Assignment 6

anonymous

General information

AI used in learning how to program the Stan code.

Setup

The following installs and loads the `cmdstanr` package and tries to install `cmdstan`.

```
if(!require(cmdstanr)){
    install.packages("cmdstanr", repos = c("https://mc-stan.org/r-packa
ges/", getOption("repos")))
    library(cmdstanr)
}
```

```
## Loading required package: cmdstanr
```

```
## This is cmdstanr version 0.6.0
```

```
## - CmdStanR documentation and vignettes: mc-stan.org/cmdstanr
```

```
## - CmdStan path: /coursedata/cmdstan
```

```
## - CmdStan version: 2.33.0
```

```
##
## A newer version of CmdStan is available. See ?install_cmdstan() to i
nstall it.
## To disable this check set option or environment variable CMDSTANR_NO
_VER_CHECK=TRUE.
```

```
cmdstan_installed <- function(){
  res <- try(out <- cmdstanr::cmdstan_path(), silent = TRUE)
  !inherits(res, "try-error")
}
if(!cmdstan_installed()){
    install_cmdstan()
}

library(rstan)
```

```
## Loading required package: StanHeaders
```

```
##
## rstan version 2.26.23 (Stan version 2.26.1)
```

```
## For execution on a local, multicore CPU with excess RAM we recommend
 calling
## options(mc.cores = parallel::detectCores()).
## To avoid recompilation of unchanged Stan programs, we recommend call
ing
## rstan_options(auto_write = TRUE)
## For within-chain threading using `reduce_sum()` or `map_rect()` Stan
 functions,
## change `threads_per_chain` option:
## rstan_options(threads_per_chain = 1)

##
## Attaching package: 'rstan'

## The following object is masked from 'package:tidyr':
##
##     extract

## The following objects are masked from 'package:posterior':
##
##     ess_bulk, ess_tail

install.packages("posterior")

## Installing package into '/usr/local/lib/R/site-library'
## (as 'lib' is unspecified)

library(posterior)
```

Stan warm-up: linear model of BDA retention with Stan (2 points)

2(a)

The fixed code is below:

```
data {
    int<lower=0> N;
    vector[N] x;
    vector[N] y;
    int<lower=0> no_predictions;
    vector[no_predictions] x_predictions;
}

parameters {
    real alpha;
    real beta;
    real<lower=0> sigma;
}

transformed parameters {
```

```
    vector[N] mu = alpha + beta * x;
}

model {
    y ~ normal(mu, sigma);
}

generated quantities {
    vector[no_predictions] mu_pred = alpha + beta * x_predictions;
    vector[no_predictions] y_pred;
    for (i in 1:no_predictions) {
        y_pred[i] = normal_rng(mu_pred[i], sigma);
    }
}
```

The first error is in “real<upper=0>” which should be corrected to”real<lower=0>“,because sigma should be constrained to be positive.

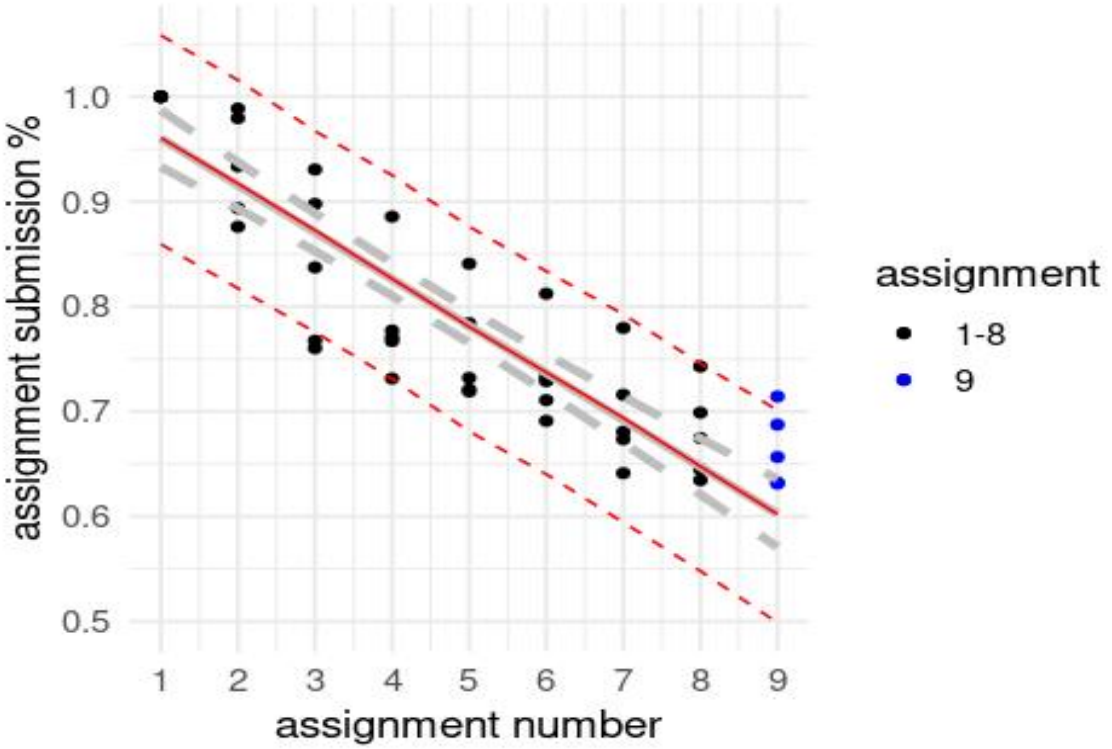
The second error is in “vector[N] mu = alpha + beta * x”, where should a semicolon at the end. Because every statement in Stan code should end with a semicolon.

