Lecture: Applied Bayesian Statistics Using Stan: Basics

Denis Cohen*

Stan

What is Stan?

In the words of the developers:

"Stan is a state-of-the-art platform for statistical modeling and high-performance statistical computation. Thousands of users rely on Stan for statistical modeling, data analysis, and prediction in the social, biological, and physical sciences, engineering, and business.

Users specify log density functions in Stan's probabilistic programming language and get:

- full Bayesian statistical inference with MCMC sampling (NUTS, HMC)
- approximate Bayesian inference with variational inference (ADVI)
- penalized maximum likelihood estimation with optimization (L-BFGS)

Source: https://mc-stan.org/

Why Stan?

- Open-source software
- Fast and stable algorithms
- High flexibility with few limitations
- Extensive documentation
  - User's Guide
  - Language Reference Manual

*Mannheim Centre for European Social Research, University of Mannheim, 68131 Mannheim, Germany. denis.cohen@uni-mannheim.de.

  - Language Functions Reference

- Highly transparent development process; see Stan Development Repository on Github
- Very responsive Development Team
- Large and active community in the Stan Forums and Stack OVerflow
- Increasing number of case studies, tutorials, papers and textbooks
- Compatibility with various editor for syntax highlighting, formatting, and checking (incl. RStudio and Emacs)

Stan interfaces

- RStan (R)
- PyStan (Python)
- CmdStan (shell, command-line terminal)
- MatlabStan (MATLAB)
- Stan.jl (Julia)
- StataStan (Stata)
- MathematicaStan (Mathematica)
- ScalaStan (Scala)

(Some) R packages

- **rstan**: General R Interface to Stan
- **shinystan**: Interactive Visual and Numerical Diagnostics and Posterior Analysis for Bayesian Models
- **bayesplot**: Plotting functions for posterior analysis, model checking, and MCMC diagnostics.
- **brms**: Bayesian Regression Models using 'Stan', covering a growing number of model types
- **rstanarm**: Bayesian Applied Regression Modeling via Stan, with an emphasis on hierarchical/multilevel models
- **edstan**: Stan Models for Item Response Theory
- **rstantools**: Tools for Developing R Packages Interfacing with 'Stan'

Caveat: Reproducibility

Under what conditions are estimates reproducible? See Stan Reference Manual, Section 19:

- Stan version
- Stan interface (RStan, PyStan, CmdStan) and version, plus version of interface language (R, Python, shell)
- versions of included libraries (Boost and Eigen)
- operating system version
- computer hardware including CPU, motherboard and memory
- C++ compiler, including version, compiler flags, and linked libraries
- same configuration of call to Stan, including random seed, chain ID, initialization and data

