3a

Yiqiao Yin

1/25/2021

Homework: 3a

This homework let us continue with some interesting simulations.

References:

- https://mc-stan.org/cmdstanr/articles/cmdstanr.html
- https://mc-stan.org/docs/2__18/stan-users-guide/logistic-probit-regression-section.html

Write a program

Let me pursue this with regular *R* first.

```
# Define program
genY = function(alpha, beta, n, x, sigma) {
  error = rnorm(n, 0, sigma)
  return(y = 1/(alpha + beta*x) + error)
}
```

Next, let me code this in *Stan*.

```
library(cmdstanr)
```

```
## This is cmdstanr version 0.3.0

## - Online documentation and vignettes at mc-stan.org/cmdstanr

## - CmdStan path set to: C:/Users/eagle/Documents/.cmdstanr/cmdstan-2.25.0

## - Use set_cmdstan_path() to change the path

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

```
cmdstan_path()
```

```
## [1] "C:/Users/eagle/Documents/.cmdstanr/cmdstan-2.25.0"
```

```r
file <- file.path(cmdstan_path(), "examples", "model", "linearModel_3a.stan")
mod <- cmdstan_model(file)
```

```
## Model executable is up to date!
```

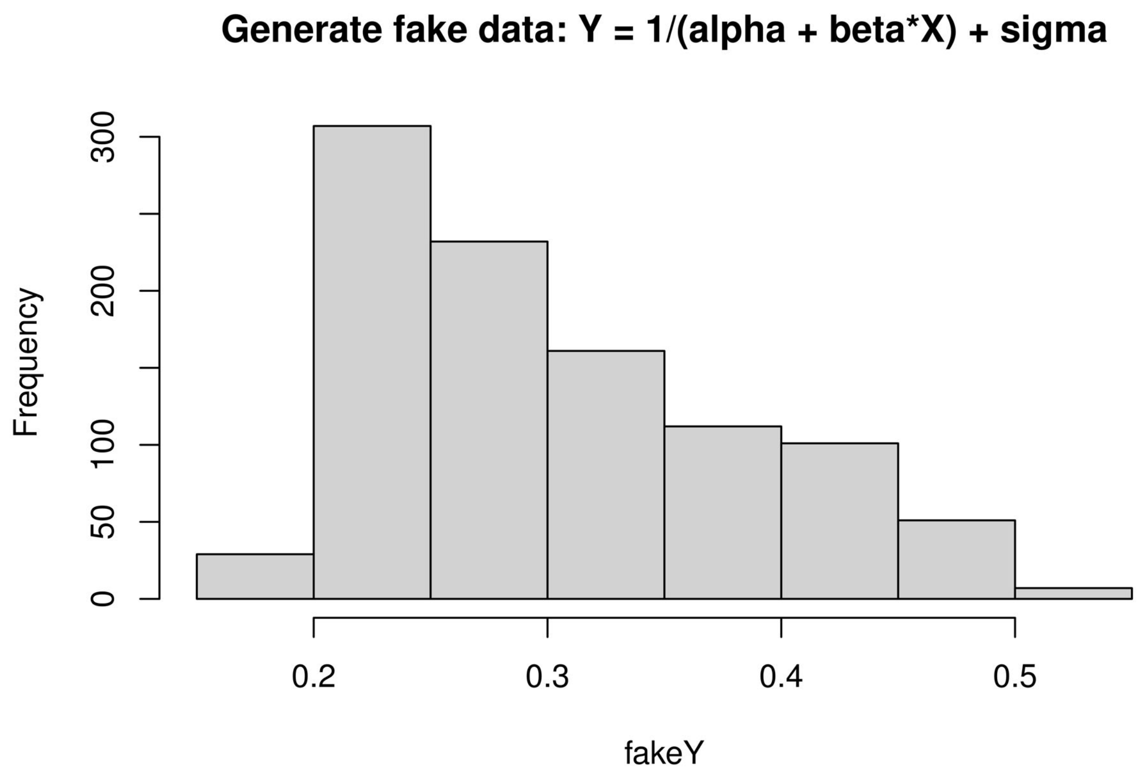
```r
mod$print()
```