The last error is in “array[no_predictions] real y_pred = normal_rng(mu, sigma);”, the corrected code should be”vector[no_predictions] y_pred = normal_rng(mu_pred, sigma);“. Because it should use mu_pred instead of mu to generate y_pred because mu_pred is the expected value calculated on the predictive covariate values x_predictions, while mu is calculated on the original x. In Stan, a vector is a more common and natural data structure to represent a series of values. Therefore, I replaced array with vector.

Plotting happens here:

```
ggplot() +
  # scatter plot of the training data:
  geom_point(
    aes(x, y, color=assignment),
    data=data.frame(x=assignment, y=propstudents, assignment="1-8")
) +
  # scatter plot of the test data:
  geom_point(
    aes(x, y, color=assignment),
    data=data.frame(x=no_assignments, y=propstudents9, assignment="9")
) +
  # you have to tell us what this plots:
  geom_line(aes(x,y=value,linetype=pct), data=mu_quantiles_df, color='g
rey', linewidth=1.5) +
  # you have to tell us what this plots:
  geom_line(aes(x,y=value,linetype=pct), data=y_quantiles_df, color='re
d') +
  # adding xticks for each assignment:
  scale_x_continuous(breaks=1:no_assignments) +
```

```
# adding labels to the plot:
labs(y="assignment submission %", x="assignment number") +
# specifying that line types repeat:
scale_linetype_manual(values=c(2,1,2)) +
# Specify colours of the observations:
scale_colour_manual(values = c("1-8"="black", "9"="blue")) +
# remove the legend for the linetypes:
guides(linetype="none")
```



2(b)

1. Solid Red Line: This line represents the median (or the 50th percentile) of the predictions. This suggests that, based on the posterior samples from the model, the value represented by this line is the most probable prediction.

Dashed Red Lines: These lines represent the 5% and 95% percentiles of the predictions. This implies that, according to the posterior samples of the model, we are 90% confident that the true value lies between these two dashed lines.

The red lines are different from the grey lines because the grey lines represent the 5%, 50%, and 95% quantiles of the predicted assignment submission proportions' mean values.

2.The student retention rate, measured by assignment submissions, shows a declining trend as the assignment number increases. This can be observed from the negative slope of the red solid line.

3.No, it falls short in forecasting the percentage of students turning in the final 9th assignment, likely due to the 9th assignment not having a linear correlation with the previous eight assignments.

4.We can consider using polynomial regression or tree-based models, such as random forests and gradient boosting, to improve the prediction.

Generalized linear model: Bioassay with Stan (4 points)

3(a)

The Rstan code is below:

```
data {
  int<lower=0> N;             // Number of data points (doses)
  vector[N] x;                // Dose amounts
  int<lower=0> n[N];          // Number of trials for each dose
  int y[N];                   // Number of successes (deaths) for each
dose
}

parameters {
  vector[2] theta;  // theta[1] = alpha, theta[2] = beta
}

model {
  vector[N] logit_p;

  // Priors
  theta ~ multi_normal([0, 10]', [[4, 12], [12, 100]]);  // Using the g
iven mu and Sigma

  // Likelihood
  for (i in 1:N) {
    logit_p[i] = theta[1] + theta[2] * x[i];
    y[i] ~ binomial_logit(n[i], logit_p[i]);
  }
}
```

The R code is below:

```
data("bioassay")
data_list <- list(
  N = 4,
  x = bioassay$x,
```

```
  n = bioassay$n,
  y = bioassay$y
)
sink(tempfile())
fit <- stan(file = 'bioassay_model.stan', data = data_list,verbose = FA
LSE)

sink()
print(fit)
```

```
## Inference for Stan model: anon_model.
## 4 chains, each with iter=2000; warmup=1000; thin=1;
## post-warmup draws per chain=1000, total post-warmup draws=4000.
##
##           mean se_mean   sd   2.5%   25%   50%   75% 97.5% n_eff Rha
t
## theta[1]  0.96    0.03 0.89  -0.66  0.34  0.93  1.54  2.79  1037 1.0
1
## theta[2] 10.56    0.15 4.78   3.45  7.05  9.81 13.36 21.71  1037 1.0
1
## lp__     -7.16    0.03 1.06 -10.03 -7.57 -6.84 -6.40 -6.14  1175 1.0
0
##
## Samples were drawn using NUTS(diag_e) at Sun Oct 15 11:53:04 2023.
## For each parameter, n_eff is a crude measure of effective sample siz
e,
## and Rhat is the potential scale reduction factor on split chains (at

## convergence, Rhat=1).
```