Bayesian workflow

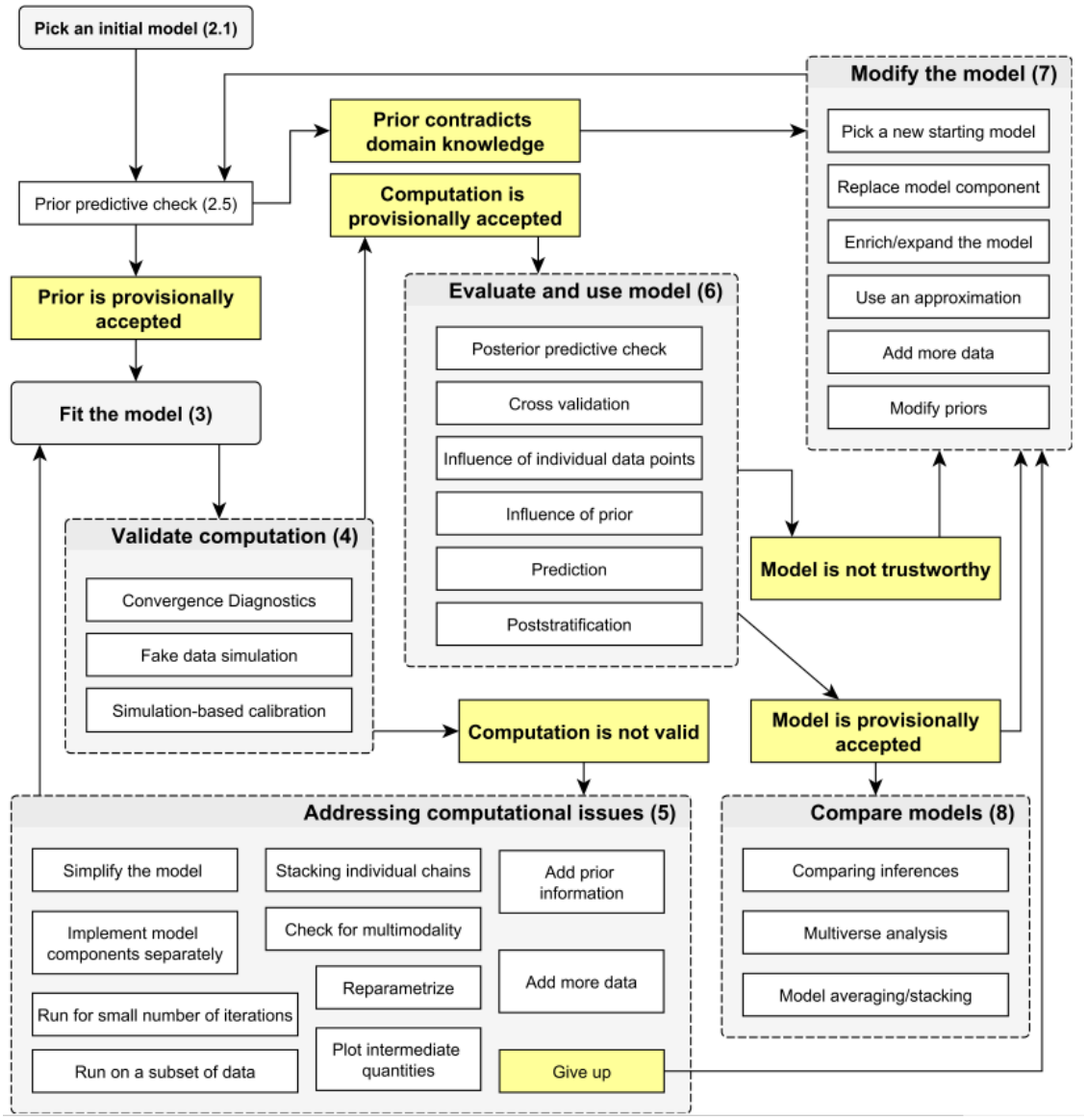
A quick overview

The short version

1. **Specification**: Specify the full probability model
    - data
    - likelihood
    - priors
2. **Model Building**: Translate the model into code
3. **Validation**: Validate the model with fake data
4. **Fitting**: Fit the model to actual data
5. **Diagnosis**: Check generic and algorithm-specific diagnostics to assess convergence
6. Posterior Predictive Checks
7. Model Comparison

Source: Jim Savage (2016) A quick-start introduction to Stan for economists. A QuantEcon Notebook.

The long version



Source: Gelman, A., Vehtari, A., Simpson, D., Margossian, C. C., Carpenter, B., Yao, Y., Kennedy, L., Gabry, J., Bürkner, P. C., & Modrák, M. (2020). Bayesian workflow.

Specification (linear model)

Reminder: Equivalent notations

1. Scalar form:

$$y_i = \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i3} + \epsilon_i \text{ for all } i = 1, ..., N$$

2. Row-vector form:

$$y_i = \mathbf{x_i'}\beta + \epsilon_i \text{ for all } i = 1, ..., N$$

3. Column-vector form:

$$\mathbf{y} = \beta_1\mathbf{x_1} + \beta_2\mathbf{x_2} + \beta_3\mathbf{x_3}\epsilon$$

4. Matrix form:

$$\mathbf{y} = \mathbf{X}\beta + \epsilon$$

Example: Linear model

Our knowledge of generalized linear models gives us *almost* everything we need!

Probability model for the data

First, let's recap the three parts of every GLM in the context of the linear model:

- Family: $\mathbf{y} \sim \text{Normal}(\eta, \sigma)$
- (Inverse) link function: $\mathbf{y}^* = \text{id}(\eta) = \eta$
- Linear component: $\eta = \mathbf{X}\beta$

The *family* specifies the probability model (a.k.a. likelihood, data-generating process, or generative model) for the *data*: The fundamental assumption of the linear model is that every observation y_i is a realization from a normal pdf with location parameter (mean) η_i and constant scale parameter (variance) σ^2.

Note: Mimicking the convention in both R and Stan, we parameterize the normal distribution in terms of its mean and *standard deviation* (not variance)!

Known and unknown quantities

- Parameters (unknown, random quantities):
  - β, the coefficient vector
  - σ, the scale parameter of the normal
  - η, the location parameter of the normal

- Data (known, fixed quantities):
  - $\mathbf{y}$, the outcome vector
  - $\mathbf{X}$, the design matrix
  - the dimensions of $\mathbf{y}_{N\times 1}$ and $\mathbf{X}_{N\times K}$
  - the dimensions of $\beta_{K\times 1}$, σ (a scalar), and $\eta_{N\times 1}$

Priors

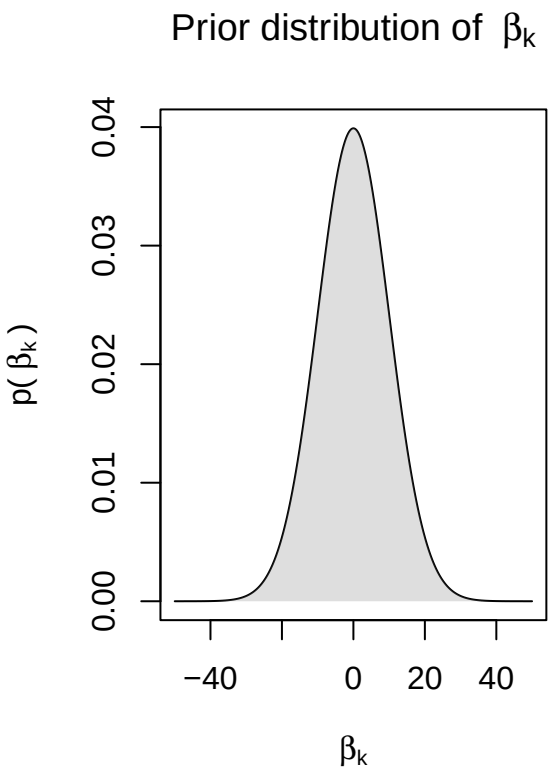
What is still missing are prior distributions for the unknown quantities.

