Bayesian Linear Models

Sarah Hunter

5/19/2020

Preparing the Data

JAGS requires a bit more data management than is necessary for the classical linear regression model. The most important difference is that JAGS requires data be in a list format in contrast to the data frame format required for canned models like `glm` or `lm`. The following example uses the Prestige data from the `car` package. The first step is to load the data into R as usual. Our practice data is from R already, so we can skip this step. Once we have the data into R, we need to take each variable out of the data and make it into its own vector. Once that is done, we can put all of the data into a list.

```
#The necessary packages
library(car)
```

```
## Loading required package: carData
```

```
library(R2jags)
```

```
## Loading required package: rjags

## Loading required package: coda

## Linked to JAGS 4.3.0

## Loaded modules: basemod,bugs

##
## Attaching package: 'R2jags'

## The following object is masked from 'package:coda':
##
##     traceplot
```

```
library(rjags)
library(ggplot2)

#making vectors out of each variable
income<-Prestige$income
education<-Prestige$education
prestige<-Prestige$prestige
N<-length(income)

#turning those vectors into a list
jagsdata<-list("income", "education", "prestige", "N")
jagsdata<-list(income=income, education=education, prestige=prestige, N=N)
```

Writing the Model

The next step in your model is to write the model. This includes specifying your priors with both the mean and variance. Every stochastic element of the model needs a prior. The one important thing to remember in JAGS is that the second parameter of a distribution is *NOT* the variance. It is actually the *precision*, which is the inverse of the variance. So, remember to put the precision in the JAGS code, not the variance.

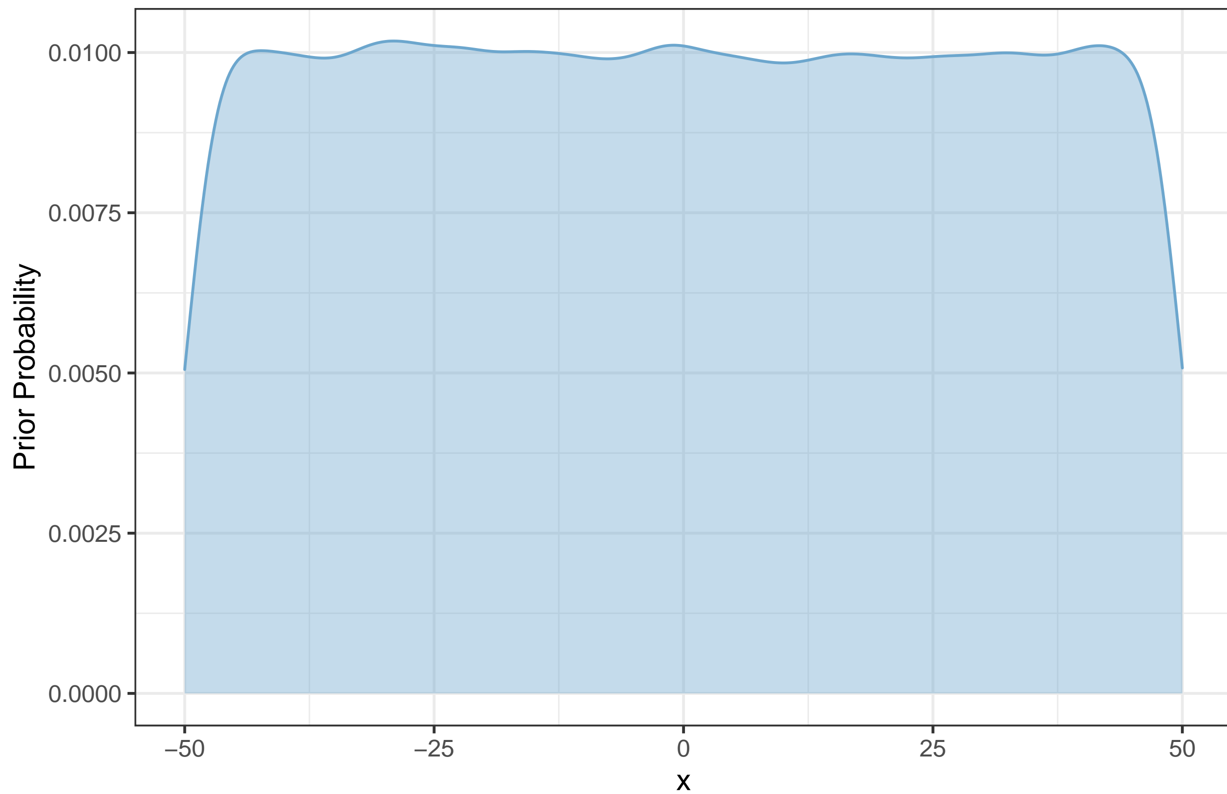
Choosing a Prior

You can choose your prior to reflect what you need from the prior. You can choose a prior that is as informative as you want. Each prior is specified in the model as a distribution of your choice with parameters that dependent on the distribution chosen. Section 9.2 of the JAGS User Manual discusses probability distributions in more detail and describes how each is parameterized. Here, I will simply go over the most common distributions and their parameters:

Probability Distribution	Call in JAGS	Parameters
Normal Distribution	dnorm	mean, precision
Uniform Distribution	dunif	minimum, maximum
Gamma Distribution	dgamma	shape, rate

