin
```



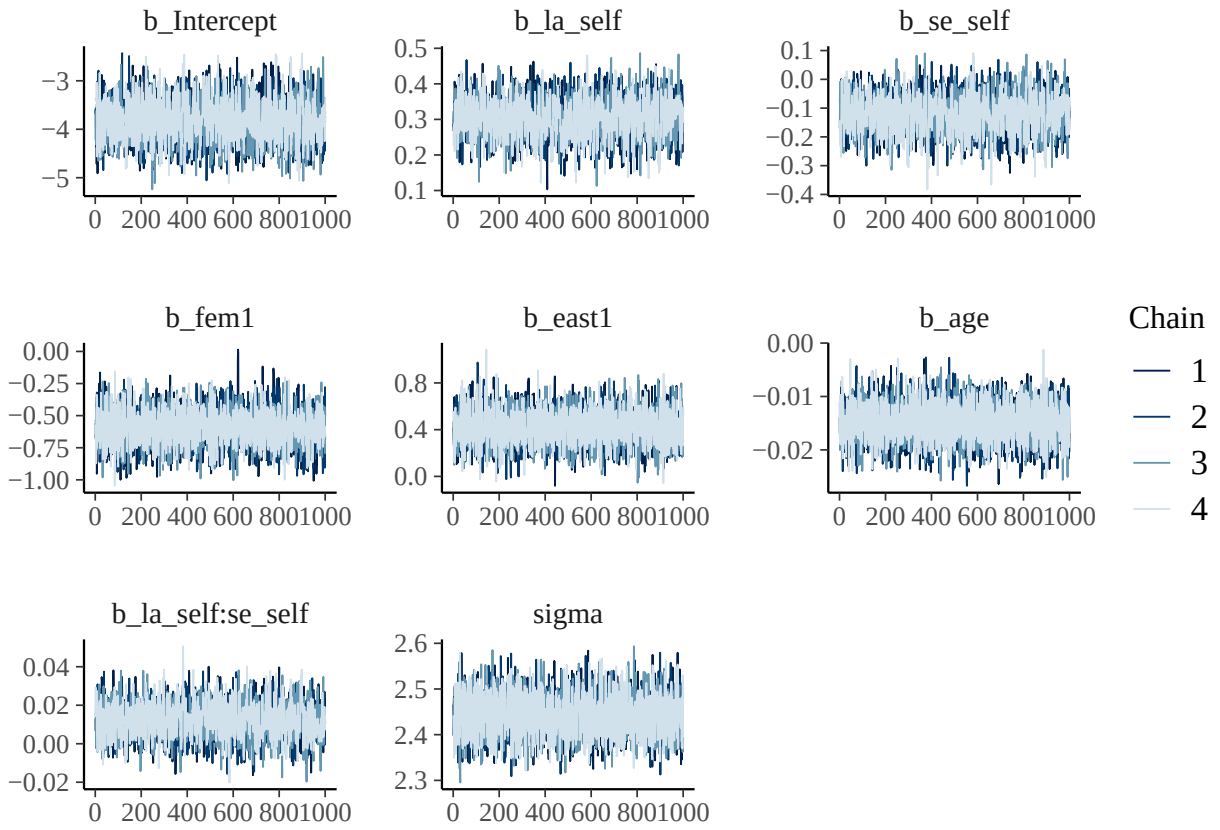

```
brms::mcmc_plot(lm_brms, type = "acf")   # Autocorrelation
```

```
## Warning: The ‘facets‘ argument of ‘facet_grid()‘ is deprecated as of ggplot2 2.2.0.
## i Please use the ‘rows‘ argument instead.
## i The deprecated feature was likely used in the bayesplot package.
##   Please report the issue at <https://github.com/stan-dev/bayesplot/issues/>.
## This warning is displayed once every 8 hours.
## Call ‘lifecycle::last_lifecycle_warnings()‘ to see where this warning was
## generated.
```




```
brms::mcmc_plot(lm_brms, type = "trace") # Trace plots
```

```
## No divergences to plot.
```



See `help(mcmc_plot)` for additional types of plots.

Algorithm-specific diagnostics

Note: We rely on the `check_hmc_diagnostics()` function from the `rstan` package. To ensure it works, we must extract the `stanfit` object nested in our `brmsfit` object via `lm_brms$fit`.

