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
```

```
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!
```

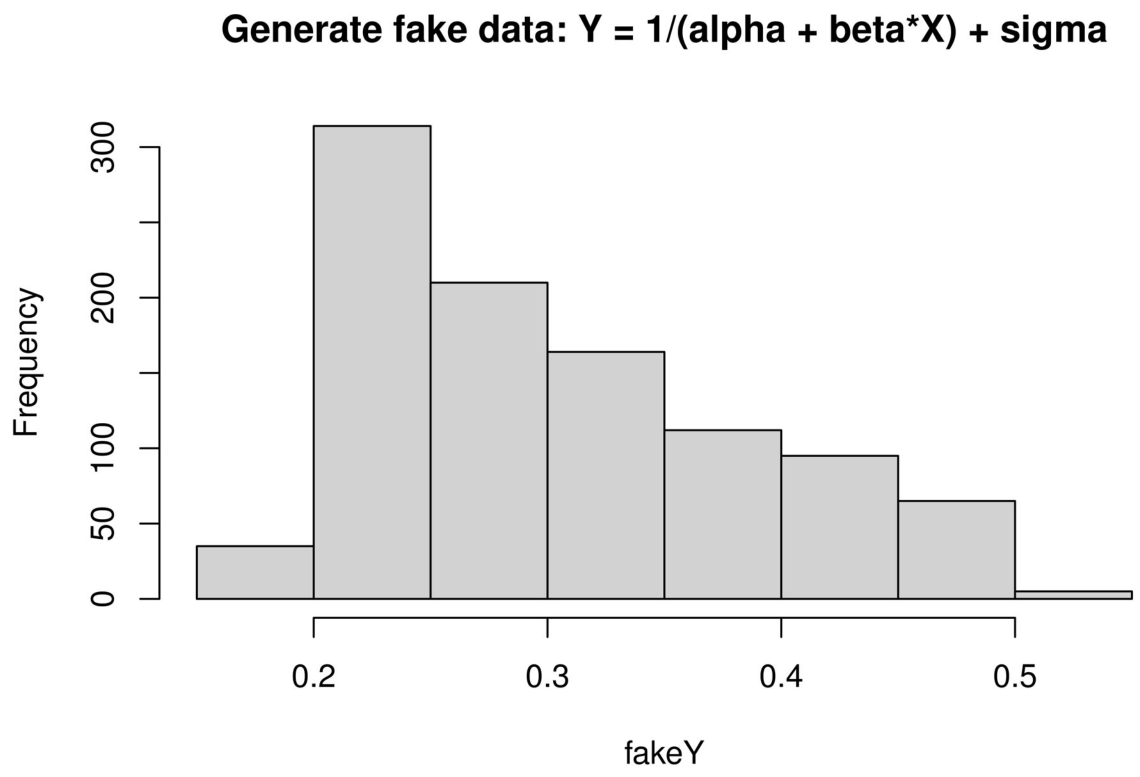
```r
mod$print()
```

```
## 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
## -0.047561 -0.016353 -0.003799  0.014877  0.072341
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.441269   0.001367   322.7   <2e-16 ***
## x           -0.271831   0.002342  -116.1   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.02144 on 998 degrees of freedom
## Multiple R-squared:  0.931,  Adjusted R-squared:  0.931
## F-statistic: 1.347e+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.

```r
# 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 = 4,
  parallel_chains = 8,
  refresh = 500
)
```

```
## Running MCMC with 4 chains, at most 8 in parallel...
##
## 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 2 Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 3 Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 4 Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 1 Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 1 Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 1 Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 2 Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 4 Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 4 Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 4 Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 1 Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 1 Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 4 Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 1 finished in 0.4 seconds.
## Chain 4 Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 4 finished in 0.5 seconds.
## Chain 2 Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 2 Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 2 Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 3 Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 2 Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 2 finished in 9.1 seconds.
## Chain 3 Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 3 Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 3 Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 3 Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 3 finished in 34.8 seconds.