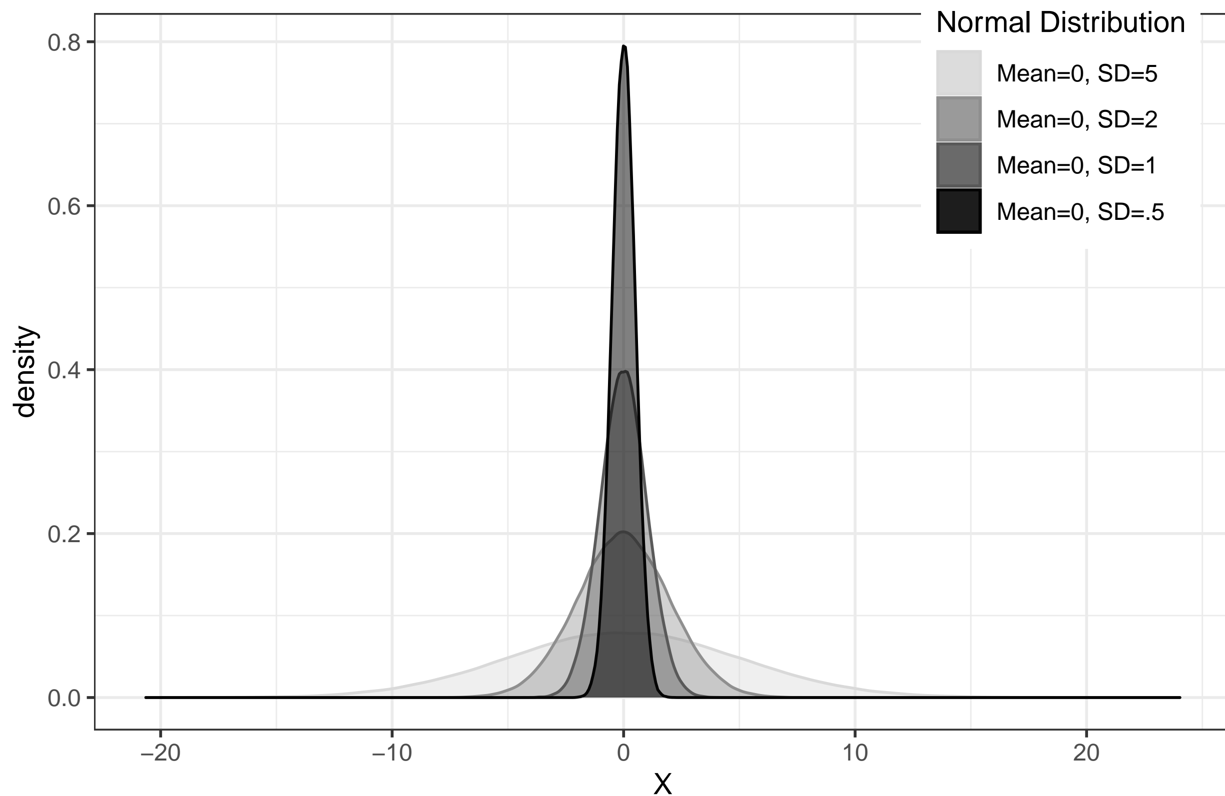
To understand what these distributions do, you can plot them in R using the base plot function. You just need to do many random draws from that distribution using the `r` prefix and plot the density function of those random draws. The `r` prefix outside of JAGS uses different, but related parameters. You can find how R parameterizes these functions using the `?` symbol before the command to get the help file. Below, I have plotted versions of distributions with different parameterizations. First, I plot the uniform distribution:

```
#Take 100000 random draws from a unifrm distribution from -1- t0 10
d<-runif(100000, min=-50, max=50)

#Make it into a dataframe for ggPlot
d<-as.data.frame(d)


ggplot(d, aes(x=d))+ geom_density(fill="skyblue3", color="skyblue3", alpha=.4)+theme_bw() +labs(title="
```



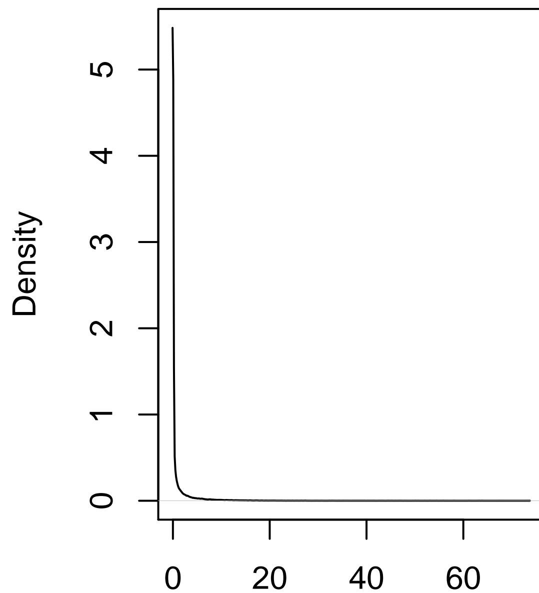
Below is the Normal distribution with the same mean and four different standard deviations.

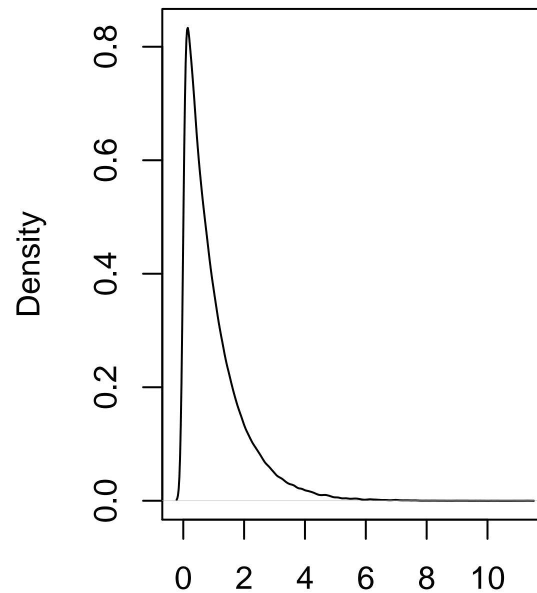
```
norm1<-rnorm(100000, mean=0, sd=5)
norm2<-rnorm(100000, mean=0, sd=2)
norm3<-rnorm(100000, mean=0, sd=1)
norm4<-rnorm(100000, mean=0, sd=.5)

norms<-as.data.frame(cbind(norm1, norm2, norm3, norm4))

ggplot(data=norms)+geom_density(aes(x=norm1, color="norm1", fill="norm1"), alpha=.4)+geom_density(aes(x=
```



The Gamma distribution is useful as a prior for the variance in a Bayes model because it is bounded by 0. It cannot be less than 0, just like a variance must be positive. The Gamma distribution can be parameterized in many different ways. Be sure to check which parameters R uses (sometimes it uses rate, other times, it uses scale). JAGS uses shape and rate. I show an example below:

```
#100000 random draws from a Gamma distribution

gam1<-rgamma(100000, shape=.1, rate=.1)
gam2<-rgamma(100000, shape=1, rate=1)

#Using base plot for density

par(mfrow=c(1,2))

plot(density(gam1), main="Gamma Distribution (0.1, 0.1)")
plot(density(gam2), main="Gamma Distribution (1.0, 1.0)")
```