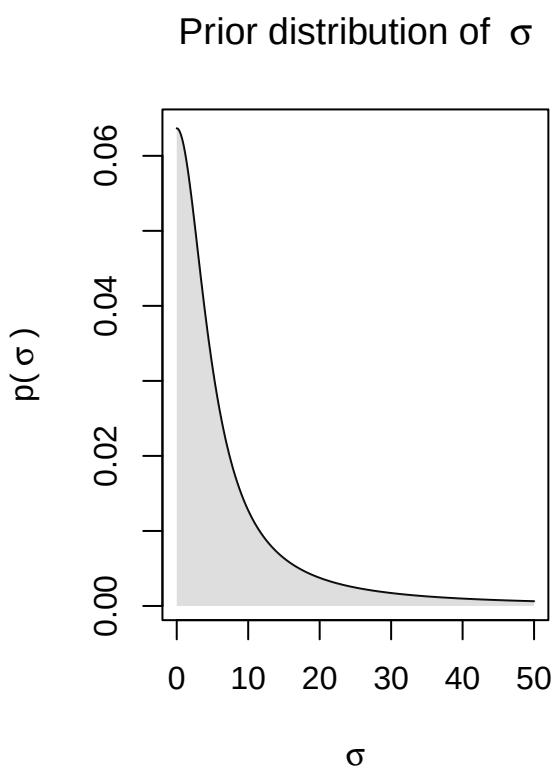
Here, we have quite some discretion. There are few rules we must adhere to:

- Our β's have unconstrained support (though by far not all value ranges may be reasonable!)
- The scale parameter σ cannot be negative

Here, we will opt for a convenience solution and specify weakly informative zero-mean normal priors for the β's and a weakly informative half-Cauchy prior for σ:

- $\beta \sim \mathrm{N}(0, 10)$
- $\sigma \sim \mathrm{Cauchy}^{+}(0, 5)$





Model building

Stan Program Blocks

1. Functions: Declare user written functions
2. **Data**: Declare all known quantities
3. Transformed Data: Transform declared data inputs (once)
4. **Parameters**: Declare all unknown quantities
5. Transformed Parameters: Transform declared parameters (each step, each iteration)
6. **Model**: Transform parameters, specify prior distributions and likelihoods
7. Generated Quantities (each iteration)

Program Blocks

	data	transformed data	parameters	transformed parameters	model	generated quantities
Execution	Per chain	Per chain	NA	Per leapfrog	Per leapfrog	Per sample
Variable Declarations	Yes	Yes	Yes	Yes	Yes	Yes
Variable Scope	Global	Global	Global	Global	Local	Local
Variables Saved?	No	No	Yes	Yes	No	Yes
Modify Posterior?	No	No	No	No	Yes	No
Random Variables	No	No	No	No	No	Yes

Source: http://mlss2014.hiit.fi/mlss_files/2-stan.pdf

Script for a Stan program

Writing scripts for Stan programs

- Start with a blank script in your preferred code editor and save it as "lm.stan" .
- This will enable syntax highlighting, formatting, and checking in RStudio and Emacs.
- Alternatively, you can save your model as a single character string in R (with some drawbacks).

Style guide

- There is a style guide. Some recommendations:
  - consistency
  - lines should not exceed 80 characters
  - lowercase variables, words separated by underscores
  - like R: space around operators: `y ~ normal(...)`, `x = (1 + 2) * 3`
  - spaces after commas are optional: `y[m,n] ~ normal(0,1)` or `y[m, n] ~ normal(0, 1)`
- Always make sure to end your script with a blank line.
- You must use a delimiter to finish lines: `;`.
- `// this is a comment`

Data block

Declare all known quantities, including data types, dimensions, and constraints:

- $\mathbf{y}_{N \times 1}$
- $\mathbf{X}_{N \times K}$

```
data {
  int<lower=1> N; // num. observations
  int<lower=1> K; // num. predictors
  matrix[N, K] x; // model matrix
  vector[N] y;    // outcome vector
}
```

Parameters block

Declare unknown 'base' quantities, including storage types, dimensions, and constraints:

- β, the coefficient vector
- σ, the scale parameter of the normal

```
parameters {
  ... declarations ...
}
```

Transformed parameters block

Declare and specify unknown transformed quantities, including storage types, dimensions, and constraints:

- $\eta = \mathbf{X}\beta$, the linear prediction

```
transformed parameters {
  ... declarations ... statements ....
}
```

Model block

Declare and specify local variables (optional) and specify sampling statements:

- $\beta_k \sim \text{Normal}(0, 10)$ for k = 1,...,K
- $\sigma \sim \text{Cauchy}^{+}(0, 5)$
- $\mathbf{y} \sim \text{Normal}(\eta, \sigma)$

```
model {
  // priors
  ... statements ...

  // log-likelihood
  ... statements ...
}
```

