Bayesian Latent Variables in R

Dr. Sarah Hunter

10/19/2021

Software Overview

For this workshop, I will be doing demonstrations in JAGS, which stands for Just Another Gibbs Sampler. JAGS is a software that does Baeysian analysis through R. Therefore, some knowledge of R is preferred. For those that need a refresher, my website contains some help files that cover the basics of R: https://www.sarahghunter.com/r-help.

Installing R

For those of you who do not have R installed on your device, or have not updated in a while, you can find the latest version of R here: https://cran.r-project.org. Having the most up to date version of R is important for many of the packages needed in later stages of Bayesian Analysis. The current version of R is 4.1.1. To either update an existing version or install R for the first time, go to the previous link and click on the appropriate version (Mac v. PC) and install the latest version.

Installing JAGS

JAGS is a free program built by Martyn Plummer. You can find the download site here: https://sourceforge.net/projects/mcmc-jags/files/. Click on the green button that says "Download Latest Version". Please remember to also download and read the User Manual, under "Manuals".

Running JAGS through R

JAGS can run seamlessly through R with the help a a couple R packages.

The JAGS programming language is very similar to R, however there some small differences, especially with the names of distributions. However, it is similar enough to R that anyone with some R training can understand JAGS easily. In order to run JAGS through the R interface, you should download the packages `rjags` and `R2jags` using the usual `install.pacakges()` command.

```
#install.packages("rjags")
#install.packages("R2jags")

library(rjags)
```

```
## Loading required package: coda
```

```
## Linked to JAGS 4.3.0
```

```
## Loaded modules: basemod,bugs
```

```
library(R2jags)
```

```
##
## Attaching package: 'R2jags'
```

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

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

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

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

```
library(tidyr)
```

Bayesian Linear Models

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`. It is simple to convert data from a dataframe format to a list format.

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
```

```
##
## Attaching package: 'car'
```

```
## The following object is masked from 'package:dplyr':
##
##     recode
```

```
library(R2jags)
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

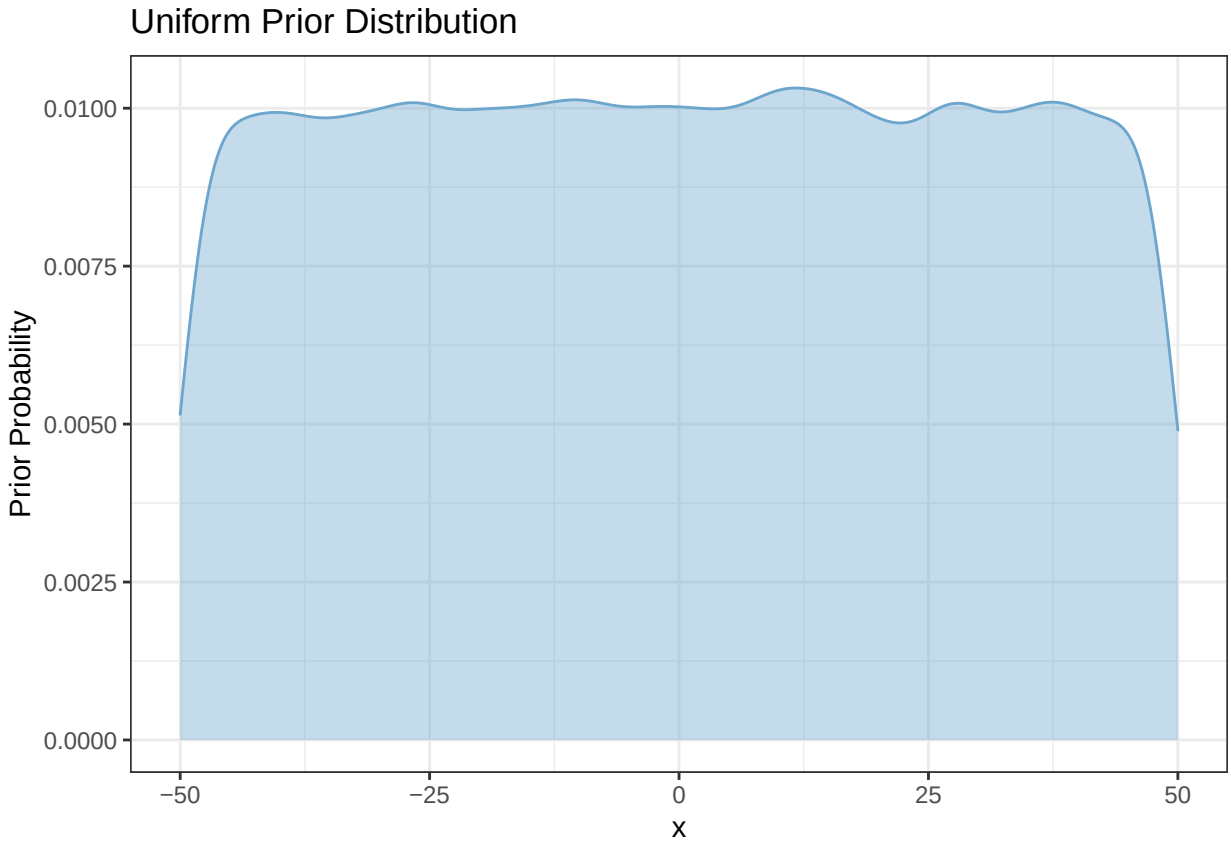
The next step in your model is to write the model. This includes specifying your priors with both the mean and variance. Every stochastic element of the model needs a prior. The one important thing to remember in JAGS is that the second parameter of a distribution is *NOT* the variance. It is actually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

