Applied Probabilistic Programming & Bayesian Machine Learning

§1-1: Getting Started with Stan

MIT IAP 2017 Max W. Shen

1 Hierarchical Models in Stan

During this section, you will write code to simulate a **hierarchical model**, then run Stan code to perform **inference** on the hierarchical model. We will emphasize the basics of interacting with Stan. This exercise is done in the command-line to match your natural development environment, to make it as easy as possible for you to apply Stan to your own data and play around with all aspects of the model.

Within this folder you find three files: hierarchical_model_1-1.py, hierarchical_model_1-1.R, and hierarchical_model_1-1.stan. You are free to use either python or R.

1.1 Defining an Example Problem

Let's say we have a normal distribution with mean 0 and standard deviation 1. We can draw samples from it, so let's draw d samples. What can we say about these D samples?

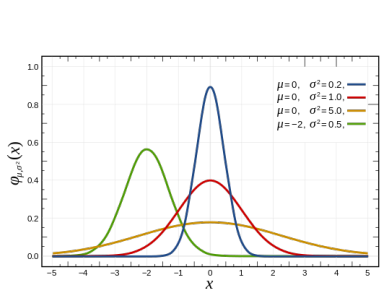
Well, intuitively, it's pretty likely that the average of the D samples will be close to 0. And since the standard deviation is 1, a lot of samples (roughly 68%) will fall between -1 and 1, and roughly 95% of the samples will fall between -2 and 2.

1.1.1 Making things more complicated

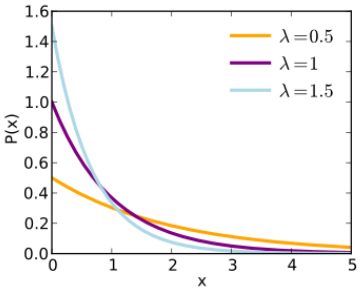
Let's ramp the complexity up a bit: What if we had N different normal distributions, each with a different mean and standard deviation? If we let i index $1, ..., N$, then we can say we have N normal distributions, each with mean μ_i and standard deviation σ_i.

Now, let's say we have an exponential distribution, which is parameterized by some λ, called the "rate". The exponential distribution describes the probability of waiting some amount of time x between events that occur randomly at some average rate λ. Intuitively, if the rate $\lambda = 0.1$, then we will wait 10 time units between events on average.

Let's draw N samples from this exponential distribution and call them $\mu_1, \mu_2, ..., \mu_N$, and let's use these as the means for our normal distribution.



(a) Normal Distribution, mean μ and standard deviation σ



(b) Exponential Distribution, rate λ

Figure 1: Probability Distributions

1.1.2 In summary...

Now, we'll write everything in a more concise manner: We draw N samples, called μ_1, ..., μ_N, from an exponential distribution parameterized by a rate $\lambda = 0.1$ or alternatively a scale $\beta = \frac{1}{\lambda} = 10$. For each of the N samples, we draw d samples from a normal distribution centered at sample μ_i, with $\sigma_i = 1$. (We'll keep all the standard deviations the same at 1 for now.)

Is it possible to infer the true value of λ from only looking at the data?

1.1.3 Concrete Example

Concretely, one could imagine that we have d measurement devices which measured n waiting times between events of interest. The exponential distribution is a natural way to model waiting times between events that occur independently at some constant average rate.

The measurement devices are known to be unbiased but have some unknown error each time they make a measurement, so one natural way to model measurement is using a normal distribution with $\mu = 0$ and some σ. The quantity of interest is the parameter λ whose true value is 0.1 (arbitrary units). The true measurement error is distributed by a normal distribution with standard deviation (σ) 1.0 (arbitrary units).

1.2 Writing down a model

We define more notation so we can write this model down mathematically. Here, we call our observations s for "samples". Let i index $1, ..., N$ for the N waiting times, and let j index $1, ..., D$ for the D measurement devices. Then, we can write s_{ij} to represent the observation made by the j-th measurement device for the i-th waiting time.

Now, we can express this model as:

$$\begin{aligned} s_{ij} &\sim normal(\mu_i, \sigma = 1) \\ \mu_i &\sim exponential(\lambda = 0.1) \end{aligned} \tag{1}$$

Where in model 1, the squiggly tilde's represent "distributed by". Translated into English, it would read,

> "For all i and all j, the observation s_{ij} is distributed by a normal distribution with unknown mean μ_i and known standard deviation $\sigma = 1$. Furthermore, the random variable μ_i is distributed by an exponential distribution with known parameter $\lambda = 0.1$."

