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 two files: hierarchical_model_1-1.py, and hierarchical_model_1-1.stan.

## 1.1 Defining the Problem

Consider the following scenario: We draw $n$ samples, called $\mu_1$, ..., $\mu_n$, from an exponential distribution parameterized by a rate $\lambda = 10$ or alternatively a scale $\beta = \frac{1}{\lambda} = 0.1$. For each of the $n$ samples, we draw $d$ samples from a normal distribution centered at sample $\mu_i$, with $\sigma = 1$.

Is it possible to infer $\lambda$ from only the data?

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 following a Poisson process. The measurement devices are known to be unbiased but have some unknown error, so one natural way to model measurement is using a normal distribution. The quantity of interest is the average waiting time between events, whose true value is 10 (arbitrary units). The true measurement error is 1.0 (arbitrary units).

The figure presents a "tree diagram" that highlights the flow of information through the model. Note that the parameters $\mu_i$ define a distribution (with $\sigma$), but that $\mu_i$ 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 $\lambda$, which is precisely what we will do later.

We can express this model as:

$$\begin{aligned} s_{ij} &\sim N(\mu_i, \sigma = 1) \\ \mu_i &\sim exp(\lambda = 10) \end{aligned} \tag{1}$$

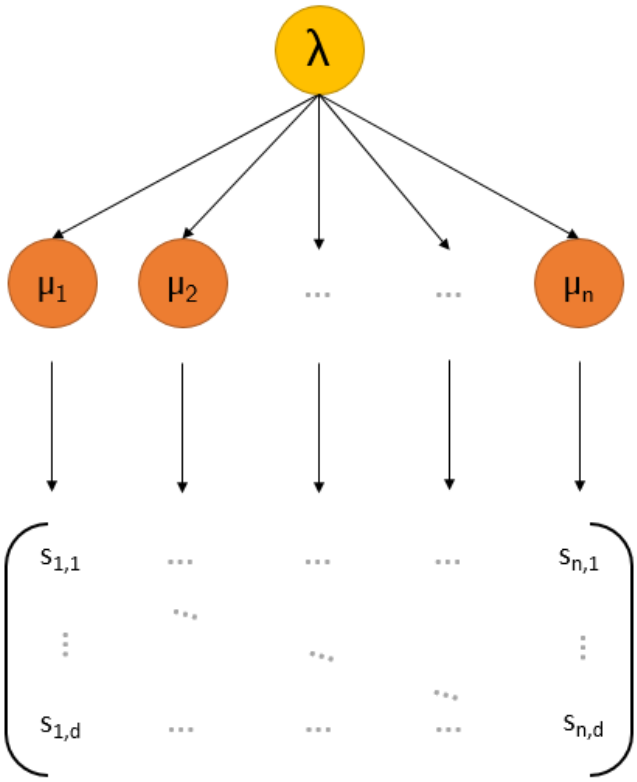


Figure 1: Rough sketch of the hierarchical model

### 1.2 hierarchical_model_1-1.stan

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

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

```
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 $\lambda$ 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:

$$\begin{aligned} s_{ij} &\sim N(\mu_i, \sigma) \\ \mu_i &\sim exp\Big(\frac{1}{\beta}\Big) \\ \sigma &\sim N(0, 100) \\ \beta &\sim N(0, 100) \end{aligned} \tag{2}$$

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.

### 1.3 hierarchical_model_1-1.py

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

```
def main():
```

```
  dataset = simulate()
  inference(dataset)
  return
```

Inference takes in a *dataset* and calls Stan.

**Exercise 1**

**Writing Simulation Code**

Complete the simulate() function. We suggest the variable values: N = 15, D = 30, *exponential_lambda* = 10 (scale = 0.1), *normal_sigma* = 1.

This link may be useful: https://docs.scipy.org/doc/numpy/reference/routines.random.html

Keep in mind that variable names and types must match exactly between Stan's data code block and the dictionary returned by simulate().

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

## 2 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 sample size of inference for 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 (set to 2000 in this model), 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.

### 2.1 Plotting the Fit

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

### 2.2 Pickling Fitted Models

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

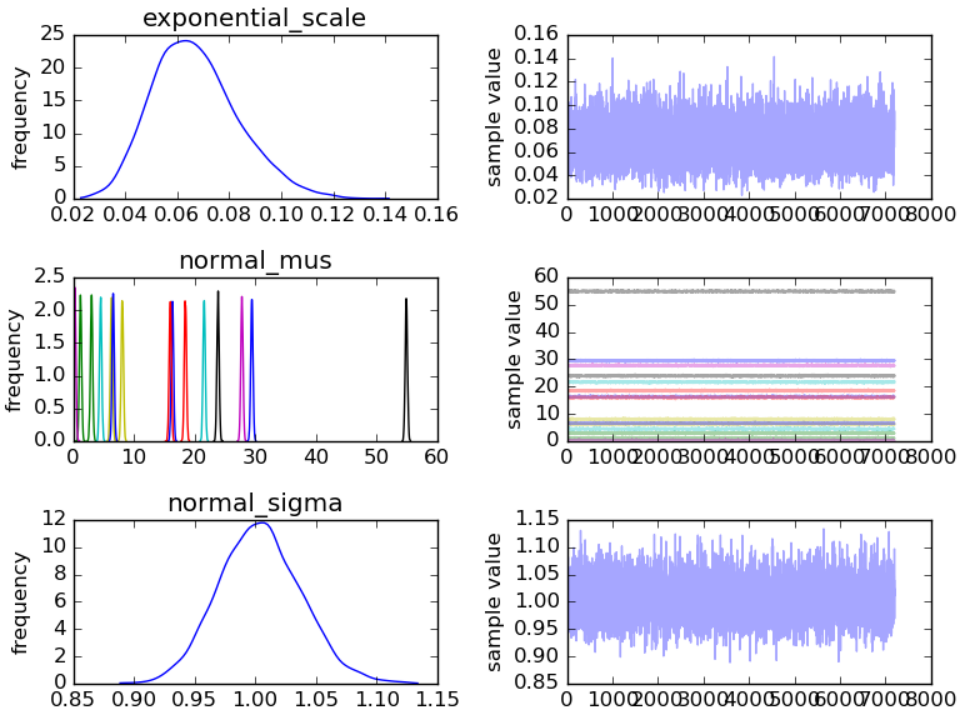


Figure 2: fit.png

The "fit" object can be saved to file by pickling "fit.extract()".

### 2.3 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**

**Exploration**

1. You can extract the final estimates for *normal_mu* from the "fit" object as:

```
normal_mus = fit['normal_mus'][-1]
```

This grabs the last iteration's mean for the parameter's posterior distribution. Compare your posterior values to the true values you simulated. How close are they?

2. 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?

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

4. 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?

## 3 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.