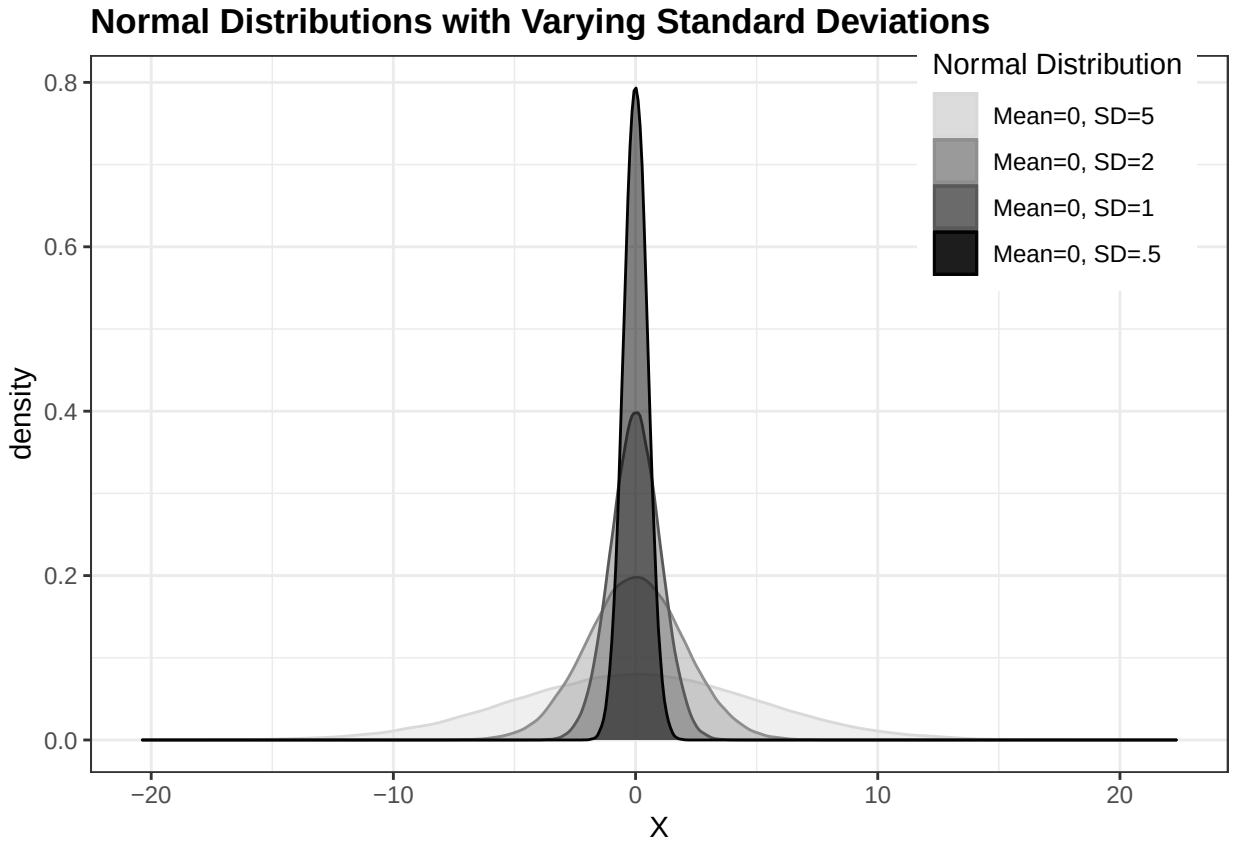
To understand what these distributions do, you can plot them in R using the base plot function. You just need to do many random draws from that distribution using the `r` prefix and plot the density function of those random draws. The `r` prefix outside of JAGS uses different, but related parameters. 