

# Stan Part 2

## More Complex Models

# Outline

- Stan Review
- Complex Models in Stan

# Stan Review

## Data Block –

- Define each piece of data that goes into your model.
- Not only do you input data here, you should also input hyperparameters and auxiliary values.
  - Like sample size, or any other counts.
- Must provide exact type and constraints on the values.

```
data {  
    int<lower=0> N;  
    vector[N] y;  
}  
parameters {  
    real mu;  
    real<lower=0> sigma;  
}  
model {  
    y ~ normal(mu, sigma);  
}
```

# Stan Review

## Parameters Block –

- Define the parameters in your model.
- Provide the type and constraints
- No transformed/functions of parameters, that goes into the transformed parameters block.

```
data {  
    int<lower=0> N;  
    vector[N] y;  
}  
parameters {  
    real mu;  
    real<lower=0> sigma;  
}  
model {  
    y ~ normal(mu, sigma);  
}
```

# Stan Review

## Model Block –

- This is where you specify the prior distributions and the likelihoods.
- You can use any data, parameters or transformed parameters in these calculations.
- Stan processes code in order. So, you need to specify priors, then specify your likelihoods.

```
data {  
    int<lower=0> N;  
    vector[N] y;  
}  
parameters {  
    real mu;  
    real<lower=0> sigma;  
}  
model {  
    y ~ normal(mu, sigma);  
}
```

we are saying our data is distributed normally with these parameters

# Linear Regression in Stan

classic linear regression model (think of  $i$  as "person")  $y_i = x_i \beta + \varepsilon_i$  what is the distr. of  $x$  sub  $i$ ? - the answer is it doesn't matter. The distr. of our predictors does not matter. The distr. of our outcome given our param DOES MATTER, but when we are looking at just  $x$  sub  $i$ , it's just observed data  
 $\varepsilon_i \sim N(0, \sigma^2)$

Consider the humble linear regression. First, let's determine what counts as data here:

- $y$  – vector of length  $N$ , taking real values with no constraints
- $\mathbf{X}$  – Matrix of dimensions  $N \times p$ , real values with no constraints.

Auxiliary information: these are all the elements that go into your data block.

- $N$  – number of observations. Positive integer.
- $p$  – number of predictor variables. Positive integer.

# Data Block for Linear Regression

- $y$  – vector of length  $N$ , taking real values with no constraints
- $\mathbf{X}$  – Matrix of dimensions  $N \times p$ , real values with no constraints.

Auxiliary information:

- $N$  – number of observations.  
Positive integer.
- $p$  – number of predictor variables.  
Positive integer.

this is all the data you need for our linear regression

```
data {  
    int<lower=0> N;  
    int<lower=0> p;  
    vector[N] y;  
    matrix[N, p] X;  
}
```

# Linear Regression in Stan

this is different from  $Y = \text{Beta\_sub\_0} + X * \text{Beta} + \text{Epsilon}$

$$y_i = x_i \beta + \varepsilon_i$$
$$\varepsilon_i \sim N(0, \sigma^2)$$

Now, what about the parameters?

- $\beta$  – vector of length  $p$ , real values, no constraints
- $\sigma^2$  - positive real value.

```
data {  
    int<lower=0> N;  
    int<lower=0> p;  
    vector[N] y;  
    matrix[N, p] X;  
}  
parameters {  
    vector[p] betas;  
    real<lower=0> sigma;  
}
```

we have to specify our beta and our sigma.



# Priors for Linear Regression in Stan

Multivariate prior on  $\beta$

- Needs to cover the real line
- Let's use independent normals.
- Note: Stan is vectorized, so that prior spec applies the same prior to all  $\beta$ s

Positive Real Valued prior on  $\sigma^2$

pronounced "Co-Shi"

- Let's use a half Cauchy here
- Because  $\sigma$  is constrained at 0, this leads to a half Cauchy

Make sure you aren't hard coding these mu and sigma values when you actually implement them. You initialize these values in the data{} parameter

```
model {  
    betas ~ normal(0,10);  
    sigma ~ cauchy(0,5);  
}
```

centered at = 0, variance of = 10

centered at = 0, variance of = 5

# Model for Linear Regression in Stan

$$y_i = \mathbf{x}_i \boldsymbol{\beta} + \varepsilon_i \quad \text{our mean is } \mathbf{X} \cdot \boldsymbol{\beta}, \text{ and our variance is still } \sigma^2$$
$$\varepsilon_i \sim N(0, \sigma^2)$$

What is the likelihood here:

- $y_i$  is distributed as a normal variable with mean  $\mathbf{x}_i \boldsymbol{\beta}$
- Why? Think about rules of expectations.

```
model {  
  betas ~ normal(0,10);  
  sigma ~ cauchy(0,5);  
  y ~ normal(X * betas, sigma);  
}
```

# All Together

```
data {  
    int<lower=0> N;  
    int<lower=0> p;  
    vector[N] y;  
    matrix[N, p] X;  
}  
parameters {  
    vector[p] betas;  
    real<lower=0> sigma;  
}  
model {  
    betas ~ normal(0,10);  
    sigma ~ cauchy(0,5);  
    y ~ normal(X * betas, sigma);  
}
```

# All Together

Generated data with the following:

- 1 intercept of value 2, 2 predictors with betas of .5 and 1.5.
- Sigma is .5
- 200 observations.

	mean	se_mean	sd
betas[1]	2.006755	0.000572	0.036063
betas[2]	0.47539	0.00057	0.036495
betas[3]	1.519859	0.000524	0.03538
sigma	0.500919	0.000406	0.025694
lp__	38.25616	0.030454	1.404715

# Generalized Linear Model

$y$  was a continuous variable, which meant that linear regression is appropriate.