Full Model Specification

For our example, the full model specification is:

$$Y_i = \alpha + \beta_1 * income_i + \beta_2 * education_i$$

where

$$y \sim \mathcal{N}(\mu, \tau)$$

$$\alpha \sim \mathcal{N}(0, 1)$$

$$\beta_1 \sim \mathcal{N}(0, 1)$$

$$\beta_2 \sim \mathcal{N}(0, 1)$$

$$\tau \sim Gamma(.1, .1)$$

For this model, I have used a standard normal distribution as the prior on all the estimated parameters, except the precision (which uses a Gamma distribution). We translate that model specification to model two in one of two ways: either a model plain text file or a function in R. For smaller models, the function methods tend to be easier. For larger, more complex models, the text file can be easier with which to work. Just remember to save the model file in your R working directory. In this example, I am going the function route. The following code translates the above model into a JAGS model:

```
prestige.model.jags<-function(){     #Name the model, tell R it is a function

  for(i in 1:N){ #Start the loop over the observations
    prestige[i]~dnorm(mu[i], tau) #prior on prestige
    mu[i]<-alpha + beta1*education[i] + beta2*income[i]
    #models the mean of the prior distribution of prestige
  }
```

```
  alpha~dnorm(0,.01) #prior on the intercept
  beta1~dnorm(0, .1) #prior on beta1
  beta2~dnorm(0, .1) #prior on beta2
  tau~dgamma(.1,.01) #prior on the precision

}
```

Implementing the Model

When implementing the model, you must make decisions about and include the following features:

- Number of chains
- Number of Iterations
- Initial values
- The parameters to monitor
- The Burn-in period

Chains

`n.chains` :For the number of chains, we nearly always run at least two chains in your model. This has a lot to do with convergence of the model, which we will discuss tomorrow.

Iterations

`n.iter`: This tells you how many times to estimate the model. The more complicated the model, the more iterations that are needed in order for the model for converge.

Initial Values

`inits`: Initial values tells the sampling algorithm where to start. You do not have to provide initial values for your model, but it can help speed up the model if you do. This model allows JAGS to randomly pick the initial values using the `inits=NULL` option.

Parameters to Monitor

`parameters.to.save=`: You do not have to save the estimates for every estimated parameters. However, you do need to tell JAGS which parameters you will need to save, and therefore provide output. We first need to create a vector of parameters to watch using the `c()` command. I called mine `params`. Like any function in R, as long as you have the order nearly correct, you do not have to type the option and assign it value. As you can see in my model code, I simply use the option of the name (`params`) of the parameters I wanted to save (instead of `parameters.to.save=params`). This is simply laziness on my part. It does not matter which way you do it.

Burn-in Periods

`n.burnin`: You do need to give the sampler a warm-up period. The first few iterations of the sampling algorithm as essentially random guesses to seek the area of highest density. Since the first few iterations are junk, we usually throw them out. You can tell JAGS when to start saving results. For this model, I used 400 iterations as a burn-in period. I usually use a burn-in period of about 10% of the total iterations, but there really is not strict rule guiding the choice of burn-in period. As long as the model converges, it only matters a little.

```r
params<-c("alpha", "beta1", "beta2") # Tell JAGS which parameters to watch

#fit the model
fit<-jags(data=jagsdata, inits=NULL, params, n.chains=2, n.iter=4000,
          n.burnin=400, model.file=prestige.model.jags)
```

```
## module glm loaded
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 102
##    Unobserved stochastic nodes: 4
##    Total graph size: 611
##
## Initializing model
```

```r
print(fit)
```

```
## Inference for Bugs model at "/var/folders/dx/pkx9cyj1089843ljm_jzj3pm0000gn/T//RtmpjhUi9E/model726b6
##  2 chains, each with 4000 iterations (first 400 discarded), n.thin = 3
##  n.sims = 2400 iterations saved
##          mu.vect sd.vect    2.5%     25%     50%     75%   97.5%  Rhat n.eff
## alpha     -5.931   3.053 -11.753  -8.003  -5.917  -3.940   0.205 1.001  2400
## beta1      4.042   0.333   3.382   3.828   4.044   4.267   4.675 1.002  1500
## beta2      0.001   0.000   0.001   0.001   0.001   0.002   0.002 1.001  2400
## deviance 709.796   2.832 706.285 707.672 709.157 711.155 716.851 1.001  2400
##
## For each parameter, n.eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor (at convergence, Rhat=1).
##
## DIC info (using the rule, pD = var(deviance)/2)
## pD = 4.0 and DIC = 713.8
## DIC is an estimate of expected predictive error (lower deviance is better).
```

```r
# Update if necessary, depending on convergence
fit.upd<-update(fit, n.iter=1000)
fit.upd<-autojags(fit)

print(fit)
```

```
## Inference for Bugs model at "/var/folders/dx/pkx9cyj1089843ljm_jzj3pm0000gn/T//RtmpjhUi9E/model726b6
##  2 chains, each with 4000 iterations (first 400 discarded), n.thin = 3
##  n.sims = 2400 iterations saved
##          mu.vect sd.vect    2.5%     25%     50%     75%   97.5%  Rhat n.eff
## alpha     -5.931   3.053 -11.753  -8.003  -5.917  -3.940   0.205 1.001  2400
## beta1      4.042   0.333   3.382   3.828   4.044   4.267   4.675 1.002  1500
## beta2      0.001   0.000   0.001   0.001   0.001   0.002   0.002 1.001  2400
## deviance 709.796   2.832 706.285 707.672 709.157 711.155 716.851 1.001  2400
##
## For each parameter, n.eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor (at convergence, Rhat=1).
##
## DIC info (using the rule, pD = var(deviance)/2)
## pD = 4.0 and DIC = 713.8
```

```
## DIC is an estimate of expected predictive error (lower deviance is better).
```

Model Presentation

Presenting a Bayesian model is a lot like presenting a logit or probit model: tables are fine, but figures are best. The following section discusses ways to present results for both.

Tables

Bayesian models do not p lay nicely with the R tools we use to create nice tables (i.e. Stargazer, xtable). However, we can still produce a table with some quick adjustments. The first step is to decide which columns of the summary that you need.

```
#The columns you need
fit$BUGSoutput$summary[, c(1,2,3,7)]
```

```
##                   mean          sd          2.5%        97.5%
## alpha     -5.930808283 3.052963609 -1.175314e+01 2.049925e-01
## beta1      4.041824963 0.333109376  3.381925e+00 4.674825e+00
## beta2      0.001380093 0.000225326  9.430733e-04 1.809296e-03
## deviance 709.796365767 2.832372939  7.062854e+02 7.168514e+02
```

Once you know that, you can use matrix notation to select what you need and use that matrix to make an xtable object.

```
library(xtable)

model.table<-xtable(fit$BUGSoutput$summary[, c(1, 2,3,7)], digits=3)

print(model.table, type="latex")
```

```
## % latex table generated in R 3.6.3 by xtable 1.8-4 package
## % Tue May 26 11:00:30 2020
## \begin{table}[ht]
## \centering
## \begin{tabular}{rrrrr}
##   \hline
##  & mean & sd & 2.5\% & 97.5\% \\
##   \hline
## alpha & -5.931 & 3.053 & -11.753 & 0.205 \\
##   beta1 & 4.042 & 0.333 & 3.382 & 4.675 \\
##   beta2 & 0.001 & 0.000 & 0.001 & 0.002 \\
##   deviance & 709.796 & 2.832 & 706.285 & 716.851 \\
##    \hline
## \end{tabular}
## \end{table}
```

After you make the table, you will still need to make adjustments to names of the variables. Notice that the output tells us the names of the parameters we monitored. You can do this easily in Latex or Word.