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="Uniform Prior Distribution", x="x",
                   y="Prior Probability")
```



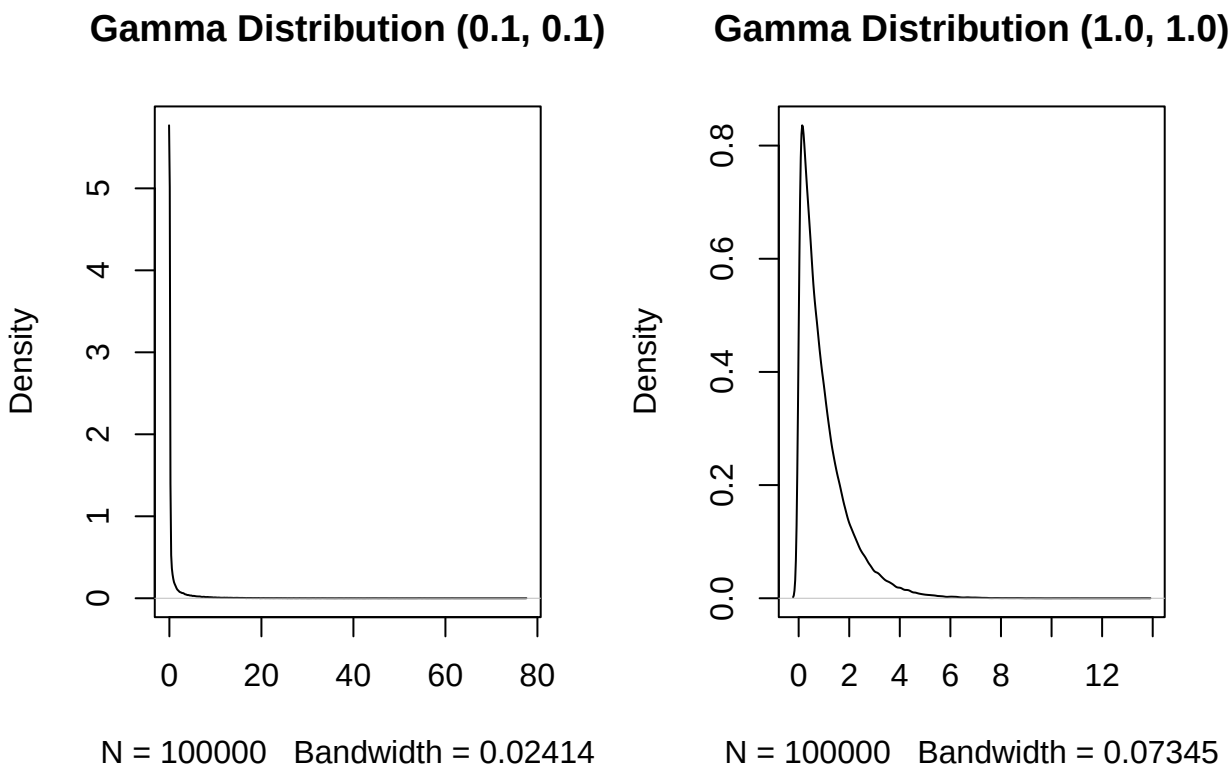
Below is the Normal distribution with the same mean and four different standard deviations.