This describes the process used to generate our observations.

Figure 2 presents a "tree diagram" that highlights the flow of information through the model. Note that the parameters μ_i define a distribution (with σ), but that μ_i also follow a distribution. This makes our model hierarchical.

There is no limit to hierarchical models - a distribution can be placed on any parameter. We could make a 3-layer model by placing a distribution on λ, which is precisely what we will do later.

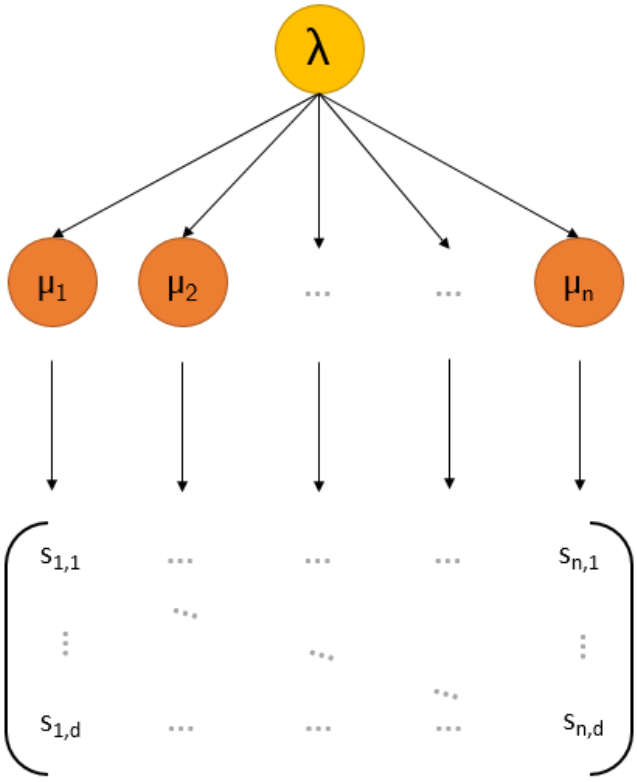


Figure 2: Rough sketch of the hierarchical model, ignoring the shared σ

2 hierarchical_model_1-1.stan

Our problem of interest is, if we only see the observations s, can we learn the true value of λ?

Based off of model 1, we can write this mathematically as:

$$
\begin{aligned}
s_{ij} &\sim normal(\mu_i, \sigma) \\
\mu_i &\sim exponential(\lambda) \\
\sigma &\sim \text{unknown-distribution}() \\
\lambda &\sim \text{unknown-distribution}()
\end{aligned}
\tag{2}
$$

In English, this would read:

> "For all i and all j, the observation s_{ij} is distributed by a normal distribution with unknown mean μ_i and unknown standard deviation σ. Furthermore, the random variable μ_i is distributed by an exponential distribution with unknown parameter λ."

For completeness, we write that σ and λ are distributed according to some unknown distributions. Sometimes, models do not explicitly write this, but σ and λ must have some distribution, implicit or explicit. Commonly, if we don't know anything about σ or λ, it makes sense to place a uniform distribution on them.

We now have a properly defined problem - let's see how we can write this in Stan to solve it.

```
data {
  int N;
  int D;
  matrix[N,D] observations;
}
```

For much of the class, we will be using pystan/rstan to call Stan code from python/R. In the data section, Stan receives the variables N, D, and *observations* from python/R.

```
parameters {
  real<lower=0> exponential_scale;
  real<lower=0> normal_mus[N];
  real<lower=0> normal_sigma;
}
```

In the parameters section, we define unknown quantities that we wish to infer through Stan. Here, *exponential_scale* and *normal_sigma* are declared as real numbers, with the constraint that they are non-negative. The *normal_mus*$[N]$ syntax defines an N-dimensional vector. Syntax for defining upper bounds looks like this:

```
  real<lower=0, upper=1> parameter;
```

Placing bounds on variables reflects mathematical definitions and prior knowledge. Mathematically, the normal distribution is defined using a non-negative standard deviation *normal_sigma*, so we reflect this in the model. Similarly, the exponential distribution is defined for a positive λ and has support for $x \in [0, \infty)$, which constrains *exponential_scale* and *normal_mus*$[N]$.