Figures

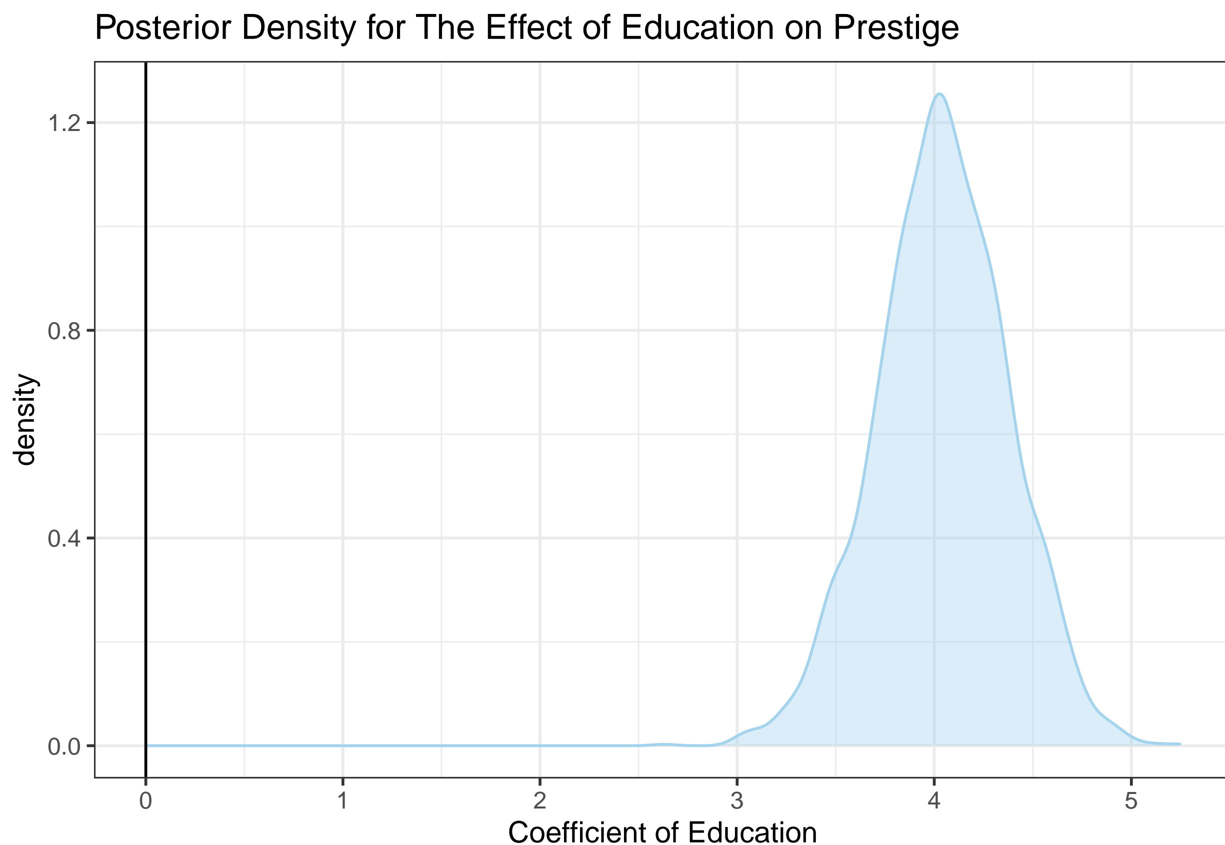
One of the best ways to display regression results with Bayes is to use plots. However, you need to convert our results from a JAGS object to an mcmc object. From an mcmc object, you can then convert the model

results into a matrix, then a data frame. Each type of object is useful to some postestimation, but not others. You do this by:

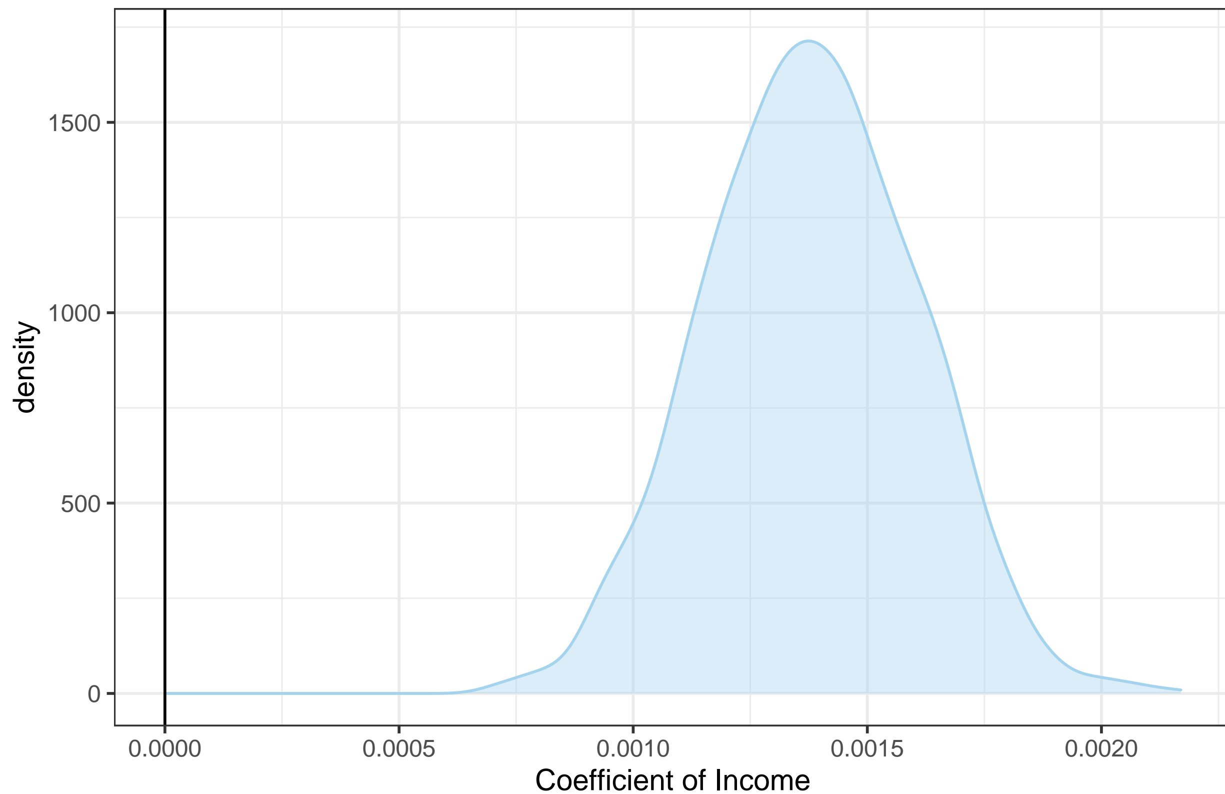
```
#Turn the JAGS object into MCMC object to manipulate
fit.mcmc<-as.mcmc(fit)

#Also, turn into a matric or dataframe for ease
fit.mat<-as.matrix(fit.mcmc)
fit.df<-as.data.frame(fit.mat)
```