Writing Stan programs in R

- You can supply Stan programs as a character string in R
- Downsides:
  - No syntax highlighting, formatting, and checking
  - Must use double quotation marks `"` around the string to avoid that the transposition operator `'` breaks the string
- Upsides: Works with the interactive `learnr` tutorials in our workshop!

```
# Save as character
lm_code <-
"data {
```

```
  int<lower=1> N; // num. observations
  int<lower=1> K; // num. predictors
  matrix[N, K] x; // design matrix
  vector[N] y;    // outcome vector
}

parameters {
  vector[K] beta;      // coef vector
  real<lower=0> sigma; // scale parameter
}

transformed parameters {
  vector[N] eta;  // declare lin. pred.
  eta = x * beta; // assign lin. pred.
}

model {
  // priors
  target += normal_lpdf(beta | 0, 10);  // priors for beta
  target += cauchy_lpdf(sigma | 0, 5);  // prior for sigma

  // log-likelihood
  target += normal_lpdf(y | eta, sigma); // likelihood
}"

# Write to script
writeLines(lm_code, con = "lm.stan")
```

Validation

Simulate the data-generating process in R

Setup and compilation

```
## Setup
library(rstan)
rstan_options(auto_write = TRUE)              # avoid recompilation of models
options(mc.cores = parallel::detectCores())  # parallelize across all CPUs

## Data as list
standat_val <- list(
  N = N,
  K = K,
  x = x,
  y = y_sim
)

## C++ Compilation
lm_mod <- rstan::stan_model(model_code = lm_code)
```

Estimation

```
lm_val <- rstan::sampling(
  lm_mod,                     # compiled model
  data = standat_val,             # data input
  algorithm = "NUTS",         # algorithm
  control = list(             # control arguments
    adapt_delta = .85),
  save_warmup = FALSE,        # discard warmup sims
  sample_file = NULL,         # no sample file
```

```
  diagnostic_file = NULL,      # no diagnostic file
  pars = c("beta", "sigma"),   # select parameters
  iter = 2000L,                # iter per chain
  warmup = 1000L,              # warmup period
  thin = 2L,                   # thinning factor
  chains = 2L,                 # num. chains
  cores = 2L,                  # num. cores
  seed = 20210329)             # seed
```

Output summary

Reminder: Here are the 'true' parameter values:

```
true_pars <- c(beta, sigma)
names(true_pars) <- c(paste0("beta[", 1:5, "]"), "sigma")
true_pars
```

```
##    beta[1]    beta[2]    beta[3]    beta[4]    beta[5]      sigma
## -0.5243799 -1.1456866 -1.4379737  0.3127257  2.4365995  2.5000000
```

And here are the estimates from our model:

```
lm_val
```

```
## Inference for Stan model: 37c095a6cbe3e8a27908a77406b6b94d.
## 2 chains, each with iter=2000; warmup=1000; thin=2;
## post-warmup draws per chain=500, total post-warmup draws=1000.
##
##             mean se_mean   sd     2.5%      25%      50%      75%    97.5%
## beta[1]    -0.61    0.00 0.08    -0.78    -0.67    -0.61    -0.55    -0.45
## beta[2]    -1.05    0.00 0.08    -1.21    -1.11    -1.05    -1.00    -0.90
## beta[3]    -1.50    0.00 0.08    -1.66    -1.56    -1.50    -1.45    -1.34
## beta[4]     0.34    0.00 0.08     0.18     0.29     0.33     0.39     0.49
```

```
## beta[5]     2.50    0.00 0.09     2.32     2.44     2.50     2.56     2.66
## sigma       2.53    0.00 0.06     2.42     2.49     2.53     2.58     2.66
## lp__    -2367.19    0.07 1.94 -2371.89 -2368.19 -2366.81 -2365.79 -2364.58
##         n_eff Rhat
## beta[1]   925 1.00
## beta[2]   913 1.01
## beta[3]   875 1.00
## beta[4]   946 1.00
## beta[5]   851 1.00
## sigma     918 1.00
## lp__      678 1.00
##
## Samples were drawn using NUTS(diag_e) at Fri Mar 26 13:55:16 2021.
## 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).
```

- When comparing these estimates, the question is, of course, how much deviation should have us worried.
- Deviations from a single validation run may be due to a circumstantial simulation of 'extreme' outcome values when mimicking the data generating process.
- Cook, Gelman, and Rubin (2006) thus recommend running many replications of such validation simulations.
- They also provide a useful test statistic.