##
## All 4 chains finished successfully.
## Mean chain execution time: 11.2 seconds.
## Total execution time: 35.0 seconds.
```

```
##
## Warning: 17 of 4000 (0.0%) transitions ended with a divergence.
## This may indicate insufficient exploration of the posterior distribution.
## Possible remedies include:
##   * Increasing adapt_delta closer to 1 (default is 0.8)
##   * Reparameterizing the model (e.g. using a non-centered parameterization)
##   * Using informative or weakly informative prior distributions
```

```
## 1000 of 4000 (25.0%) transitions hit the maximum treedepth limit of 10 or 2^10-1 leapfrog steps.
## Trajectories that are prematurely terminated due to this limit will result in slow exploration.
## Increasing the max_treedepth limit can avoid this at the expense of more computation.
## If increasing max_treedepth does not remove warnings, try to reparameterize the model.
```

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

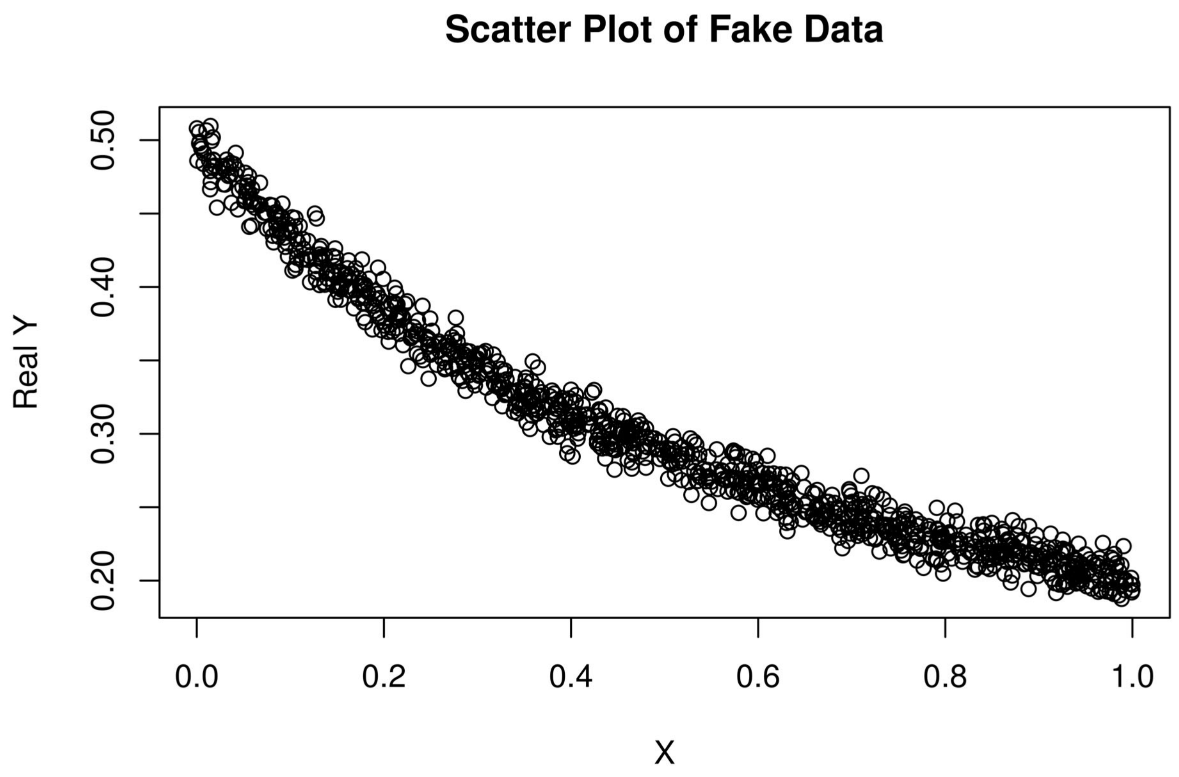
```
## # A tibble: 4 x 10
##   variable      mean  median       sd     mad        q5      q95  rhat ess_bulk
##   <chr>        <dbl>   <dbl>    <dbl>   <dbl>     <dbl>    <dbl> <dbl>    <dbl>
## 1 lp__     -2.15e+ 8 657.    3.72e+ 8 2.57e+3 -8.67e+ 8 4.12e+ 3  3.20     4.48
## 2 alpha    -3.05e+18   1.21  6.27e+18 2.63e+2 -2.00e+19 2.01e+ 0  4.60     4.54
## 3 beta      6.47e+17   3.00  1.78e+18 2.00e+3 -3.29e+17 5.16e+18  2.97     9.82
## 4 sigma     2.15e- 1   0.258 1.24e- 1 8.22e-2  9.63e- 3 3.24e- 1  2.68     5.21
## # ... 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*.

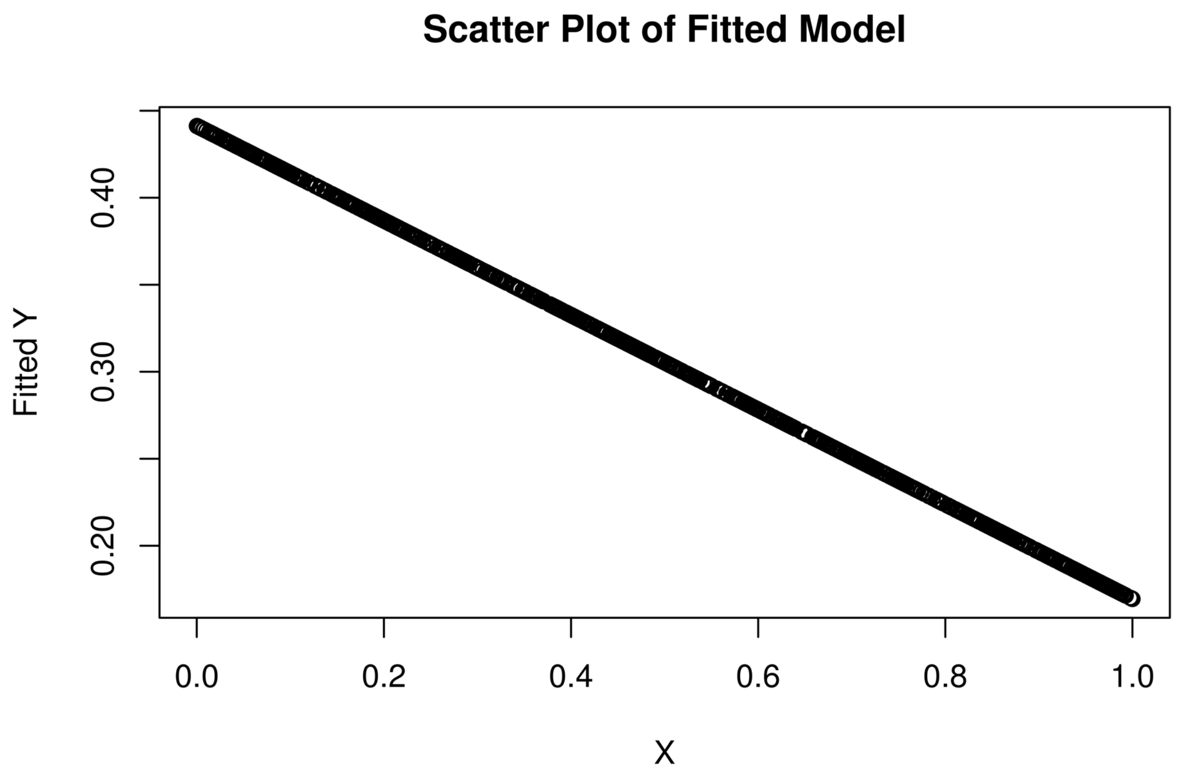
Make a single graph

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")
```



```
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*.