```
rstan::check_hmc_diagnostics(lm_brms$fit)
```

```
##
## Divergences:

## 0 of 4000 iterations ended with a divergence.

##
## Tree depth:
```

```
## 0 of 4000 iterations saturated the maximum tree depth of 10.

##
## Energy:

## E-BFMI indicated no pathological behavior.
```

What if I find signs of non-convergence?

The standard answer is: Increase the length of your chains. It may especially help with warnings about Rhat, ESS, and low BFMI.

But this is not always the optimal strategy, and it may not solve your problem.

So, suppose that after running longer chains, one or several of the following still apply:

- Your algorithm-specific diagnostics throw warnings (that won't go away)
- Your convergence diagnostics indicate signs of non-convergence (and increasing the warm-up period doesn't help)
- Your algorithm is painfully slow

Dealing with non-covergence and computational problems

Here are some answers, partly based on Gelman et al. (2020) and the Stan Development Team's guide *Runtime warnings and convergence problems*:

- Do read *Runtime warnings and convergence problems*. It can help you understand a specific problem and potential solutions.
- Do you get algorithm-specific warnings about divergences/max_treedepth? Adjust the HMC/NUTS control arguments `adapt_delta`, `stepsize`, and/or `max_treedepth`.
- Check if your model is well specified (e.g., do you have problems of separation in logistic regression?)
- Adopt an efficient workflow for debugging:
  - Reduce model complexity. Start with a simpler specification, gradually build up. See where things start to go wrong.

  - Use smaller sets of data, few chains, and short runs.

- Optimize priors:
  - If you use custom priors: Do your priors allow for posterior density in regions where you'd expect it?
  - If you use default flat or very vague priors: Use stronger priors (within reason)

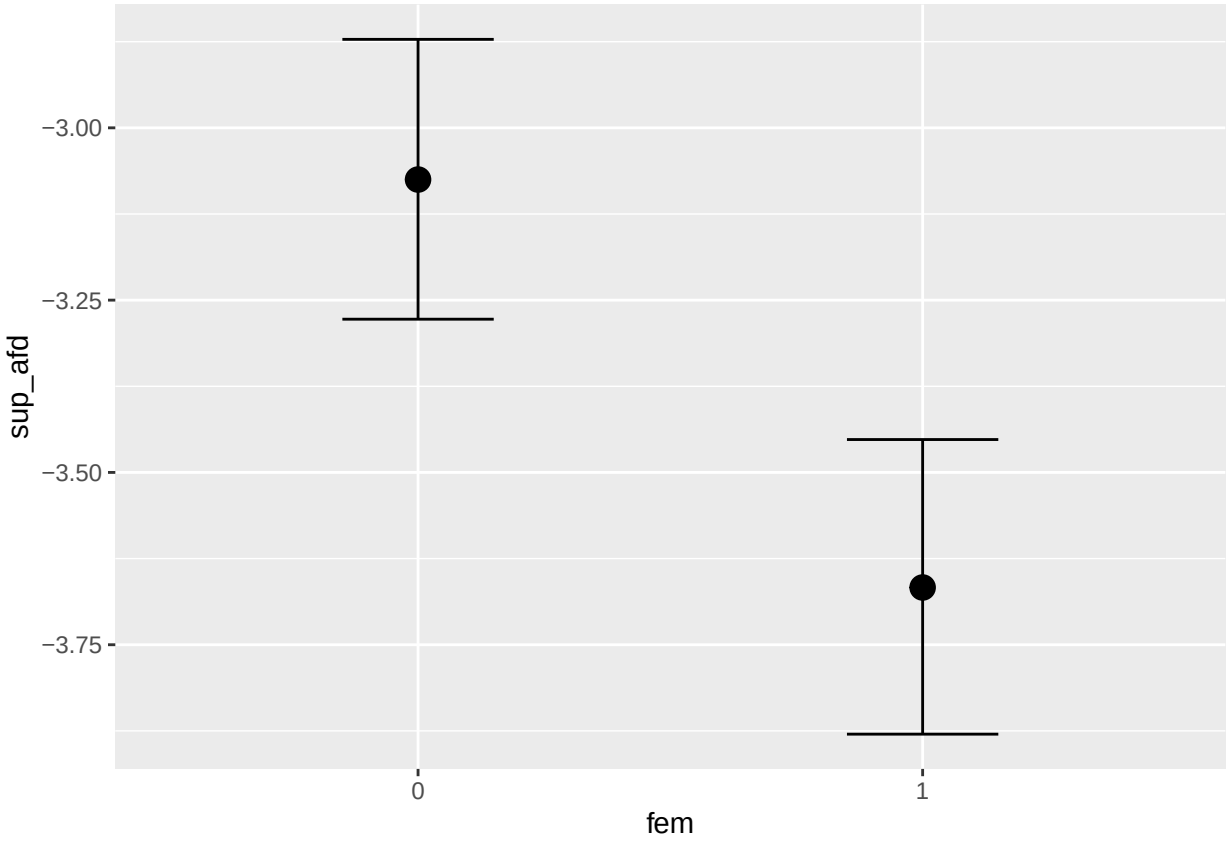
Interpretation: Quantities of interest

brms functions

`brms` offers pre-implemented functions for plotting conditional expectations (aka expected values, linear predictions, or adjusted predictions).

Below, for instance, are the expected values of AfD support for men (0) and women (1).