```
## data {
##   int<lower=0> N;
##   vector[N] x;
##   vector[N] y;
## }
## parameters {
##   real alpha;
##   real beta;
##   real<lower=0> sigma;
## }
## model {
##   y ~ normal((alpha + beta*x).^(-1), sigma);
## }
```

In R, simulate fake data

In R, simulate fake data for this model with N=100, x uniformly distributed between 0 and 10, and a, b, sigma taking on the values 2, 3, 0.2.

```r
x = runif(1000, 0, 1)
fakeY = genY(alpha = 2, beta = 3, n = 1000, x = x, sigma = 0.01)
hist(fakeY, main = "Generate fake data: Y = 1/(alpha + beta*X) + sigma")
```



Fit the model

Fit the *Stan* model using your simulated data and check that the true parameter values are approximately recovered. Check also that you get approximately the same answer as from fitting a classical linear regression.

```
# we can check with linear model
summary(lm(fakeY~x))
```

```
##
## Call:
## lm(formula = fakeY ~ x)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.048714 -0.016006 -0.002891  0.013309  0.070382
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.441798   0.001357   325.6   <2e-16 ***
## x           -0.274179   0.002337  -117.3   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.02084 on 998 degrees of freedom
## Multiple R-squared:  0.9324, Adjusted R-squared:  0.9323
## F-statistic: 1.377e+04 on 1 and 998 DF,  p-value: < 2.2e-16
```

We can read off the estimated parameters that $a \approx 2$ and $b \approx 3$.

In *Stan*, we do the following

Next, let us write in *Stan*. Using *mod$sample()* function, we are able to generate MCMC simulation.

```
# names correspond to the data block in the Stan program
data_list <- list(N = 1000, x = x, y = fakeY)

fit <- mod$sample(
  data = data_list,
  seed = 123,
  chains = 1,
  parallel_chains = 8,
  refresh = 1000
)
```

```
## Running MCMC with 1 chain...
##
## Chain 1 Iteration:    1 / 2000 [  0%]  (Warmup)

## Chain 1 Informational Message: The current Metropolis proposal is about to be rejected because of the

## Chain 1 Exception: normal_lpdf: Scale parameter is 0, but must be > 0! (in 'C:/Users/eagle/AppData/Lo

## Chain 1 If this warning occurs sporadically, such as for highly constrained variable types like cova

## Chain 1 but if this warning occurs often then your model may be either severely ill-conditioned or mi

## Chain 1

## Chain 1 Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 1 Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 1 Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 1 finished in 0.4 seconds.
```