A stanfit object

```
str(lm_val)
```

```
## Formal class 'stanfit' [package "rstan"] with 10 slots
##   ..@ model_name: chr "37c095a6cbe3e8a27908a77406b6b94d"
##   ..@ model_pars: chr [1:4] "beta" "sigma" "mu" "lp__"
```

```
##   ..@ par_dims  :List of 4
##   .. ..$ beta : num 5
##   .. ..$ sigma: num(0)
##   .. ..$ mu   : num 1000
##   .. ..$ lp__ : num(0)
##   ..@ mode      : int 0
##   ..@ sim       :List of 12
##   .. ..$ samples    :List of 2
##   .. .. ..$ :List of 7
##   .. .. .. ..$ beta[1]: num [1:500] -0.563 -0.583 -0.666 -0.715 -0.704 ...
##   .. .. .. ..$ beta[2]: num [1:500] -1.046 -1.133 -0.912 -0.954 -0.943 ...
##   .. .. .. ..$ beta[3]: num [1:500] -1.36 -1.51 -1.56 -1.41 -1.33 ...
##   .. .. .. ..$ beta[4]: num [1:500] 0.347 0.418 0.387 0.285 0.332 ...
##   .. .. .. ..$ beta[5]: num [1:500] 2.58 2.44 2.42 2.38 2.3 ...
##   .. .. .. ..$ sigma  : num [1:500] 2.57 2.58 2.66 2.55 2.56 ...
##   .. .. .. ..$ lp__   : num [1:500] -2367 -2366 -2369 -2367 -2371 ...
##   .. .. .. ..- attr(*, "test_grad")= logi FALSE
##   .. .. .. ..- attr(*, "args")=List of 16
##   .. .. .. .. ..$ append_samples    : logi FALSE
##   .. .. .. .. ..$ chain_id          : num 1
##   .. .. .. .. ..$ control           :List of 12
##   .. .. .. .. .. ..$ adapt_delta      : num 0.85
##   .. .. .. .. .. ..$ adapt_engaged    : logi TRUE
##   .. .. .. .. .. ..$ adapt_gamma      : num 0.05
##   .. .. .. .. .. ..$ adapt_init_buffer: num 75
##   .. .. .. .. .. ..$ adapt_kappa      : num 0.75
##   .. .. .. .. .. ..$ adapt_t0         : num 10
##   .. .. .. .. .. ..$ adapt_term_buffer: num 50
##   .. .. .. .. .. ..$ adapt_window     : num 25
##   .. .. .. .. .. ..$ max_treedepth    : int 10
##   .. .. .. .. .. ..$ metric           : chr "diag_e"
##   .. .. .. .. .. ..$ stepsize         : num 1
```

```
##   .. .. .. .. .. ..$ stepsize_jitter  : num 0
##   .. .. .. .. ..$ enable_random_init: logi TRUE
##   .. .. .. .. ..$ init              : chr "random"
##   .. .. .. .. ..$ init_list         : NULL
##   .. .. .. .. ..$ init_radius       : num 2
##   .. .. .. .. ..$ iter              : int 2000
##   .. .. .. .. ..$ method            : chr "sampling"
##   .. .. .. .. ..$ random_seed       : chr "20210329"
##   .. .. .. .. ..$ refresh           : int 200
##   .. .. .. .. ..$ sampler_t         : chr "NUTS(diag_e)"
##   .. .. .. .. ..$ save_warmup       : logi FALSE
##   .. .. .. .. ..$ test_grad         : logi FALSE
##   .. .. .. .. ..$ thin              : int 2
##   .. .. .. .. ..$ warmup            : int 1000
##   .. .. .. ..- attr(*, "inits")= num [1:1006] -1.928 -1.9 -0.547 -0.936 1.761 ...
##   .. .. .. ..- attr(*, "mean_pars")= num [1:1006] -0.611 -1.053 -1.505 0.34 2.504 ...
##   .. .. .. ..- attr(*, "mean_lp__")= num -2367
##   .. .. .. ..- attr(*, "adaptation_info")= chr "# Adaptation terminated\n# Step size = 0.591
##   .. .. .. ..- attr(*, "elapsed_time")= Named num [1:2] 0.277 0.247
##   .. .. .. .. ..- attr(*, "names")= chr [1:2] "warmup" "sample"
##   .. .. .. ..- attr(*, "sampler_params")=List of 6
##   .. .. .. .. ..$ accept_stat__: num [1:500] 0.979 0.896 1 1 0.99 ...
##   .. .. .. .. ..$ stepsize__   : num [1:500] 0.592 0.592 0.592 0.592 0.592 ...
##   .. .. .. .. ..$ treedepth__  : num [1:500] 3 3 3 3 3 3 2 2 2 3 ...
##   .. .. .. .. ..$ n_leapfrog__ : num [1:500] 7 7 7 7 7 7 7 3 3 7 ...
##   .. .. .. .. ..$ divergent__  : num [1:500] 0 0 0 0 0 0 0 0 0 0 ...
##   .. .. .. .. ..$ energy__     : num [1:500] 2373 2372 2372 2370 2372 ...
##   .. .. .. ..- attr(*, "return_code")= int 0
##   .. .. ..$ :List of 7
##   .. .. .. ..$ beta[1]: num [1:500] -0.772 -0.604 -0.594 -0.581 -0.694 ...
##   .. .. .. ..$ beta[2]: num [1:500] -1.1 -1.06 -1.05 -1.05 -1.12 ...
##   .. .. .. ..$ beta[3]: num [1:500] -1.72 -1.52 -1.55 -1.45 -1.42 ...
```

```
##   .. .. .. ..$ beta[4]: num [1:500] 0.311 0.388 0.277 0.381 0.291 ...
##   .. .. .. ..$ beta[5]: num [1:500] 2.57 2.37 2.45 2.57 2.59 ...
##   .. .. .. ..$ sigma  : num [1:500] 2.5 2.6 2.57 2.46 2.45 ...
##   .. .. .. ..$ lp__   : num [1:500] -2371 -2366 -2365 -2366 -2367 ...
##   .. .. .. ..- attr(*, "test_grad")= logi FALSE
##   .. .. .. ..- attr(*, "args")=List of 16
##   .. .. .. .. ..$ append_samples    : logi FALSE
##   .. .. .. .. ..$ chain_id          : num 2
##   .. .. .. .. ..$ control           :List of 12
##   .. .. .. .. .. ..$ adapt_delta      : num 0.85
##   .. .. .. .. .. ..$ adapt_engaged    : logi TRUE
##   .. .. .. .. .. ..$ adapt_gamma      : num 0.05
##   .. .. .. .. .. ..$ adapt_init_buffer: num 75
##   .. .. .. .. .. ..$ adapt_kappa      : num 0.75
##   .. .. .. .. .. ..$ adapt_t0         : num 10
##   .. .. .. .. .. ..$ adapt_term_buffer: num 50
##   .. .. .. .. .. ..$ adapt_window     : num 25
##   .. .. .. .. .. ..$ max_treedepth    : int 10
##   .. .. .. .. .. ..$ metric           : chr "diag_e"
##   .. .. .. .. .. ..$ stepsize         : num 1
##   .. .. .. .. .. ..$ stepsize_jitter  : num 0
##   .. .. .. .. ..$ enable_random_init: logi TRUE
##   .. .. .. .. ..$ init              : chr "random"
##   .. .. .. .. ..$ init_list         : NULL
##   .. .. .. .. ..$ init_radius       : num 2
##   .. .. .. .. ..$ iter              : int 2000
##   .. .. .. .. ..$ method            : chr "sampling"
##   .. .. .. .. ..$ random_seed       : chr "20210329"
##   .. .. .. .. ..$ refresh           : int 200
##   .. .. .. .. ..$ sampler_t         : chr "NUTS(diag_e)"
##   .. .. .. .. ..$ save_warmup       : logi FALSE
##   .. .. .. .. ..$ test_grad         : logi FALSE
```