```
brms::conditional_effects(lm_brms,
                          effects = c("fem"),
                          points = TRUE)
```

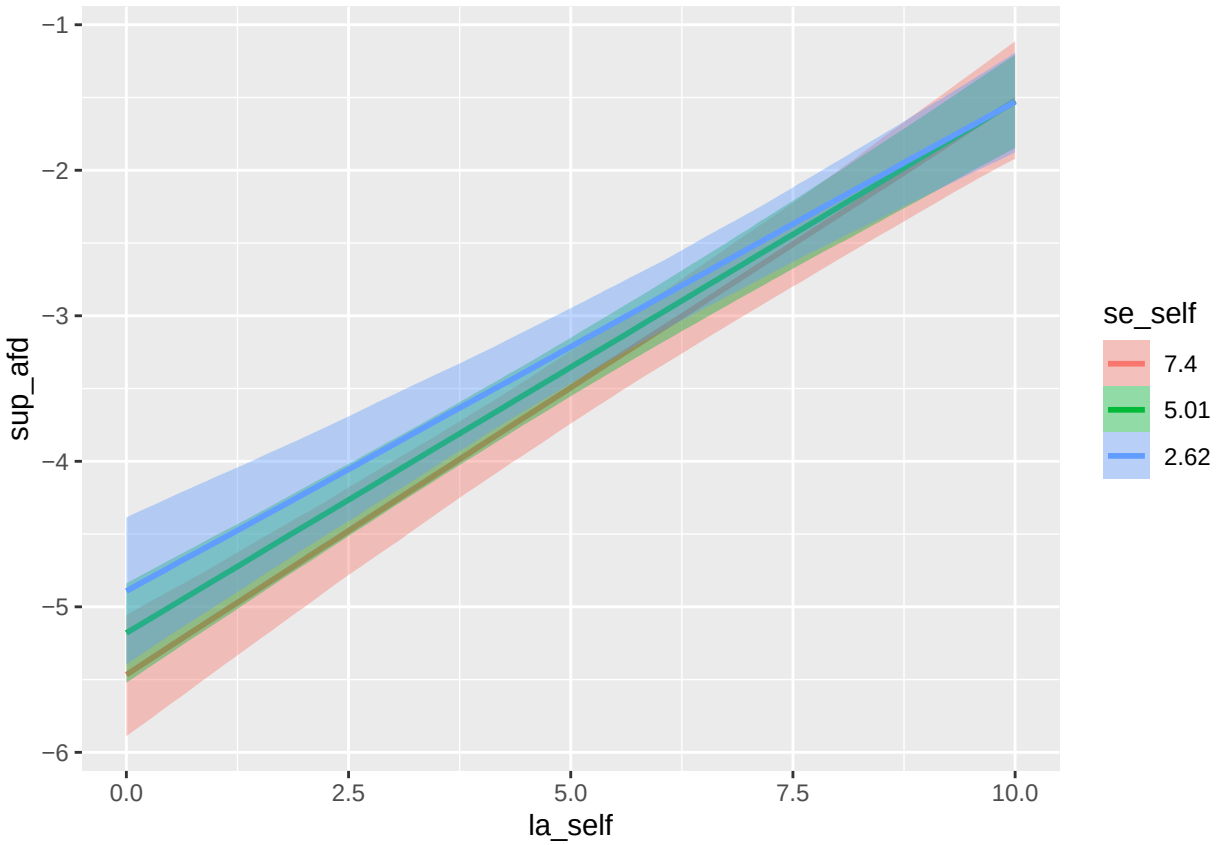


Limits of brms functions

However, brms' functions are somewhat limited for more complex quantities of interest.

For instance, the continuous-by-continuous interaction of `la_self` and `se_self`, `brms::conditional_effects()` will give you the conditional expectation of `sup_afd` as a function of `la_self` at three characteristic values of `se_self` (`mean(se_self) + c(-1, 0, 1) * sd(se_self)`), fixing all else at mean values:

```
brms::conditional_effects(lm_brms,
                          effects = c("la_self:se_self"))
```



What we would really like to get, however, is the conditional marginal effect of `la_self`, which shows how the effect of `la_self` changes as a function of the values of the moderator `se_self` (see Brambor et. al, 2006).

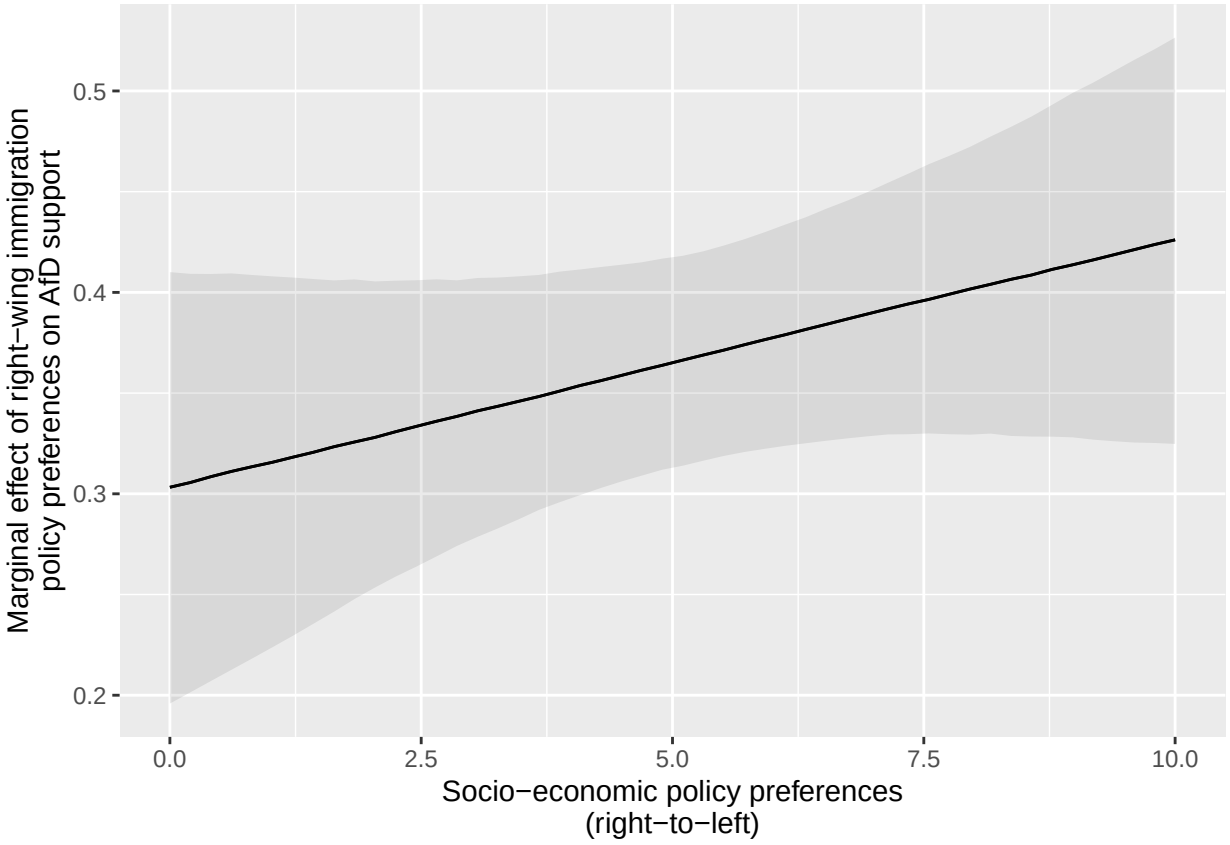
marginaleffects

The `marginaleffects` package ("Predictions, Comparisons, Slopes, Marginal Means, and Hypothesis Tests"), developed by Vincent Arel-Bundock, allows users to *"compute and plot predictions, slopes, marginal means, and comparisons (contrasts, risk ratios, odds, etc.) for over 100 classes of statistical and machine learning models in R"*.

A few years ago, compatibility with `brms` models was added.

Conditional marginal effects plot