```
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=norm2, color="norm2",
                                                     fill="norm2"), alpha=.4)+
  geom_density(aes(x=norm3, color="norm3", fill="norm3"), alpha=.4)+
  geom_density(aes(x=norm4, color="norm4", fill="norm4"), alpha=.5)+
  theme_bw()+labs(x="X", title="Normal Distributions with Varying Standard Deviations")+
  scale_colour_manual(name="Normal Distribution",
values=c(norm1="gray85", norm2="gray56", norm3="gray35", norm4="gray1"),
labels=c(norm1="Mean=0, SD=5", norm2="Mean=0, SD=2",norm3="Mean=0, SD=1",
         norm4="Mean=0, SD=.5"))+scale_fill_manual(name="Normal Distribution",
                         values=c(norm1="gray85", norm2="gray56", norm3="gray35",
                                  norm4="gray1"), labels=c(norm1="Mean=0, SD=5",
                                                           norm2="Mean=0, SD=2",
                                                           norm3="Mean=0, SD=1",
                                                           norm4="Mean=0, SD=.5"))+
  theme(plot.title=element_text(face="bold"))+theme(legend.position=c(.85, .85))
```



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:

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

```
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
```

```
print(fit)
```

```
## Inference for Bugs model at "/var/folders/dx/pkx9cyj1089843ljm_jzj3pm0000gn/T//RtmpNoAnRT/model14e113
##  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     -5.806   3.043 -11.778  -7.894  -5.845  -3.743   0.015 1.001  2400
## beta1      4.027   0.339   3.373   3.806   4.027   4.254   4.687 1.001  2400
## beta2      0.001   0.000   0.001   0.001   0.001   0.002   0.002 1.001  2400
## deviance 709.764   2.910 706.207 707.666 709.051 711.158 717.054 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 = 4.2 and DIC = 714.0
## DIC is an estimate of expected predictive error (lower deviance is better).
```

```
# 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//RtmpNoAnRT/model14e113
##  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     -5.806   3.043 -11.778  -7.894  -5.845  -3.743   0.015 1.001  2400
## beta1      4.027   0.339   3.373   3.806   4.027   4.254   4.687 1.001  2400
## beta2      0.001   0.000   0.001   0.001   0.001   0.002   0.002 1.001  2400
## deviance 709.764   2.910 706.207 707.666 709.051 711.158 717.054 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 = 4.2 and DIC = 714.0
## DIC is an estimate of expected predictive error (lower deviance is better).
```

For more information on interpretation and presentation of results, look at my website under the "Summer Bayes Workshop'' Section. There, you will find several R Markdown files like this one that goes into more depth.

Estimating Latent Variable Models with JAGS

Estimating a Bayesian latent variable only requires a series of linear models. With these models, it is important to remember that the latent concept is the *independent variable* that causes the observed indicator. Otherwise, the model flows in much the same way as a simple Bayesian linear model. The same steps apply:

- Prepare the data
- Write the model
- Choose your priors
- Estimate the model with JAGS

In the following sections, I will demonstrate how to create a Bayesian latent variable model from a now defunct past research project. The description, model, model specification, and process for estimating it in JAGS will be discussed.

Setting up the latent concept to be estimated

The research question I had with this project was how population mobility impacted United Nations Emergency Response Funding allocation in response to epidemics (which is now suddenly a very relevant research topic, incidentally). My latent concept here is population mobility. Mobility is a latent concept because it is not something that can be directly observed with one indicator, but is a complex function of legal restrictions (or lack thereof) and the physical infrastructure to actually move people. The observed indicators that I chose to include in my model are: outbound tourism, inbound tourism, traffic in ports, number of air passengers, and Ciri's Freedom of Foreign Movement index (Cingranelli, Richards, and Clay 2014). Other than Freedom of Foreign movement, the data were acquired from the World Bank's World Development Indicators.

Preparing the Data

As above, the data used in a JAGS model must be in a list format. The best way to do this is to load the data file into R as usual, then convert everything to a list. The chunk of code below shows how to do this:

```
#Set a working directory

setwd("/Users/sarahhunter/Documents")

#Load the .csv file

mob<-read.csv("mobilityexample.csv")

#Centering the variables for ease of estimation, then
#getting the variables as objects in R, not dataset

##Centering

library(arm)
```

```
## Loading required package: MASS
```

```
##
## Attaching package: 'MASS'
```

```
## The following object is masked from 'package:dplyr':
##
##     select
```

```
## Loading required package: Matrix
```

```
##
## Attaching package: 'Matrix'

## The following objects are masked from 'package:tidyr':
##
##     expand, pack, unpack

## Loading required package: lme4

##
## arm (Version 1.11-2, built: 2020-7-27)

## Working directory is /Users/sarahhunter/Documents

##
## Attaching package: 'arm'

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

## The following object is masked from 'package:R2jags':
##
##     traceplot

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

```
inb.tour<-rescale(mob$ST.INT.ARVL, "center")
outb.tour<-rescale(mob$ST.INT.DPRT, "center")
ports<-rescale(mob$IS.SHP.GOOD.TU, "center")
air.pass<-rescale(mob$IS.AIR.PSGR, "center")

#Freedom of Foreign movement measured as 0, 1, 2
#0: severely restricted, 1:somewhat restricted, 2:unrestricted
ffm<-mob$FORMOV
N<-length(mob$country)
id<-mob$id


#Make the data into a list:

latent.data<-list("outb.tour", "inb.tour", "ports", "air.pass",
                  "ffm", "N", "id")

latent.data<-list(outb.tour=outb.tour, inb.tour=inb.tour, ports=ports,
                  air.pass=air.pass, ffm=ffm, N=N, id=id)
```

Writing the Model

Latent variables can be written as a series of linear models with the latent concept as the independent variable and the observed indicator as the dependent variable:

$$X^*_{ij} = \Lambda_j \phi_i + \epsilon_{ij}$$

where:

X^*_{ij} = the observed indicator j for observation i

ϕ_i = the estimate of the latent variable for observation i

Λ_j = the factor loading for observed indicator j

ϵ_{ij} = the errors

In the case of our mobility example, we would have a series of equations with the observed indicators: Outbound Tourism, Inbound Tourism, Port Traffic, Air Passengers, and Freedom of Foreign Movement.

Choosing Priors

Often, the most difficult part of estimating a Bayesian Model is choosing the priors. It is generally best to remember that the priors will not have a huge impact on the results of the model if the relationship is strong enough. That being said, a uninformative prior would be fairly standard for this type of model.

Another thing to note about prior distributions in JAGS: as noted above, the normal distribution in JAGS is parameterized with the mean and the precision, not the mean and the standard deviation. Do not forget this, or you will end up with a very informative prior on accident! Also, what is different about a latent variable model is that you need to start with a prior for your latent variable.