- How about if  $y$  is dichotomous?
- Or a count?

Stan makes changing the response distribution very easy.

```
data {  
    int<lower=0> N;  
    int<lower=0> p;  
    array[N] int<lower=0, upper=1> y;  
    matrix[N, p] X;  
}  
parameters {  
    vector[p] betas;  
}  
model {  
    betas ~ normal(0,10);  
    //For a logistic regression  
    y ~ bernoulli_logit(X * betas);  
}
```

we are not putting sigma here b/c the error will roll into this distr.  
we therefore don't need to specify this

^this is our logistic regression. It's pretty simple to do and just use the  
convenience function above

# Generalized Linear Model

$y$  was a continuous variable, which meant that linear regression is appropriate.

- How about if  $y$  is dichotomous?
- Or a count? Poisson Distr. to model counts (THINK FISH!)

Stan makes changing the response distribution very easy.

```
data { same data & same parameters.
  int<lower=0> N;
  int<lower=0> p;
  array[N] int<lower=0> y;
  matrix[N, p] X;
}
parameters {
  vector[p] betas;
}
model {
  betas ~ normal(0,10);
  //For a Poisson regression
  //0 is for intercept, as that is in
  //the betas
  y ~ poisson_log_glm(X, 0, betas);
}
```

you can't have a neg. count, so the appropriate function is the poisson log glm. Instead of just multiplying data and treating it as the poisson mean it's actually  $X \cdot \text{Beta}$ .

# Generalized Linear Model

Generated 1000 observations:

- Betas: 0, .25, -.25
- Outcome, logistic y.

pos int., pos 1st predictor (beta), neg last val

	mean	se_mean	sd
betas[1]	-0.10199	0.00097	0.063958
betas[2]	0.250536	0.001123	0.066339
betas[3]	-0.20258	0.001138	0.066409
lp__	-682.011	0.026912	1.248414

Generated 1000 observations:

- Betas: 2, 3, -2
- Outcome: Poisson y.
- Note: Coefficients are in log form.

	mean	se_mean	sd
betas[1]	0.52455	0.000543	0.027194
betas[2]	0.803861	0.000385	0.019549
betas[3]	-0.51686	0.000396	0.01998
lp__	1090.908	0.030559	1.250715

# Training/Testing in Stan

Cross-validation is an important component of any DS stack.

- Train our model on some set of data.
- Test the performance of our model on a new set of data.

In Bayes, we can compute the posterior predictive distribution:

- Given our posteriors, what are our predictions for a new set of data.

In Stan:

- We can input our new data as part of our data block.
- Then use the generated quantities block to sample our predictions.



# Training/Testing in Stan

In Stan:

- We can input our new data as part of our data block.
- Then use the generated quantities block to sample our predictions.

Generated quantities:

- No impact on model fit.
- Can be extracted with `extract` in R

```
data {  
  Nt num of testing samples. N is num of training data  
  int<lower=0> N;  
  int<lower=0> Nt  
  int<lower=0> p;  
  vector[N] y;  
  matrix[N, p] X;  
  this should be a vector  
  matrix[N_t, p] X_tilde;  
  matrix[N_t, p] Y_tilde;  
}  
//Parameters and model block are the same  
generated quantities {  
  vector[N_t] tilde_y = x_tilde * betas;  
  vector[N_t] error = Y_tilde - tilde_y;  
}  
regular training and testing set up
```

# Training/Testing in Stan

Linear regression:

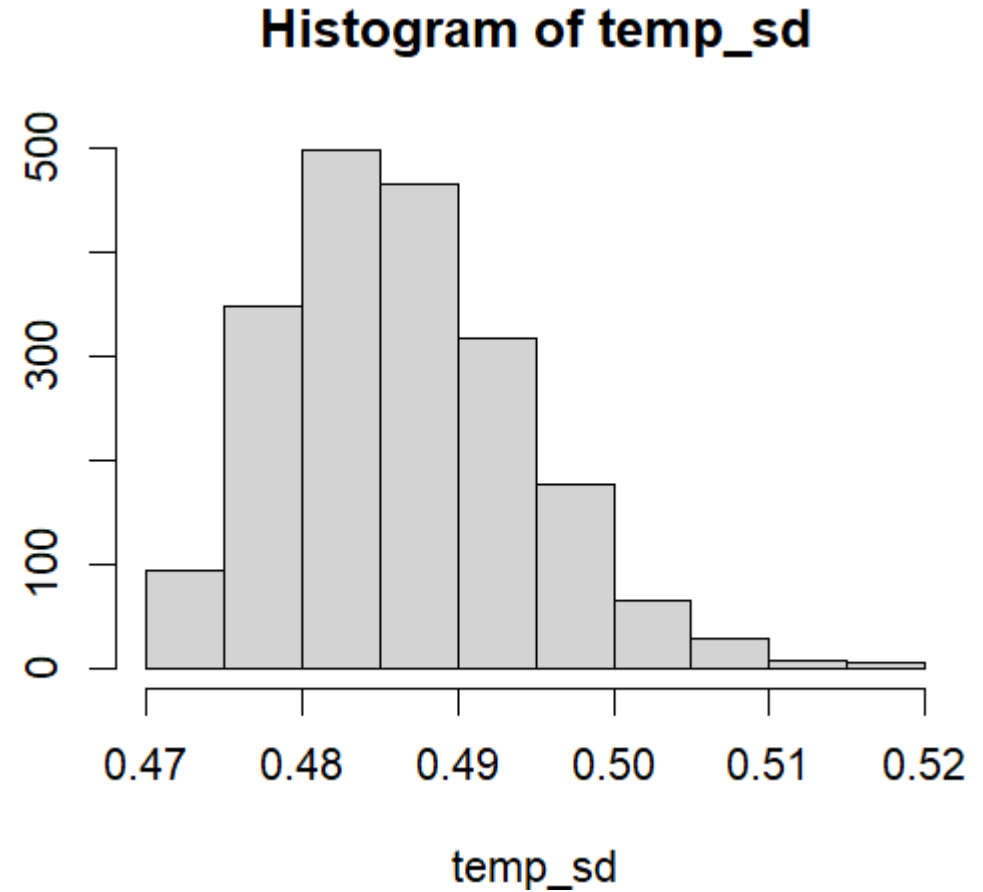
- 50 testing samples

Plot the distribution of estimated testing error standard deviations

- This should be approximately .5

Slight underestimate of testing error residual...

- Result of the prior choice with small sample (100 training)



# K-Fold Validation in Stan

You can program a random split procedure into any Stan model.

First, we need to define a user function that can generate permutations of the data.

functions block-

- Define user functions in C++ type notation.

```
functions {  
  array[] int permutation_rng(int N) {  
    array[N] int y;  
    for (n in 1 : N) {  
      y[n] = n;  
    }  
    vector[N] theta = rep_vector(1.0 / N, N);  
    for (n in 1 : size(y)) {  
      int i = categorical_rng(theta);  
      int temp = y[n];  
      y[n] = y[i];  
      y[i] = temp;  
    }  
    return y; y is our new set of permuted indices  
  }  
}
```

# K-Fold Validation in Stan

Next, we need to use that function:

- In the transformed data block, we use the permutation function to define testing and training sets.

```
data {  
  int<lower=0> N;  
  int<lower=0> p;  
  matrix[N,p] x;  
  vector[N] y;  
  int<lower=0, upper=N> N_test;  
}  
transformed data {  
  int N_train = N - N_test;  
  array[N] int permutation = permutation_rng(N);  
  matrix[N_train,p] x_train = x[permutation[1 : N_train],];  
  vector[N_train] y_train = y[permutation[1 : N_train]];  
  matrix[N_test,p] x_test = x[permutation[N_train + 1 : N],];  
  vector[N_test] y_test = y[permutation[N_train + 1 : N]];  
} the above is equivalent to selecting at random in training and testing data
```

# K-Fold Validation in Stan

Next, we need to use that function:

- In the transformed data block, we use the permutation function to define testing and training sets.

This setup will compile into a model that randomly selects  $N_{\text{test}}$  testing cases each time its run.

```
model {  
  // Same priors as before  
  betas ~ normal(0,10);  
  sigma ~ cauchy(0,5);  
  y_train ~ normal(x_train*betas, sigma);  
}  
generated quantities {  
  the below is simulated: "generate me a normal var for every obs for a mean of ... and our sigma of ..."  
  vector[N_test] y_test_sim =  
    as_vector(normal_rng(x_test*betas,  
    sigma));  
  vector[N_test] err = y_test_sim -  
    x_test*betas;  
}
```

every time you run the model, you will get a new fold. These folds are not mutually exclusive. you could reset the seed at each fold, which helps you say there's specific folds at each time. You could also prepare the testing and training model and run the model 10 times on your fold.

# Summary and Important Concepts

Stan is a strongly typed language: Be careful as to how you define different variables.

New Block Types:

- functions – User defined functions.
- transformed data – Manipulations of provided data.
- transformed parameters – Manipulations of parameters.
- generated quantities – Any output you want after the model is done.
  - Usually for predictive distributions...

# Next Time

No class 10/8 and 10/10 (I'm away at a conference)

- Review course materials!
- Play around with Stan!

No class 10/15 – University Reading Days!

Next Class: 10/17

- Midterm Review!

Midterm: 10/22