```r
marginaleffects::plot_slopes(lm_brms,
                             variable = "la_self",
                             condition = "se_self") +
  ggplot2::xlab(paste("Socio-economic policy preferences",
                      "(right-to-left)",
                      sep = "\n")) +
  ggplot2::ylab(
    paste(
      "Marginal effect of right-wing immigration",
      "policy preferences on AfD support",
      sep = "\n"
    )
  )
```



Further reading

See the `marginaleffects` Case Study 8: Bayes for a complete overview of the package's compatibility with `brms`.

Posterior predictive checks

Posterior predictive checks involve simulating the data-generating process to obtain replicated data given the estimated model. They can help us determine how well our model fits the data.

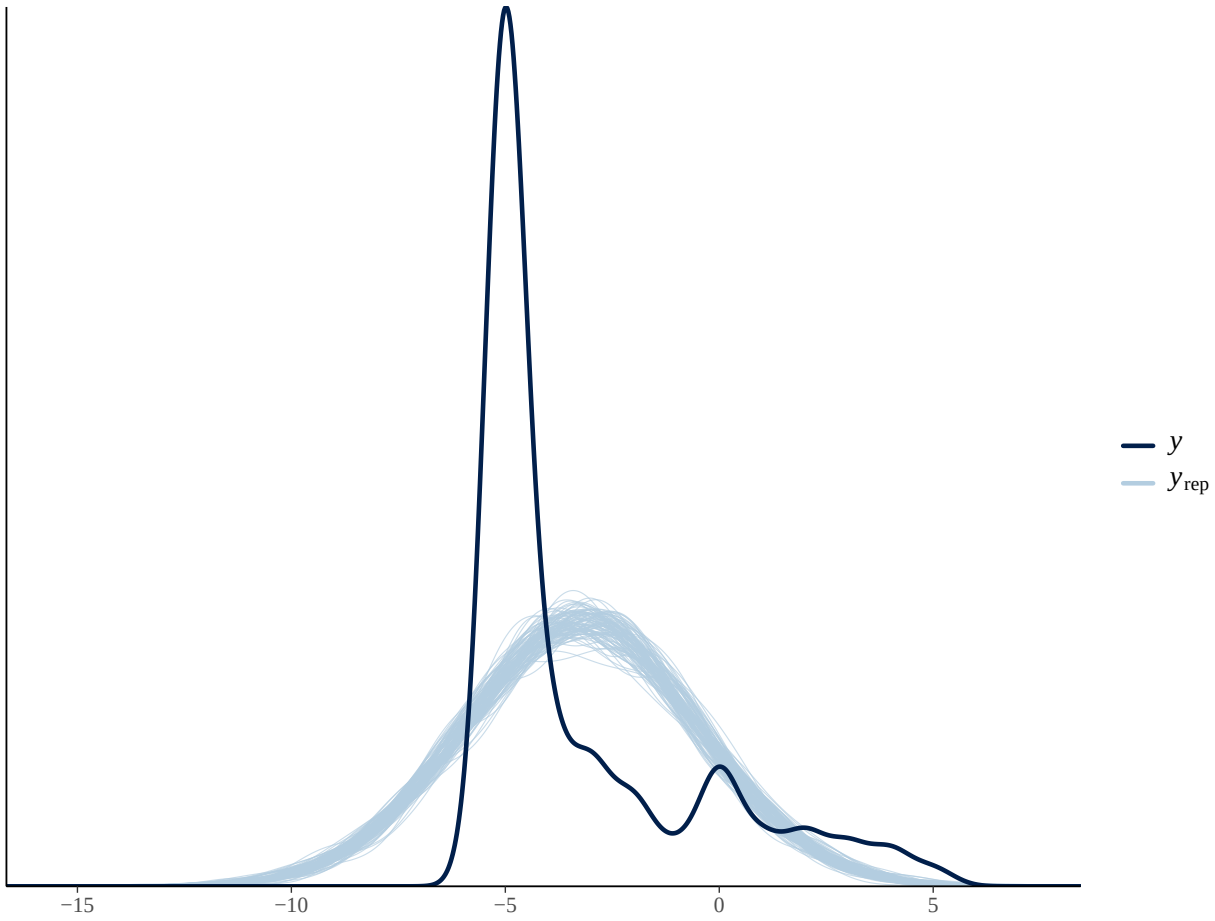
This usually involves two questions:

1. Does the *family* yield an adequate generative model?
    - Does a Gaussian (normal) data-generating processes produce realistic replications of the observed values of `sup_afd` (support for the AfD on the -5 to +5 scale)?
    - Does the simulated distribution of the replications match the observed distribution of the outcome in the *sample*?
2. Does the *systematic component* accurately predict outcomes?
    - Do our predictors accurately predict which individuals are more likely to support the AfD?
    - How large is the *observation-level discrepancy* between simulated replications and observed data?

Distributional congruence

To check whether the generative model produces distributions of replicated outcomes that match the distribution of the observed outcome, we can compare the density of the observed outcome with those of, say, `ndraws = 100` simulations. Each simulation is based on one post-warm-up sample from the posterior distribution.