```
# check out the summary of the fit
fit$summary()
```

```
## # A tibble: 4 x 10
##   variable    mean  median      sd     mad      q5     q95  rhat ess_bulk
##   <chr>      <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl> <dbl>    <dbl>
## 1 lp__     4.07e+3 4.07e+3 1.23e+0 1.07e+0 4.07e+3 4.07e+3  1.00     404.
## 2 alpha    1.99e+0 1.99e+0 4.73e-3 4.91e-3 1.99e+0 2.00e+0  1.00     494.
## 3 beta     3.01e+0 3.01e+0 1.38e-2 1.28e-2 2.98e+0 3.03e+0  1.00     448.
## 4 sigma    1.03e-2 1.03e-2 2.19e-4 2.10e-4 9.93e-3 1.07e-2  1.00     723.
## # ... with 1 more variable: ess_tail <dbl>
```

We can observe from the above table in *fit&summary()* we have mean of alpha to be approximately 2, mean of beta to be approximately 3, and mean of sigma to be approximately 0.2. This corresponds to the results coming from classical linear regression in *R*.

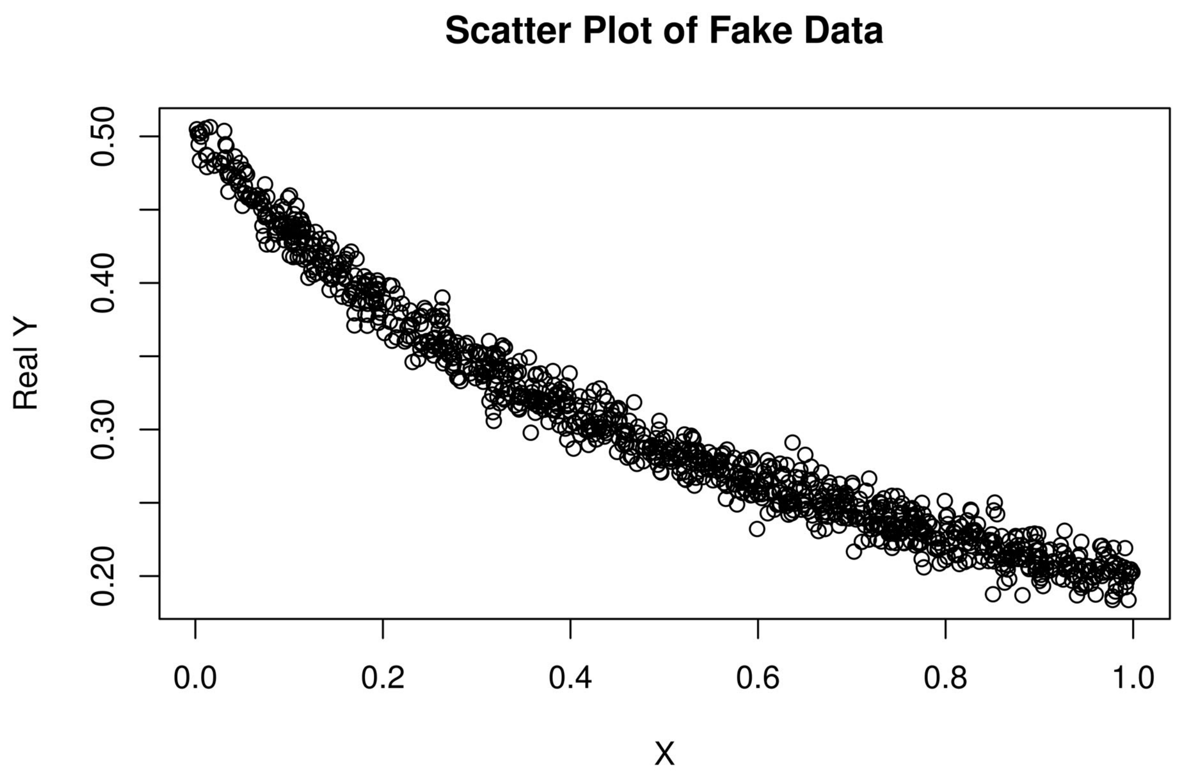
Comment after discussion with Charles:

I made some edits above. I am still unclear what is the tuning parameter *chain* is doing inside of the package. However, a change of this parameter dropped the standard deviation of the mean estimates of the parameters in the model. In addition, we observe $\hat{R}$ is much smaller. Here the $\hat{R}$ is about 1 for the parameters though the last homework I submitted these values are around 3-4 which indicates this trial has mixing chains.

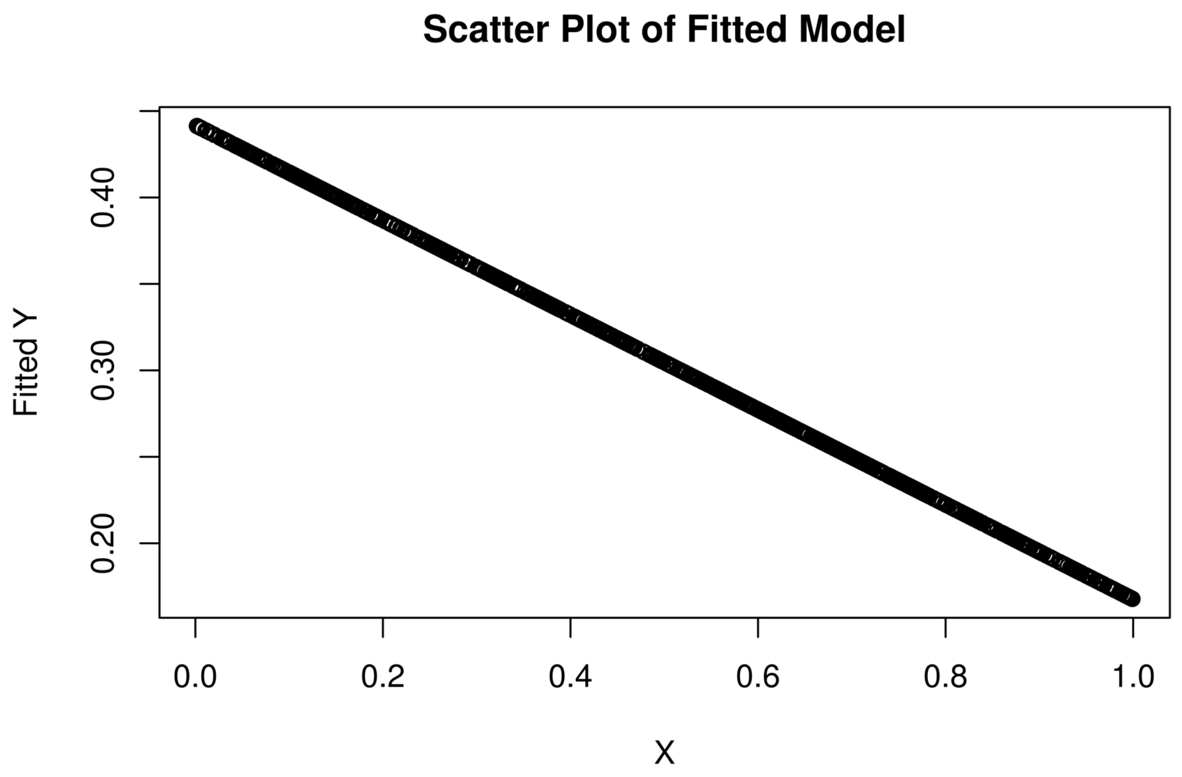
Make a single graph

Make a single graph showing a scatterplot of the simulated data and the fitted model.