Now, the model itself:

```
model {
  for (n in 1:N) {
```

```
    observations[n] ~ normal(normal_mus[n], normal_sigma);
  }
  normal_mus ~ exponential(exponential_scale);
  exponential_scale ~ normal(0, 100);
  normal_sigma ~ normal(0, 100);
}
```

This corresponds to the model:

$$\begin{aligned} s_{ij} &\sim N(\mu_i, \sigma) \\ \mu_i &\sim exp\Big(\frac{1}{\beta}\Big) \\ \sigma &\sim N(0, 100) \\ \beta &\sim N(0, 100) \end{aligned} \tag{3}$$

where we have used the alternative parameterization $\beta = \frac{1}{\lambda} = exponential\_scale$ for the exponential distribution.

The d-dimensional vector $observations[n]$ is normally distributed with real number mean, $normal\_mus[n]$, and real number standard deviation $normal\_sigma$.

The n-dimensional vector $normal\_mus$ is exponentially distributed with scale $exponential\_scale$.

We place weakly-informative priors on $exponential\_scale$ and $normal\_sigma$ to indicate a lack of knowledge about them. Notice that a normal distribution with standard deviation 100 looks similar to a uniform distribution, while encoding some weak information that both parameter's true value is probably not on the order of -1,000,000 or 1,000,000.

Inference runs much faster (discussed in more depth in lecture 2) when using a weakly informative distribution with long tails rather than a uniform distribution because an unconstrained uniform distribution requires exploration of infinite parameter space.

Notice that we have placed a distribution over $exponential\_scale = \beta = \frac{1}{\lambda}$. This can be described as a hyperprior. Our model can be said to have 3 layers, though only 2 layers have parameters we are trying to estimate.

3 hierarchical_model_1-1.py

This python script consists of two functions: inference() and simulate(), related in the following manner:

```
def simulate():
  ...
  dataset = {
    'N': N,
    'D': D,
    'observations': observations,
  }
```

```
  return dataset

...

def main():
  dataset = simulate()
  inference(dataset)
  return
```

Inference takes in a *dataset* and calls Stan, where *dataset* is a dictionary of objects. The names and types of the objects must **match exactly** with Stan's received input in the data block:

```
data {
  int N;
  int D;
  matrix[N,D] observations;
}
```

It is convenient to use a numpy array for the matrix of observations. You can instantiate a matrix of zeros using:

```
import numpy as np
...
matrix = np.zeros((num_rows, num_columns))
```

4 hierarchical_model_1-1.R

This R script simulates data then passes it to RStan.

The intermediary *dataset* object,

```
dataset <- list(N = N,
                D = D,
                observations = obs);
```

constructs a list of objects that is passed to Stan. The names and types of the objects must **match exactly** with Stan's received input in the data block:

```
data {
  int N;
  int D;
  matrix[N,D] observations;
}
```

Exercise 1

Writing Simulation Code

Write code to simulate data in either python or R.

We suggest the variable values: N = 15, D = 30, $exponential\_rate = 0.1$ (scale = 10), $normal\_sigma = 1$. Take special case to pass the right parameter in when simulating from the exponential distribution - some libraries use rate, some use scale.

For python users, this link may be useful:

`https://docs.scipy.org/doc/numpy/reference/routines.random.html`

For R users, this may be useful:

`http://www.statmethods.net/advgraphs/probability.html`

5 Interacting with Stan

Running the code will likely give warning messages about Metropolis proposals being rejected. These warnings typically aren't an issue unless they occur after the chains have warmed up. However, if you receive an excessive amount (hundreds) before warmup, your model probably has some issues impacting inference.

A successful run will show something like this:

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
exponential_scale	0.07	2.3e-4	0.02	0.04	0.06	0.07	0.08	0.1	5292.0	1.0
normal_mus[0]	16.36	2.2e-3	0.18	15.99	16.24	16.35	16.48	16.71	7200.0	1.0
normal_mus[1]	3.02	2.6e-3	0.18	2.65	2.91	3.03	3.15	3.38	4857.0	1.0
normal_mus[2]	18.49	2.1e-3	0.18	18.15	18.37	18.49	18.62	18.85	7200.0	1.0
normal_mus[3]	21.59	2.1e-3	0.18	21.24	21.46	21.59	21.71	21.94	7200.0	1.0
normal_mus[4]	27.82	2.2e-3	0.18	27.46	27.69	27.81	27.94	28.17	7200.0	1.0
normal_mus[5]	8.1	2.4e-3	0.18	7.73	7.98	8.1	8.22	8.45	5804.0	1.0
normal_mus[6]	23.88	2.1e-3	0.18	23.53	23.76	23.88	24.0	24.23	7200.0	1.0
normal_mus[7]	29.45	2.2e-3	0.18	29.09	29.33	29.45	29.58	29.81	7200.0	1.0
normal_mus[8]	1.21	2.8e-3	0.18	0.86	1.09	1.21	1.33	1.56	4041.0	1.0
normal_mus[9]	16.01	2.2e-3	0.18	15.65	15.88	16.01	16.13	16.36	7200.0	1.0
normal_mus[10]	4.52	2.9e-3	0.18	4.16	4.4	4.53	4.64	4.88	4006.0	1.0
normal_mus[11]	0.29	2.6e-3	0.16	0.03	0.17	0.28	0.39	0.62	3537.0	1.0
normal_mus[12]	6.32	2.7e-3	0.18	5.96	6.2	6.32	6.44	6.68	4618.0	1.0
normal_mus[13]	54.92	2.2e-3	0.19	54.55	54.79	54.91	55.04	55.28	7200.0	1.0
normal_mus[14]	6.59	2.1e-3	0.18	6.24	6.47	6.59	6.71	6.94	7200.0	1.0
normal_sigma	1.0	4.3e-4	0.03	0.94	0.98	1.0	1.03	1.07	6284.0	1.0
lp__	-253.2	0.05	2.96	-259.7	-255.0	-252.9	-251.1	-248.3	3169.0	1.0

The last two columns are particularly important. **Rhat** is a measure of inference convergence, with 1.0 meaning full convergence. Convergence is linked to **n_eff**, the effective number of iterations for inference on that parameter. Some parameters are harder to estimate than others, such as the parameters at the "top" of a hierarchical model with many layers. If your model has unconverged parameters (rhat $\neq 1.0$), n_eff can be increased by increasing the number of iterations, increasing the number of chains, or respecifying your model if n_eff is a small fraction of the total number of iterations run.

The posterior distributions of the quantities of interest are summarized by the mean, standard deviation, and varying quantiles. We can see that *exponential_scale* (true value = 0.10) is not too far off at 0.07. However, the standard deviation is 0.02. What is a change to the simulated data you could make to improve inference of *exponential_scale*? What is a change to the model or the priors that would improve inference of *exponential_scale*?

The final row, **lp__**, is the log of the total probability density posterior of the model up to a constant factor. It can loosely be used to compare different models (discussed in a later lecture). Notably, lp__ can be positive, unlike the log likelihood.

You may notice that our n_eff is larger than yours, that's simply because we ran our model with more iterations.