For this model that I have used a prior for the mobility latent that is a standard normal distribution (mean=0, precision=1). For the priors on the factor loadings, I used a normal distribution with a mean=0 and precision=.1. For the scale of the variables, this is a fairly uninformative prior. You can always choose a uniform prior, as they can be more straightforward to work with.

Beware the Identification Restrictions Latent variable models have a tendency to flip axes. To prevent this and lock in the direction of the latent variable (more negative means less mobile, more positive means more mobile), You can put an identification restriction on your prior for one factor loading. This truncates the normal distribution and forces it to be positive or negative. You need to be careful choosing this parameter. Rely on prior research and common sense. Here, I place my identification restriction on Air Passengers. It makes sense that the more people travel by air, the more mobile a population is. Therefore, I have forced that particular factor to be positive. You set this restiction using the `;T(0,)` immediately after setting the prior.

Estimating the Model in JAGS

Estimating the model in JAGS can be the most difficult part of the latent variable journey. There are a few steps to take that we will follow in order in this section.

Writing the Model and Choosing the Priors Now, I will estimate the latent variable model in JAGS. Notice that I first set the prior for our latent variable (mobility), then I use a series of linear models (just like the section above) where the latent concept is the independent variable and the observed indicator is the dependent variable. After the models, I set the priors.

I write the entire model in R as a function. This is not the only option. You can always write the model in a plain text file and save it as a .txt file.

```
#Creating the model function

#lat.mob is the model name
lat.mob<-function(){
  for(i in 1:length(id)){
  #setting a prior for the latent variable
    mobility[i]~dnorm(0, 1)

  #First linear model for outbound tourism
```

```
    outb.tour[i]~dnorm(mu2[i], tau[1])
    mu2[i]<- b[1]*mobility[i]

    inb.tour[i]~dnorm(mu3[i], tau[2])
    mu3[i]<- b[2]*mobility[i]

    ports[i]~dnorm(mu4[i], tau[3])
    mu4[i]<- b[3]*mobility[i]

    air.pass[i]~dnorm(mu5[i], tau[4])
    mu5[i]<- b[4]*mobility[i]

    ffm[i]~dnorm(mu6[i], tau[5])
    mu6[i]<- b[5]*mobility[i]
  }
#Setting priors
  for (j in 1:3){
    b[j]~dnorm(0, .1)}
  #Setting and identification restriction on air passengers
    b[4]~dnorm(0, .1);T(0,)
    b[5]~dnorm(0, .1)
 #prior on the precisions (gamma is most appropriate)
     for(j in 1:5){
    tau[j]~dgamma(1, .1)
  }

}
```

Implementing the model Like implementing a linear model, you must include the following features:

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

```
my.params<-c("b", "mobility")
inits.1<-list("b[i]"=c(0))
inits.2<-list("b[i]"=c(0))
inits.mod<-list(inits.1, inits.2)

mod<-jags(data=latent.data, inits=NULL, my.params, n.chains=2, n.iter=1000,
          n.burnin=100, model.file=lat.mob)
```

```
## Warning in jags.model(model.file, data = data, inits = init.values, n.chains =
## n.chains, : Unused variable "N" in data
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 1825
##    Unobserved stochastic nodes: 375
##    Total graph size: 4393
```

```
##
## Initializing model
```

```
print(mod$BUGSoutput$summary[1:7,])
```

```
##                     mean          sd         2.5%          25%          50%
## b[1]           0.4176581  0.02388709    0.3745576    0.4022185    0.4172867
## b[2]           0.4000209  0.02489464    0.3553384    0.3839385    0.3998963
## b[3]           0.3047797  0.02663791    0.2540955    0.2869340    0.3034086
## b[4]           0.4241258  0.02437456    0.3756168    0.4081480    0.4224261
## b[5]          -0.1196038  0.10180268   -0.3106952   -0.1905370   -0.1203355
## deviance    2171.1090311 42.87675721 2109.7710115 2148.8870528 2169.3981464
## mobility[1]   -0.4206592  0.34888145   -1.0935911   -0.6600831   -0.4158647
##                      75%        97.5%     Rhat n.eff
## b[1]           0.43333766    0.4649866 1.043942    40
## b[2]           0.41717068    0.4468524 1.019591    97
## b[3]           0.32225351    0.3574419 1.011017   150
## b[4]           0.44090378    0.4713296 1.032643    52
## b[5]          -0.05120518    0.0771448 1.001014  1800
## deviance    2190.92854661 2238.5885844 1.004435   390
## mobility[1]   -0.18639281    0.2692240 1.000996  1800
```