```
brms::pp_check(lm_brms, ndraws = 100, type = "dens_overlay")
```



So, what do you think?

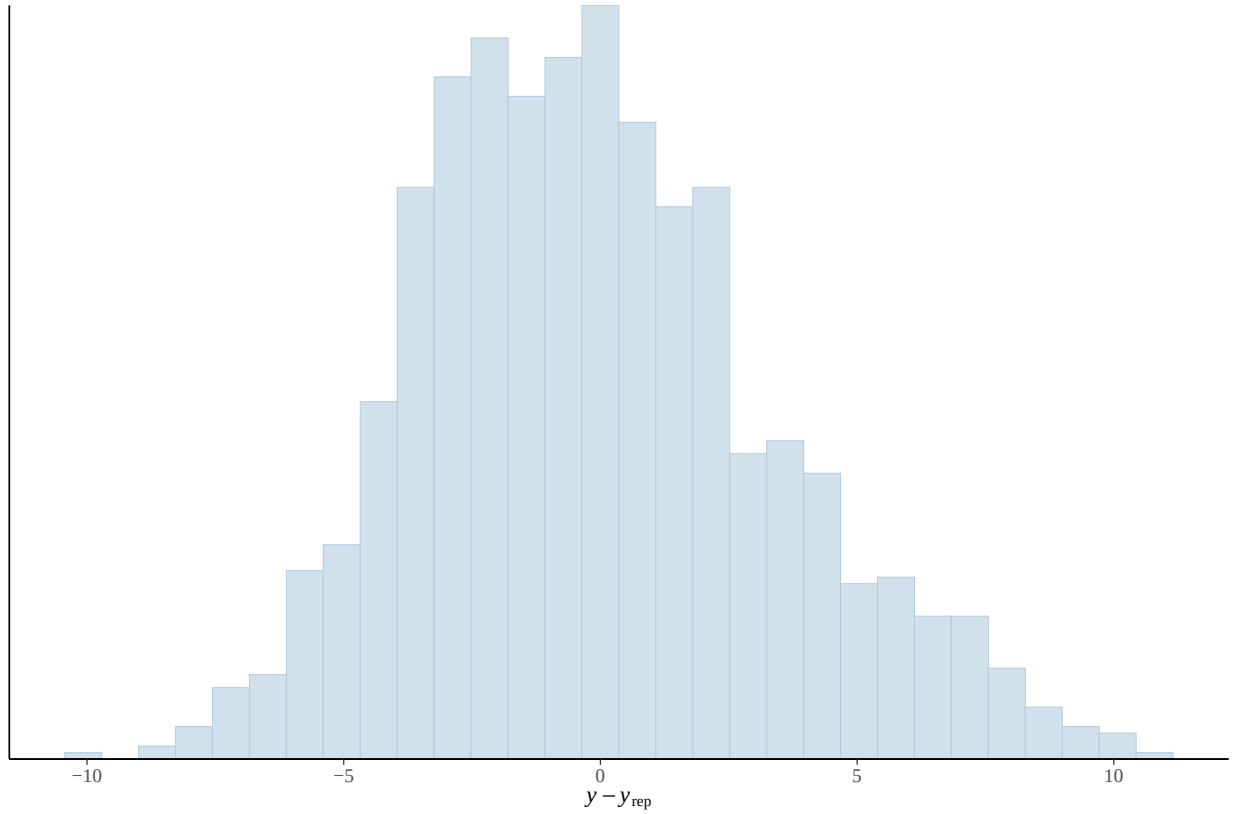
Observation-level prediction error

To check the predictive accuracy of the model, we can investigate the distribution of observation-level prediction errors. A model with perfect fit would produce an error of 0 for all N observations.

Below, you see the distribution of errors for our linear model. What do you think?