Once your results are in the proper format, you can start plotting. You can create any of your favorite plots from your model once the results are in the data frame format. Here is an example of using `ggPlot2` to plot the posterior density that shows the density in relation to 0. `ggPlot` requires the data be in a data frame format.

```
##ggPlot of the posteror densities
library(ggplot2)

ggplot(fit.df, aes(x=beta1))+geom_density(color="lightskyblue2",
              fill="lightskyblue2", alpha=.4)+geom_vline(xintercept = 0)+theme_bw()+
              labs(title="Posterior Density for The Effect of Education on Prestige",
              x="Coefficient of Education")
```




```
ggplot(fit.df, aes(x=beta2))+geom_density(color="lightskyblue2",
              fill="lightskyblue2", alpha=.4)+geom_vline(xintercept = 0)+theme_bw()+
              labs(title="Posterior Density for The Effect of Income on Prestige",
              x="Coefficient of Income")
```

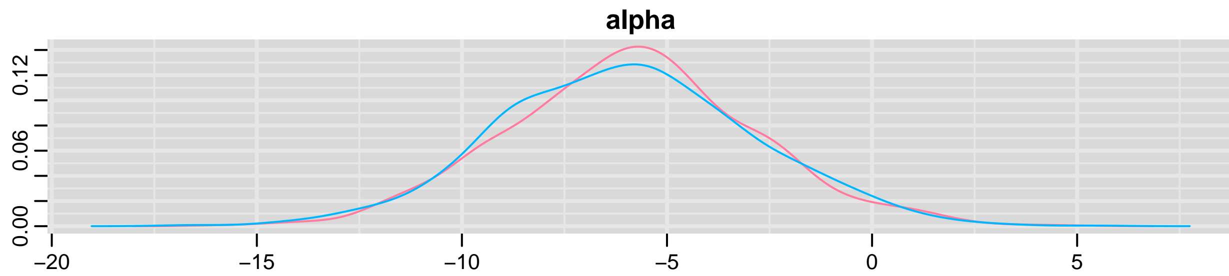


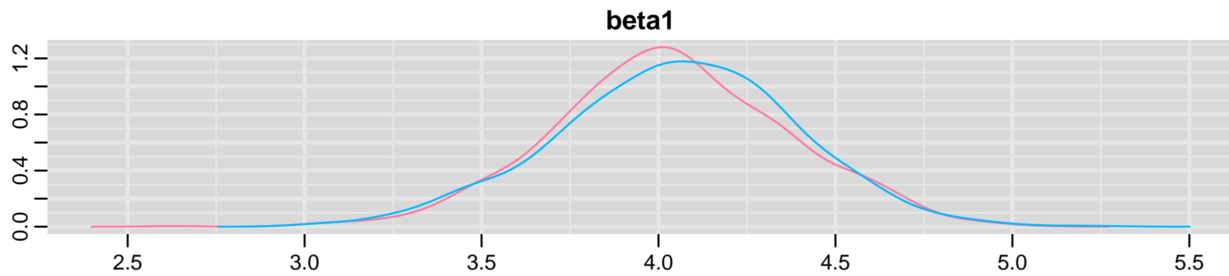
There are many of other plots available for Bayesian models, and they all look fairly cool and have some convenient canned commands. There are many plots you can do, these are just a few. The first package is `mcmcplots` and the second is `ggmcmc`. These two packages both start with mcmc objects. You do need to convert the mcmc object for the `ggmcmc` package.

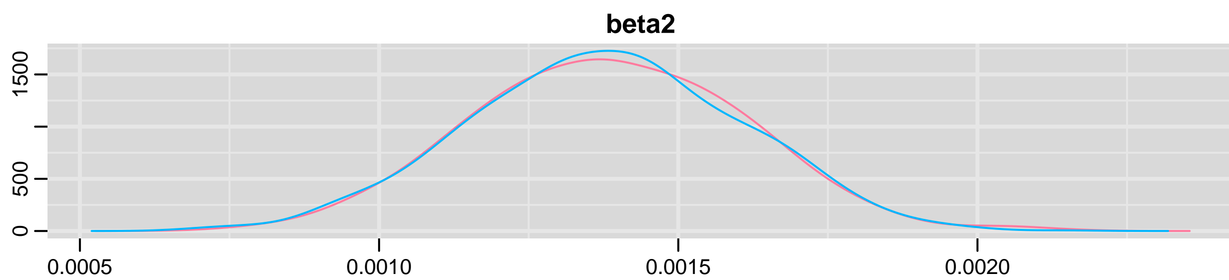
```
#cool Bayes plots
#install.packages("mcmcplots")
library(mcmcplots)
```

```
## Registered S3 method overwritten by 'mcmcplots':
##   method        from
##   as.mcmc.rjags R2jags
```