```
##   .. .. .. .. ..$ thin              : int 2
##   .. .. .. .. ..$ warmup            : int 1000
##   .. .. .. ..- attr(*, "inits")= num [1:1006] -1.31 1.16 -1.46 1.21 1.92 ...
##   .. .. .. ..- attr(*, "mean_pars")= num [1:1006] -0.609 -1.054 -1.503 0.334 2.492 ...
##   .. .. .. ..- attr(*, "mean_lp__")= num -2367
##   .. .. .. ..- attr(*, "adaptation_info")= chr "# Adaptation terminated\n# Step size = 0.61
##   .. .. .. ..- attr(*, "elapsed_time")= Named num [1:2] 0.221 0.256
##   .. .. .. .. ..- attr(*, "names")= chr [1:2] "warmup" "sample"
##   .. .. .. ..- attr(*, "sampler_params")=List of 6
##   .. .. .. .. ..$ accept_stat__: num [1:500] 0.811 0.997 1 0.996 0.911 ...
##   .. .. .. .. ..$ stepsize__   : num [1:500] 0.612 0.612 0.612 0.612 0.612 ...
##   .. .. .. .. ..$ treedepth__  : num [1:500] 3 3 3 3 3 3 3 3 3 3 ...
##   .. .. .. .. ..$ n_leapfrog__ : num [1:500] 7 7 7 7 7 7 7 7 7 7 ...
##   .. .. .. .. ..$ divergent__  : num [1:500] 0 0 0 0 0 0 0 0 0 0 ...
##   .. .. .. .. ..$ energy__     : num [1:500] 2374 2368 2366 2369 2369 ...
##   .. .. .. ..- attr(*, "return_code")= int 0
##   .. ..$ chains     : int 2
##   .. ..$ iter       : int 2000
##   .. ..$ thin       : int 2
##   .. ..$ warmup     : int 1000
##   .. ..$ n_save     : num [1:2] 500 500
##   .. ..$ warmup2    : int [1:2] 0 0
##   .. ..$ permutation:List of 2
##   .. .. ..$ : int [1:500] 468 256 57 341 143 358 295 43 478 71 ...
##   .. .. ..$ : int [1:500] 392 226 291 225 316 498 459 367 212 472 ...
##   .. ..$ pars_oi    : chr [1:3] "beta" "sigma" "lp__"
##   .. ..$ dims_oi    :List of 3
##   .. .. ..$ beta : num 5
##   .. .. ..$ sigma: num(0)
##   .. .. ..$ lp__ : num(0)
##   .. ..$ fnames_oi  : chr [1:7] "beta[1]" "beta[2]" "beta[3]" "beta[4]" ...
##   .. ..$ n_flatnames: int 7
```

```
##   ..@ inits     :List of 2
##   .. ..$ :List of 3
##   .. .. ..$ beta : num [1:5(1d)] -1.928 -1.9 -0.547 -0.936 1.761
##   .. .. ..$ sigma: num 4.8
##   .. .. ..$ mu   : num [1:1000(1d)] -4.23 0.2 -2.39 -4.47 -4.64 ...
##   .. ..$ :List of 3
##   .. .. ..$ beta : num [1:5(1d)] -1.31 1.16 -1.46 1.21 1.92
##   .. .. ..$ sigma: num 0.587
##   .. .. ..$ mu   : num [1:1000(1d)] -3.72 -4.64 -4.09 -5.97 -1.11 ...
##   ..@ stan_args :List of 2
##   .. ..$ :List of 10
##   .. .. ..$ chain_id   : int 1
##   .. .. ..$ iter       : int 2000
##   .. .. ..$ thin       : int 2
##   .. .. ..$ seed       : int 20210329
##   .. .. ..$ warmup     : num 1000
##   .. .. ..$ init       : chr "random"
##   .. .. ..$ algorithm  : chr "NUTS"
##   .. .. ..$ save_warmup: logi FALSE
##   .. .. ..$ method     : chr "sampling"
##   .. .. ..$ control    :List of 1
##   .. .. .. ..$ adapt_delta: num 0.85
##   .. ..$ :List of 10
##   .. .. ..$ chain_id   : int 2
##   .. .. ..$ iter       : int 2000
##   .. .. ..$ thin       : int 2
##   .. .. ..$ seed       : int 20210329
##   .. .. ..$ warmup     : num 1000
##   .. .. ..$ init       : chr "random"
##   .. .. ..$ algorithm  : chr "NUTS"
##   .. .. ..$ save_warmup: logi FALSE
##   .. .. ..$ method     : chr "sampling"
```

```
##   .. .. ..$ control    :List of 1
##   .. .. .. ..$ adapt_delta: num 0.85
##   ..@ stanmodel :Formal class 'stanmodel' [package "rstan"] with 5 slots
##   .. .. ..@ model_name  : chr "37c095a6cbe3e8a27908a77406b6b94d"
##   .. .. ..@ model_code  : chr "data {\n  int<lower=1> N; // num. observations\n  int<lower=
##   .. .. .. ..- attr(*, "model_name2")= chr "37c095a6cbe3e8a27908a77406b6b94d"
##   .. .. ..@ model_cpp   :List of 2
##   .. .. .. ..$ model_cppname: chr "model3f785c5348a8_37c095a6cbe3e8a27908a77406b6b94d"
##   .. .. .. ..$ model_cppcode: chr "// Code generated by Stan version 2.21.0\n\n#include <sta
##   .. .. ..@ mk_cppmodule:function (object)
##   .. .. ..@ dso         :Formal class 'cxxdso' [package "rstan"] with 7 slots
##   .. .. .. .. ..@ sig         :List of 1
##   .. .. .. .. .. ..$ file3f7838657364: chr(0)
##   .. .. .. .. ..@ dso_saved   : logi TRUE
##   .. .. .. .. ..@ dso_filename: chr "file3f7838657364"
##   .. .. .. .. ..@ modulename  : chr "stan_fit4model3f785c5348a8_37c095a6cbe3e8a27908a77406b6
##   .. .. .. .. ..@ system      : chr "x86_64, mingw32"
##   .. .. .. .. ..@ cxxflags    : chr "CXXFLAGS = -O2 -Wall $(DEBUGFLAG) -mfpmath=sse -msse2 -
##   .. .. .. .. ..@ .CXXDSOMISC :<environment: 0x000002399847bc18>
##   ..@ date      : chr "Fri Mar 26 13:55:16 2021"
##   ..@ .MISC     :<environment: 0x0000023998e2b150>
```