```
brms::pp_check(lm_brms, ndraws = 1, type = "error_hist")
```

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



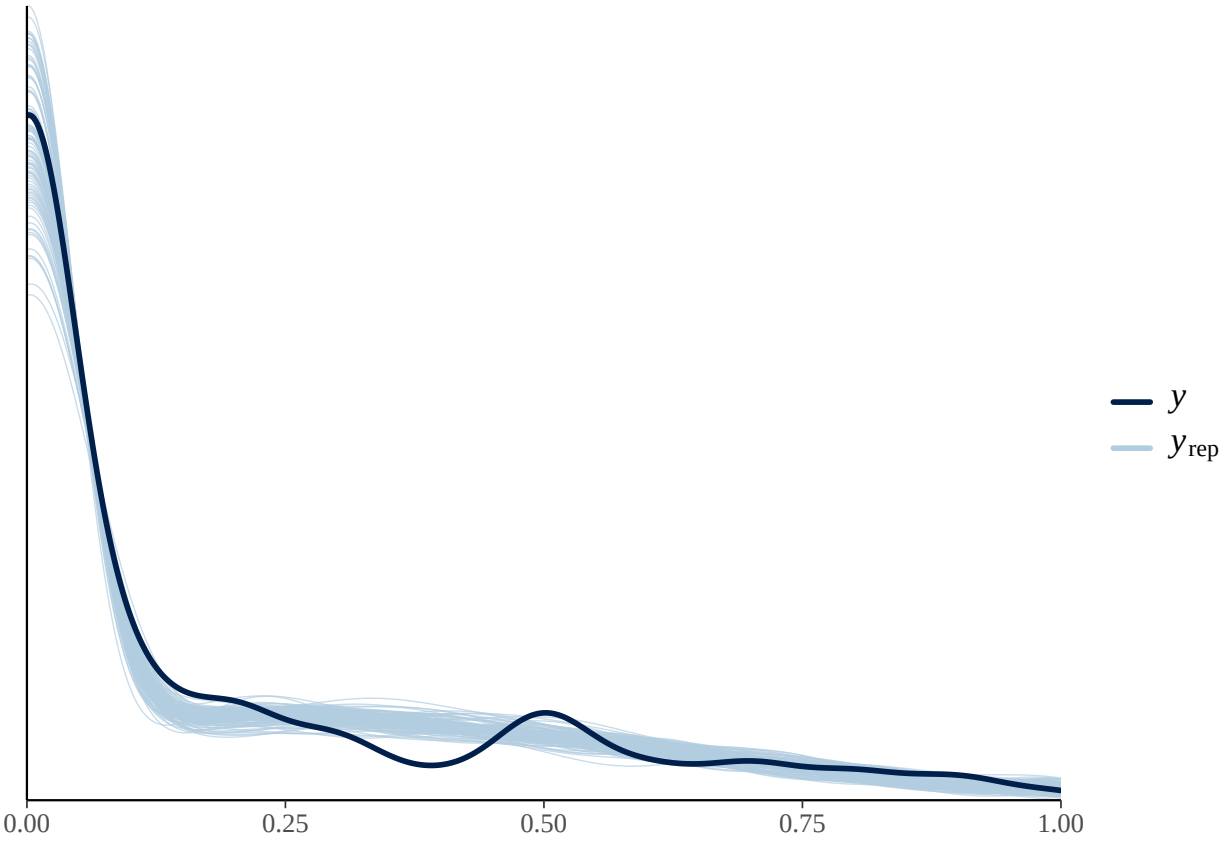
Comparison the model as a zero-one-inflated beta (ZOIB) regression

Zero-one-inflated beta (ZOIB) regression models bounded continuous outcomes on the unit (i.e., $[0, 1]$) interval. The ZOIB model is a GLM with a multi-family likelihood, meaning that its likelihood is composed of a mixture of several constitutive likelihoods. Specifically, it supplements a beta pdf for values $y \in]0, 1[$ with additional pmfs for the boundary values $y \in \{0, 1\}$.