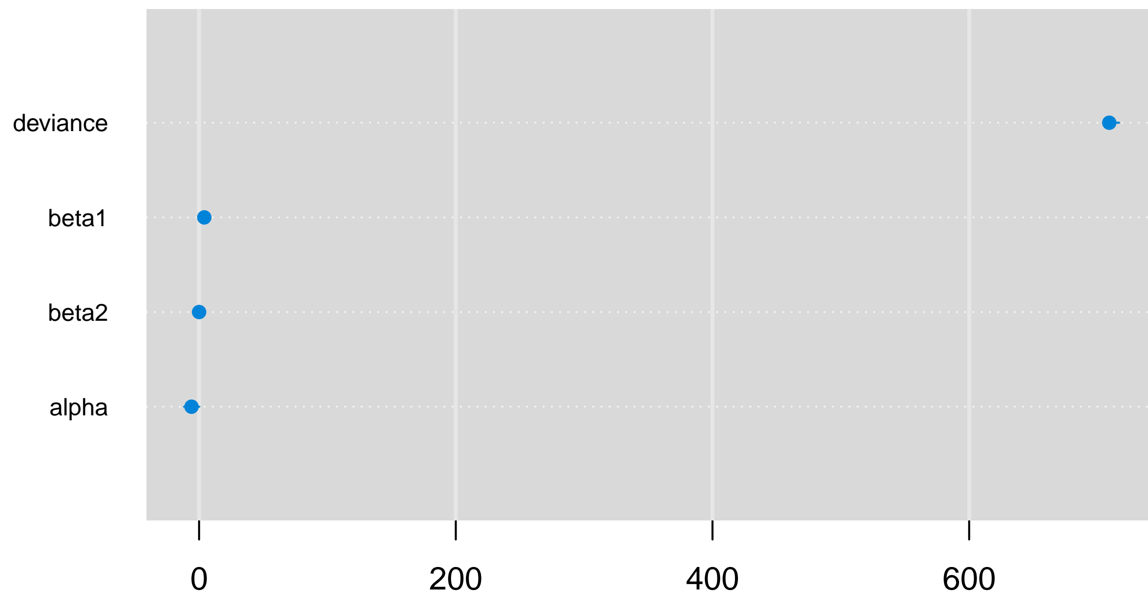
```
#Density Plots
denplot(fit.mcmc, parms = c("alpha", "beta1", "beta2"))
```



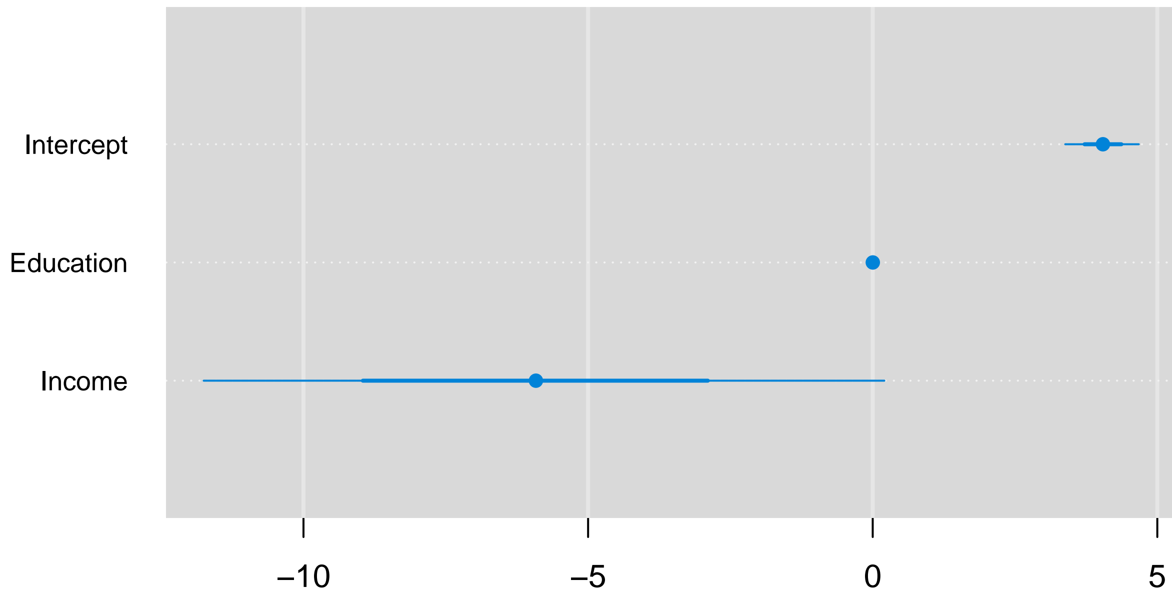





```
#Dot Plots
caterplot(fit.mcmc)
```



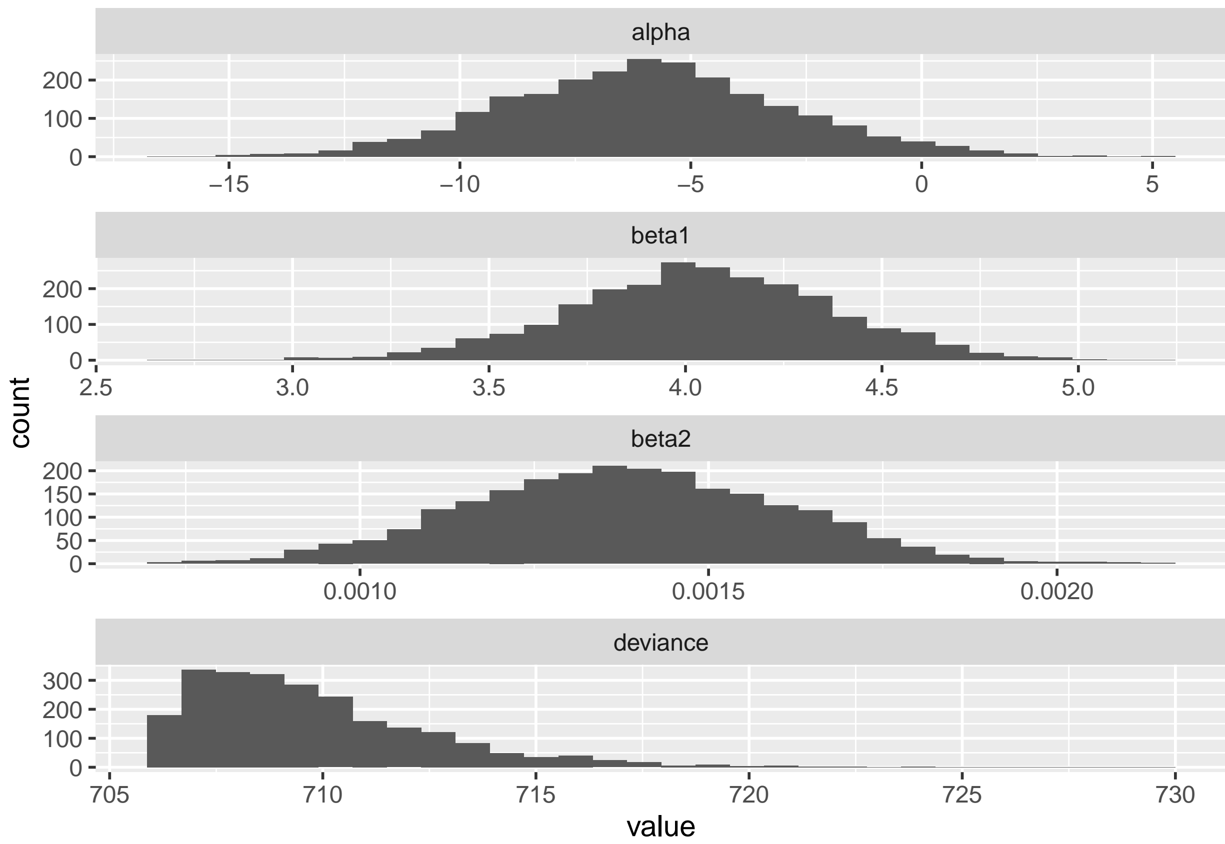

```
caterplot(fit.mcmc, parms = c("alpha", "beta1", "beta2"),
          labels = c("Intercept", "Education", "Income"))
```