```r
plot(x, fakeY, main = "Scatter Plot of Fake Data",
     xlab = "X", ylab = "Real Y")
```



```r
plot(x, predict(lm(fakeY~x), data.frame(x)), main = "Scatter Plot of Fitted Model",
     xlab = "X", ylab = "Fitted Y")
```



Report

Report on any difficulties you had at any of the above steps.

It took a while to read through the documentation of *Stan*, but after some discussion with classmates it is quite clear. However, I have the following thoughts:

I am not sure what the motivation of using *Stan* is. After carrying out such approach using *Stan*, I understand that the philosophy is to develop a pipeline with ingredients and recipes being user friendly and then the kitchen automatically cooks amazing meal! (In this analogy, the kitchen is *Stan* and since *Stan* compiles *C++* the selling point is that it's "faster' '.)

However, in computer science knowledge, a pipeline is convincing if it has more optimal performance in time / space complexity. For example, engineers present two pipelines: (A) program A has time of $O(n)$ and space of $O(n)$, and (B) program B has time of $O(n^2)$ and space of $O(\exp(n))$. Then obviously (A) is more optimal. I am unclear *Stan* survives this measurement comparing with *apply()* in *R* (also compiled from *C++*), and *numpy*, *random* or *tensorflow* in *Python*.

Comment after speaking with Charles:

Let me recap the misunderstanding here. The *Stan* package does different things than other libraries in *R* or *Python*. Another thing I want to add is that *Stan* efficiently packs a list of output together and prints comments. This has never been efficiently done in Bayesian workflow in other packages.