Inference

For the sake of illustration, we use the replication data from Bischof and Wagner (2019), made available through the American Journal of Political Science Dataverse.

The original analysis uses Ordinary Least Squares estimation to gauge the effect of the assassination of the populist radical right politician Pim Fortuyn prior to the Dutch Parliamentary Election in 2002 on micro-level ideological polarization.

The outcome variable contains squared distances of respondents' left-right self-placement to the pre-election median self-placement of all respondents. The main predictor is a binary indicator whether the interview was conducted before or after Fortuyn's assassination.

Getting actual data

```
## Retrieve and manage data
bw_ajps19 <-
  read.table(
    paste0(
      "https://dataverse.harvard.edu/api/access/datafile/",
      ":persistentId?persistentId=doi:10.7910/DVN/DZ1NFG/LFX4A9"
    ),
    header = TRUE,
    stringsAsFactors = FALSE,
    sep = "\t",
    fill = TRUE
  ) %>%
  select(wave, fortuyn, polarization) %>% ### select relevant variables
  subset(wave == 1) %>%                   ### subset to pre-election wave
  na.omit()                               ### drop incomplete rows

## Define data
x <- model.matrix(~ fortuyn, data = bw_ajps19)
y <- bw_ajps19$polarization
N <- nrow(x)
K <- ncol(x)

## Collect as list
standat_inf <- list(
  N = N,
  K = K,
  x = x,
  y = y)
```

Inference

```
lm_inf <- rstan::sampling(
  lm_mod,                       # compiled model
  data = standat_inf,               # data input
  algorithm = "NUTS",           # algorithm
  control = list(               # control arguments
    adapt_delta = .85),
  save_warmup = FALSE,          # discard warmup sims
  sample_file = NULL,           # no sample file
  diagnostic_file = NULL,       # no diagnostic file
  pars = c("beta", "sigma"),    # select parameters
  iter = 2000L,                 # iter per chain
  warmup = 1000L,               # warmup period
  thin = 2L,                    # thinning factor
  chains = 2L,                  # num. chains
  cores = 2L,                   # num. cores
  seed = 20210329)              # seed
```

Posterior summaries

The original analysis reports point estimates (standard errors) of 1.644 (0.036) for the intercept and -0.112 (0.076) for the before-/after indicator.