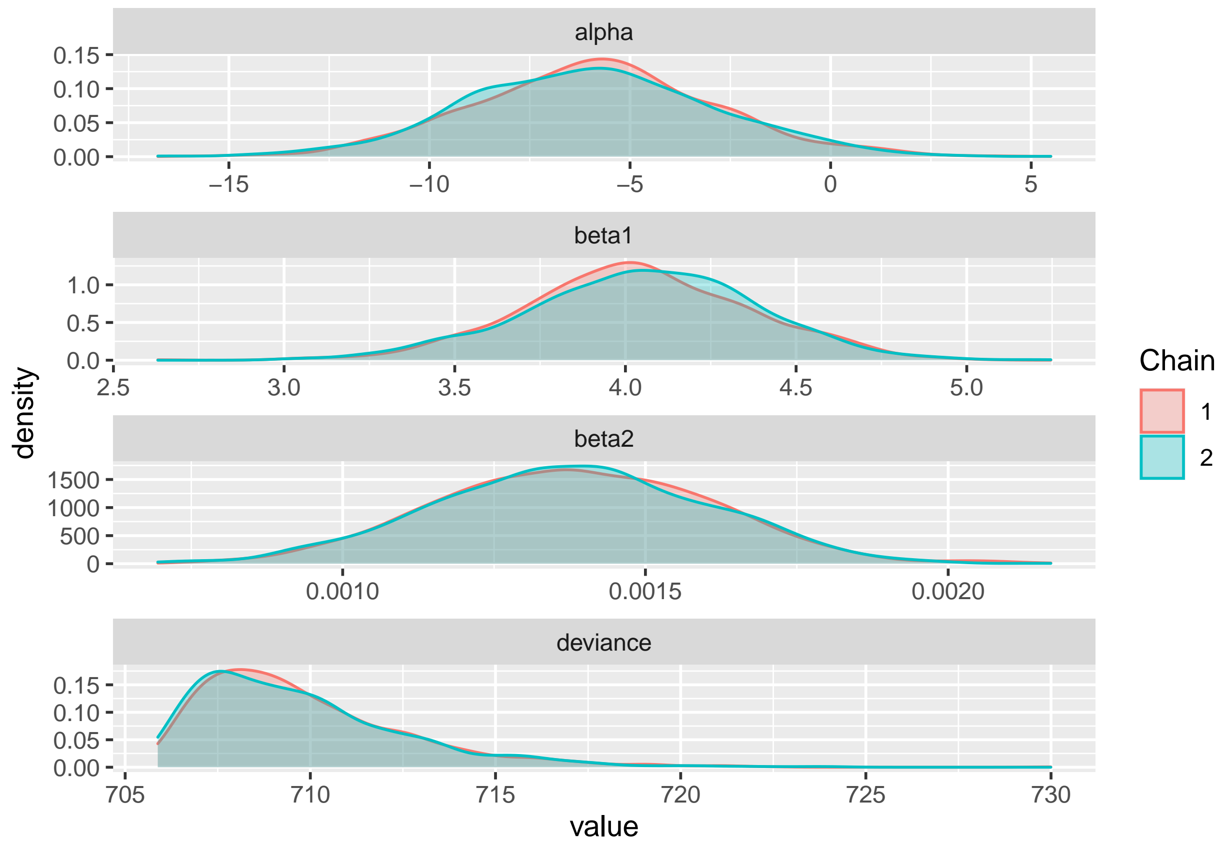

```r
#install.packages("ggmcmc")
library(ggmcmc)
```

```
## Loading required package: dplyr

##
## Attaching package: 'dplyr'

## The following object is masked from 'package:car':
##
##     recode

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union

## Loading required package: tidyr

## Registered S3 method overwritten by 'GGally':
##   method from
##   +.gg   ggplot2
```