3(b)

```
fit_summary <- summary(fit)
alpha_summary <- fit_summary$summary["theta[1]",]
beta_summary <- fit_summary$summary["theta[2]",]
print("alpha summary")
```

```
## [1] "alpha summary"
```

```
print(alpha_summary)
```

```
##          mean       se_mean            sd          2.5%           2
5%
##    0.96346580    0.02768124    0.89128239   -0.66319185    0.3392585
1
##           50%           75%         97.5%         n_eff          Rha
t
##    0.93486268    1.54226438    2.79125934 1036.71573739    1.0055498
0
```

```
print("beta summary")
```

```
## [1] "beta summary"

print(beta_summary)

##         mean      se_mean           sd         2.5%          25%
      50%
##   10.5607404    0.1485458    4.7834399    3.4538966    7.0546417
9.8139759
##          75%        97.5%        n_eff         Rhat
##   13.3626940   21.7144520 1036.9548524    1.0087133

alpha_rhat <- fit_summary$summary["theta[1]", "Rhat"]
beta_rhat <- fit_summary$summary["theta[2]", "Rhat"]

print(paste("Rhat for alpha (theta[1]):", alpha_rhat))

## [1] "Rhat for alpha (theta[1]): 1.00554979733481"

print(paste("Rhat for beta (theta[2]):", beta_rhat))

## [1] "Rhat for beta (theta[2]): 1.00871332156418"
```

The Rhat value is part of the Gelman-Rubin diagnostic and is used to assess the convergence of MCMC (Markov Chain Monte Carlo) sampling. It measures the ratio of between-chain variability to within-chain variability. The common goal is to ensure that the Rhat value is close to 1, indicating that there is little difference in variation between different chains compared to within a single chain, which suggests that the sampling process has converged. In simple terms, the closer the Rhat value is to 1, the more likely it is that MCMC sampling has converged. If the Rhat value is significantly greater than 1, further diagnostic and adjustments to the sampling process may be needed.

From the Rhat values,

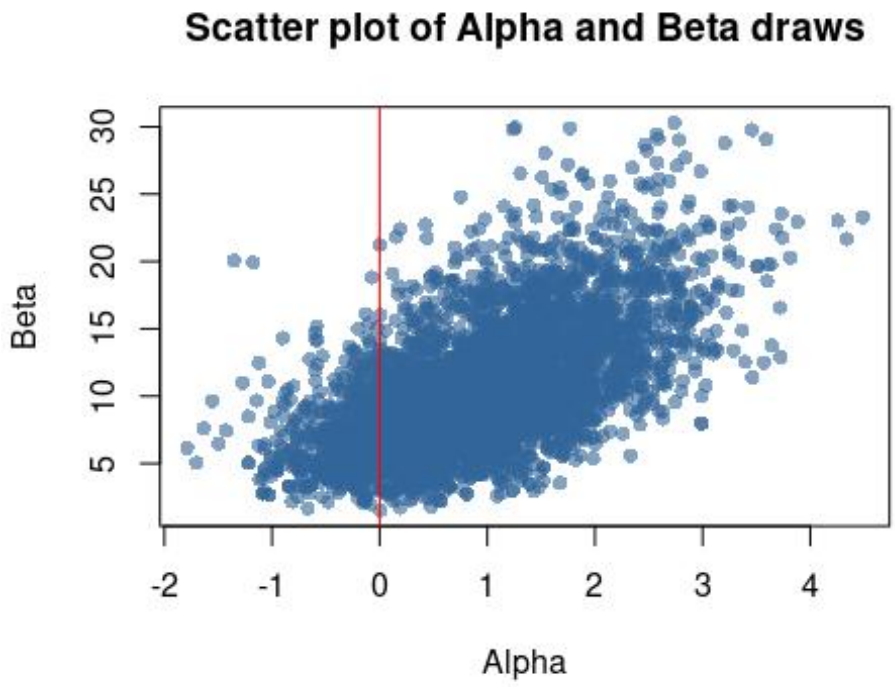
Rhat for alpha (theta[1]): `1.00554979733481`

Rhat for beta (theta[2]): `1.00871332156418`

We can confidently say that both chains for α and β have converged. This means that we can use the estimates from these posterior distributions with relative confidence.

3(c)

```
samples <- extract(fit)
alpha_draws <- samples$theta[,1]
beta_draws <- samples$theta[,2]
plot(alpha_draws, beta_draws, xlab = "Alpha", ylab = "Beta", main = "Sc
atter plot of Alpha and Beta draws", pch = 16, col = rgb(0.2,0.4,0.6,0.
6))
abline(h = 0, v = 0, col = "red")  # Adds lines for reference if needed
```



3(d)

1.jupyter.cs.aalto.fi

2.R

3.RStan

4.No, I didn't. Everything in the installation and compilation worked without any problem.

5.Perhaps the Stan developers can consider optimizing the execution speed for processing large data chains.