5.1 The Fit Object in Python

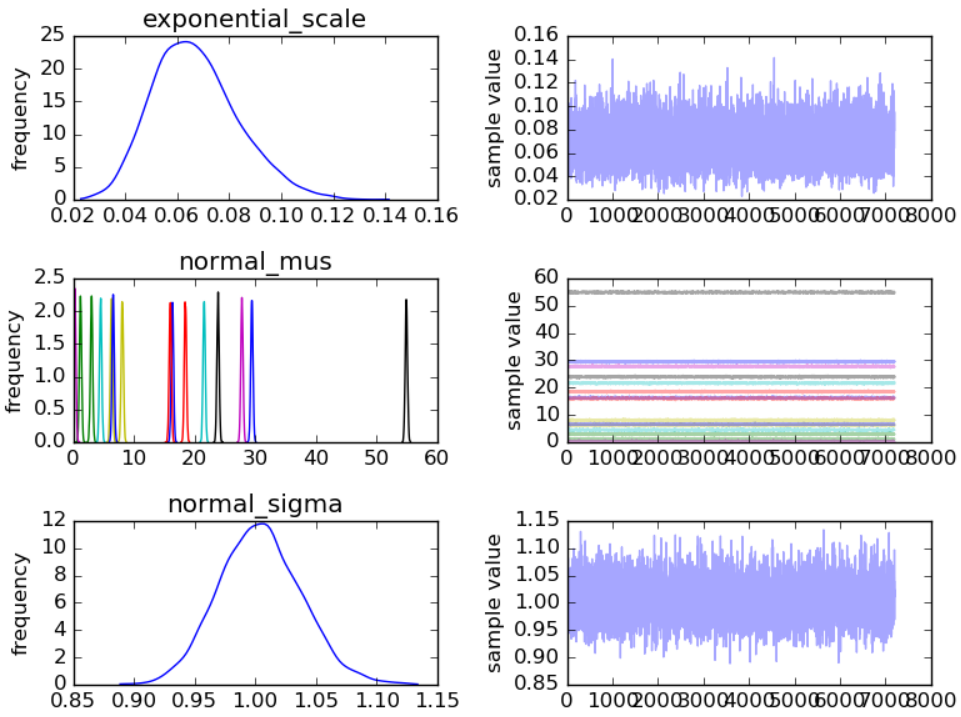


Figure 3: fit.png

The final model is returned by Stan and placed into a variable called "fit" in python. **Plotting the "fit"** object produces figure 3, which graphically depicts the full posterior distributions for each of the estimated parameters. The three plots on the right side of figure 3 describe the parameter space explored during inference (x-axis is 8000 iterations).

You can extract the sampled values for *normal_mu* from "fit" (here, we have 15 such values) using:

```
normal_mus = fit['normal_mus']
```

"fit$normal_mus" is a matrix that stores the inferred values of normal_mus for each iteration of

inference. To get the posterior mean, you would average over all of the iterations. We recommend randomly shuffling the matrix since consecutive samples are correlated due to MCMC/HMC inference (discussed more next lecture).

The **summary table** in the previous section, with n_eff and Rhat, comes from the command "print(fit)".

```
fit = pystan.stan(file = STAN_FN,
                  data = dataset,
                  iter = NUM_ITER,
                  warmup = WARMUP,
                  chains = NUM_CHAINS,
                  n_jobs = NUM_CORES)
print(fit)
```

The "fit" object can be saved to file by **pickling "fit.extract()"**. We choose to defer further discussion of pickling to Google.

5.2 The Fit Object in R

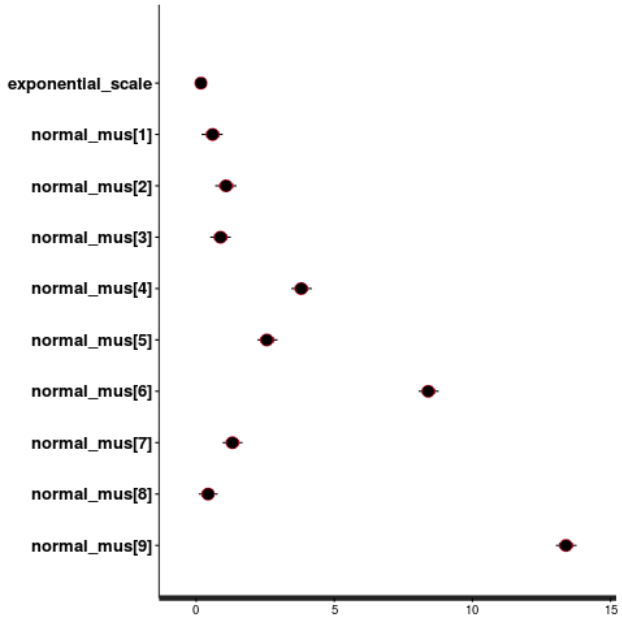


Figure 4: R's plot(fit)

The final model is returned by Stan and placed into a variable called "fit" in R. **Plotting the "fit"** object produces figure 4, which graphically summarizes the full posterior distributions for each of the estimated parameters. With default settings, only 10 parameters are shown. Equivalently, the command "stan_plot(fit)" produces the same plot.

Figure 5 comes from the command:

```
stan_trace(fit);
```

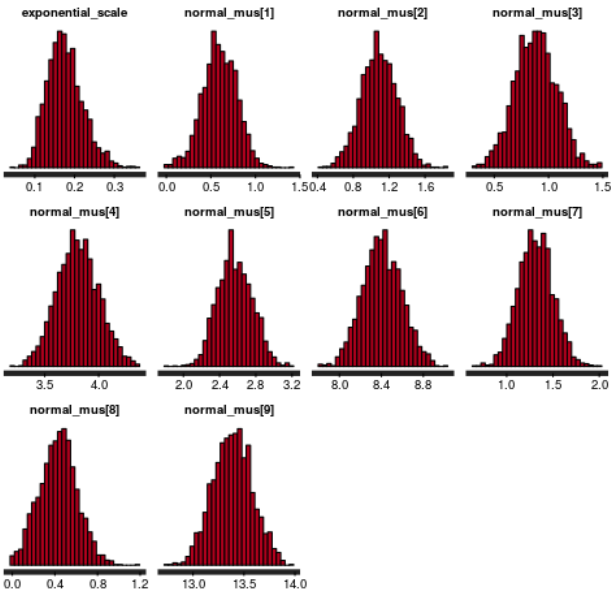


Figure 5: R's stan_hist(fit)

and depicts histograms of the posterior distributions.