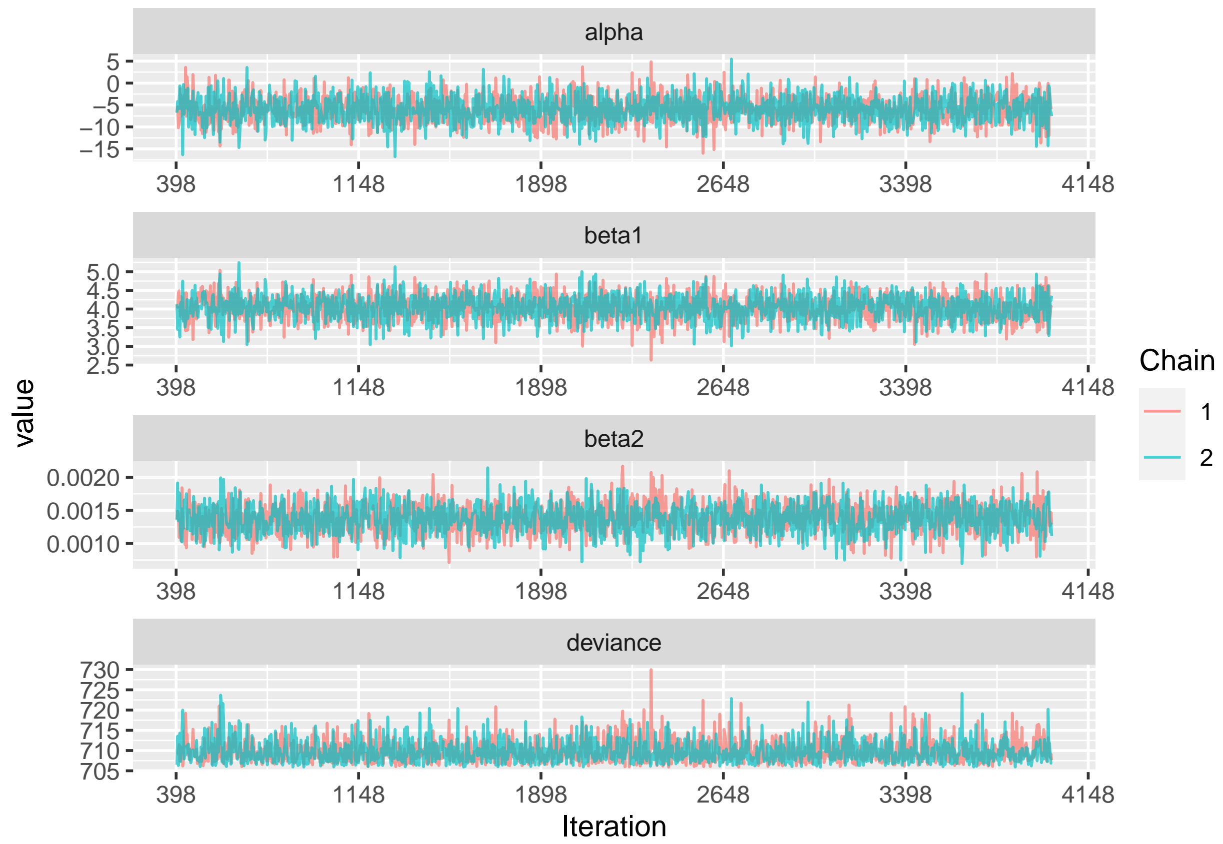
```r
#Convert mcmc object to ggs object
fit.gg<-ggs(fit.mcmc)


#Individual plots

#Histogram
ggs_histogram(fit.gg)
```



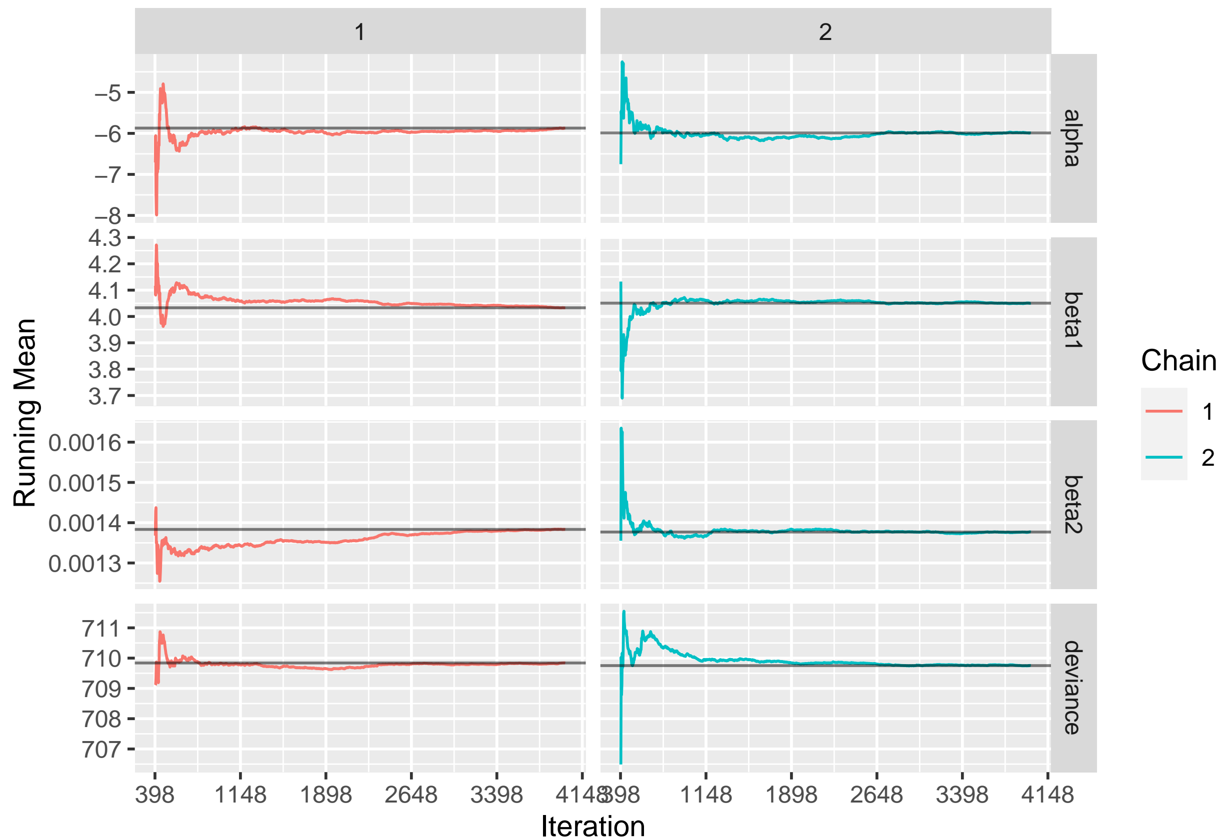

```
#Density
ggs_density(fit.gg)
```




```
#Traceplots (useful for convergence diagnostics)
ggs_traceplot(fit.gg)
```




```r
ggs_running(fit.gg)
```



Getting the proportion of the distribution above or below zero (depending on the sign), is fairly easy. It uses native R functions. The one caveat is that you need to know the shape of your posterior density. You can know this by using conjugate priors. Below is the code to find the proportion of the distribution above zero for β_1 and β_2 and below zero for α.

```
#For beta 1
1-pnorm(0, mean=mean(fit.df$beta1), sd=sd(fit.df$beta1))
```

```
## [1] 1
```

```
#For beta 2

1-pnorm(0, mean=mean(fit.df$beta2), sd=sd(fit.df$beta2))
```

```
## [1] 1
```

```
#For alpha

pnorm(0, mean=mean(fit.df$alpha), sd=sd(fit.df$alpha))
```

```
## [1] 0.9739701
```

For the rest of this lab, I would like you to estimate the previous model 5 more times all with different priors. I encourage you to try both informative and uninformative priors.