How do our estimates compare?

```
print(lm_inf,
      pars = c("beta", "sigma"),
      digits_summary = 3L)
```

Model summary

```
## Inference for Stan model: 37c095a6cbe3e8a27908a77406b6b94d.
## 2 chains, each with iter=2000; warmup=1000; thin=2;
## post-warmup draws per chain=500, total post-warmup draws=1000.
##
##           mean se_mean    sd   2.5%    25%    50%    75% 97.5% n_eff  Rhat
## beta[1]  1.644   0.001 0.035  1.576  1.620  1.644  1.667 1.710   917 1.000
## beta[2] -0.113   0.002 0.074 -0.252 -0.166 -0.115 -0.063 0.037   949 1.004
## sigma    1.239   0.001 0.022  1.199  1.223  1.239  1.254 1.283   795 1.004
##
## Samples were drawn using NUTS(diag_e) at Fri Mar 26 13:55:44 2021.
## 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).
```

```r
# Extract posterior samples for beta[2]
beta2_posterior <- rstan::extract(lm_inf)$beta[, 2]

# Probability that beta[2] is greater than zero
mean(beta2_posterior > 0)
```

Hypothesis testing

```
## [1] 0.063
```

Convergence diagnostics

Generic diagnostics: `Rhat` and `n_eff`

1. $\hat{R} < 1.1$: Potential scale reduction statistic (aka Gelman-Rubin convergence diagnostic)
    - low values indicate that chains are stationary (convergence to target distribution within chains)

- low values indicate that chains mix (convergence to same target distribution across chains)

2. $\frac{n_{eff}}{n_{iter}} > 0.001$: Effective sample size
   - A small effective sample size indicates high autocorrelation within chains
   - This indicates that chains explore the posterior density very slowly and inefficiently

Algorithm-specific diagnostics

In the words of the developers:

"Hamiltonian Monte Carlo provides not only state-of-the-art sampling speed, it also provides state-of-the-art diagnostics. Unlike other algorithms, when Hamiltonian Monte Carlo fails it fails sufficiently spectacularly that we can easily identify the problems."

Source: https://github.com/stan-dev/stan/wiki/Stan-Best-Practices

- Divergent transitions after warmup (validity concern)
  - increase `adapt_delta` (target acceptance rate)
  - reparameterize/optimize your code
- Maximum treedepth exceeded (efficiency concern)
  - increase `max_treedepth`
- Diagnostics summary for `stanfit` object: `check_hmc_diagnostics(object)`
- For further information, see the Guide to Stan's warnings

Print summaries of generic and algorithm-specific diagnostics

Michael Betancourt, one of the core developers of Stan, has provided a utility function that performs all diagnostic checks at once:

```
# Source function directly from GitHub
devtools::source_url(
  paste0(
```

```
    "https://github.com/betanalpha/",
    "knitr_case_studies/blob/master/principled_bayesian_workflow/",
    "stan_utility.R?raw=TRUE"
  )
)
```

```
## i SHA-1 hash of file is "23fa85de7cef8bcef8c0c7389ddb6ab619a4087d"
```

```
# Apply function to stanfit object
check_all_diagnostics(lm_inf)
```

```
## n_eff / iter looks reasonable for all parameters
## Rhat looks reasonable for all parameters
## 0 of 1000 iterations ended with a divergence (0%)
## 0 of 1000 iterations saturated the maximum tree depth of 10 (0%)
## E-FMI indicated no pathological behavior
```

Visual diagnostics using shinystan

Note: `shinystan` launches a ShinyApp, which cannot be launched during an active `learnr` session.

```
library(shinystan)
launch_shinystan(lm_inf)
```

Additional Functionality:

- `generate_quantity()`: Add a new parameter as a function of one or two existing parameters
- `deploy_shinystan()`: Deploy a 'ShinyStan' app on shinyapps.io

Visual diagnostics using bayesplot

bayesplot offers a vast selection of visual diagnostics for `stanfit` objects:

- Diagnostics for No-U-Turn-Sampler (NUTS)

  - Divergent transitions
  - Energy
  - Bayesian fraction of missing information

- Generic MCMC diagnostics
  - $\hat{R}$
  - $\texttt{n}_{\texttt{eff}}$
  - Autocorrelation
  - Mixing (trace plots)

For full functionality, examples, and vignettes:

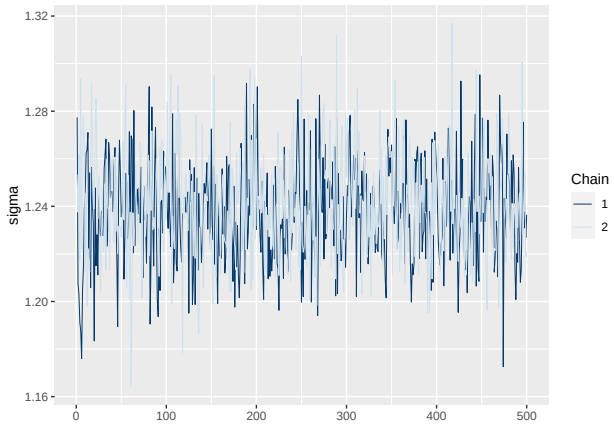
- GitHub Examples
- CRAN Vignettes
- `available_mcmc()` function

Example: Trace plot for sigma

```
library(bayesplot)


# Extract posterior draws from stanfit object
lm_post_draws <- extract(lm_inf, permuted = FALSE)


# Traceplot
mcmc_trace(lm_post_draws, pars = c("sigma"))
```


1.32
1.28
1.24
1.20
1.16
sigma
0
100
200
300
400
500
Chain
1
2