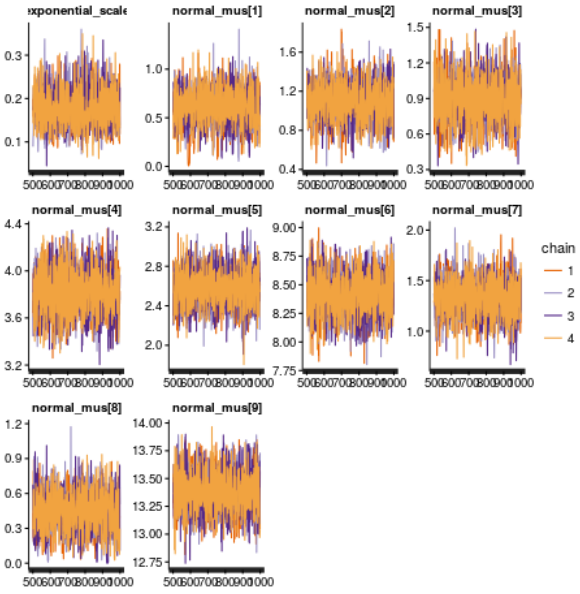


Figure 6: R's stan_trace(fit)

Figure 6 comes from the command:

```
stan_trace(fit);
```

This plots the regions of parameter space explored across all inference iterations. For more settings and plots, see the rstan package manual at

```
https://cran.r-project.org/web/packages/rstan/rstan.pdf
```

You can extract the sampled values for *normal_mu* from "fit" (here, we have 15 such values) using:

```
fe <- extract(fit, permuted = TRUE);
mus <- fe$normal_mus;
```

The "permuted" flag in extract keeps only the sampling draws after warmup, and permutes them to avoid issues of autocorrelation arising from MCMC/HMC (discussed more next lecture). By default, "fit$normal_mus" is a matrix that stores the inferred values of normal_mus for each iteration of inference. To get the posterior mean, you would average over all of the iterations.

The **summary table** in the previous section, with n_eff and Rhat, comes from the command "print(fit);".

```
fit <- stan(file = 'hierarchical_model_1-1.stan',
            data = dataset,
            iter = 1000,
            chains = 4);
print(fit);
```

The "fit" object can be saved to file with "save" and "load", commonly used to save the entire workspace. We defer further discussion to Google.

5.3 Coding in Stan

Comments in Stan are marked with "//" or "#".

An example print statement looks like this:

```
  print("y=", y, " z=", z);
```

For more information on commenting and printing, refer to page 32 of the Stan reference manual.

Exercise 2

In reality, all models are wrong, but some are useful. Try an incorrect model - replace the exponential distribution with a normal distribution. Do you expect the estimate of the mean to be biased, and if so, in which direction? Run inference to find the posterior distributions for the mean and standard deviation.

If your posterior distribution varies a lot over separate runs, you may consider adjusting N and D in your simulated data for more stable inference.

How different is your model's confidence in its estimate of the average waiting time under the correct model vs. the incorrect model?

Exercise 3

In the parameters block of the Stan file, try removing some constraints.

```
parameters {
  real<lower=0> exponential_scale;
  real<lower=0> normal_mus[N];
  real<lower=0> normal_sigma;
}
```

What kind of errors do you get, if any?

Exercise 4

What do you think will happen to inference on *exponential_scale* if you increase *normal_sigma*?

6 Bibliography and Additional Resources

To go beyond normal and exponential distributions, refer to sections V and VI in the Stan reference at http://mc-stan.org/documentation/ for a list of all probability distributions that Stan supports.

The RStan package manual is available at:

`https://cran.r-project.org/web/packages/rstan/rstan.pdf`

The pystan API is available at:

`https://pystan.readthedocs.io/en/latest/api.html`