To model a bounded continuous outcome on the unit interval, we must transform the scale of AfD support to range from 0 to 1 (with midpoint 0.5) instead of -5 to +5 (with midpoint 0), but we will scale it back later on.

We first observe the distributional congruence of the ZOIB-generated outcome simulations.

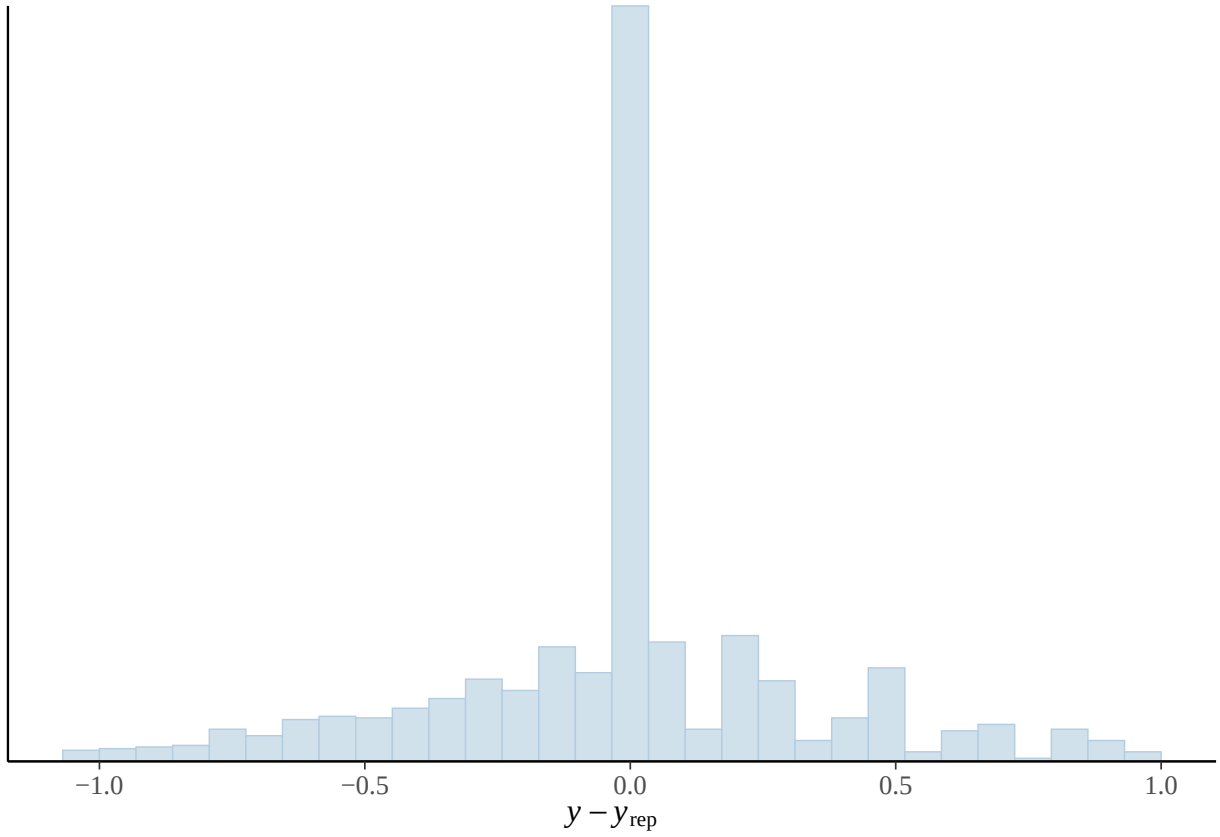
```
brms::pp_check(zoib_brms, ndraws = 100, type = "dens_overlay")
```



We then turn to checking observation-level prediction errors.

```
brms::pp_check(zoib_brms, ndraws = 1, type = "error_hist")
```

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



What do you conclude? Does the ZOIB-family accurately model the observed sample-level distribution of the outcome? Are you happy with the predictive accuracy of our current systematic component?