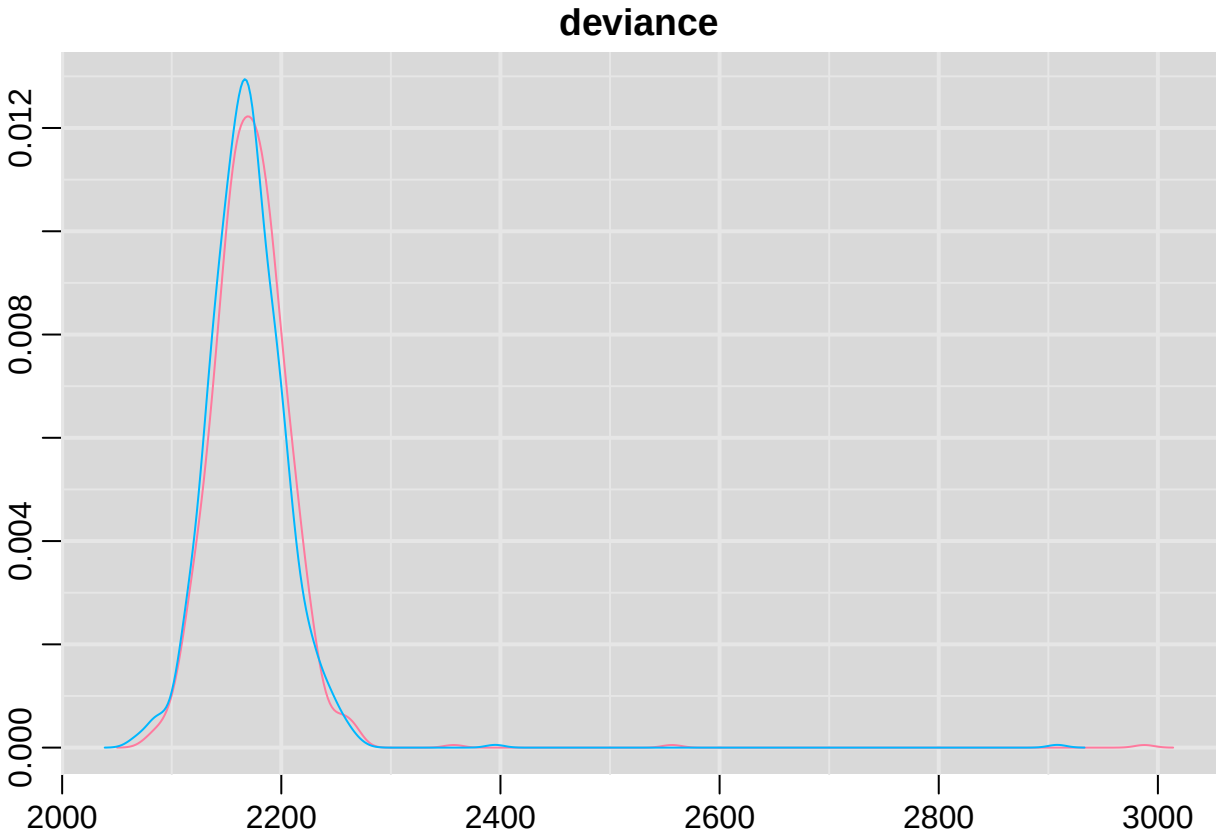
Checking Convergence One additional step in JAGS that you will need to take is to check if your model as converged. Or at least, you need to check to make sure there is no evidence of non-convergence. The quickest way to do so is to use visual diagnostics. For more information on convergence diagnostics, please see my website.

It is best practice to convert the JAGS output into an MCMC or matrix format. This will be helpful also for model presentation later.

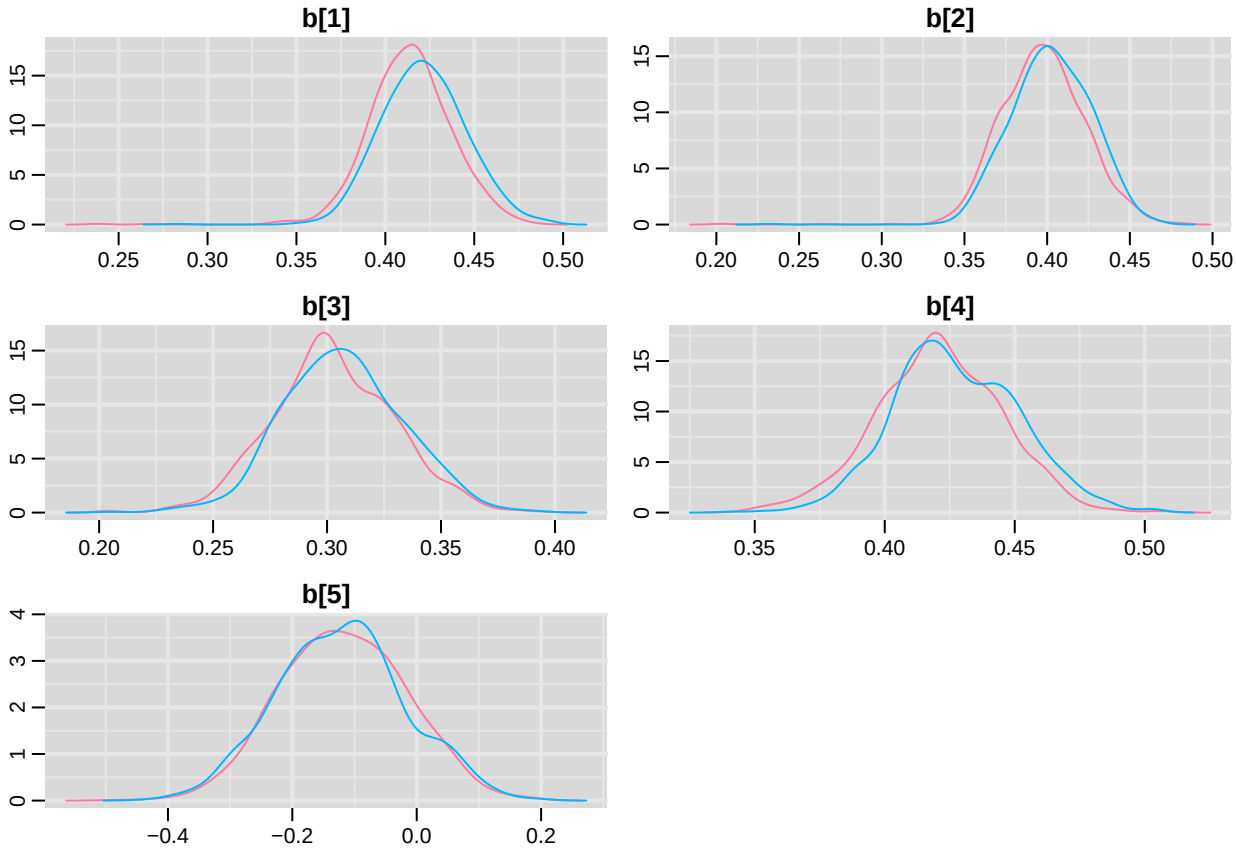
```
#first to an mcmc object
mod.mcmc<-as.mcmc(mod)
#then to a matrix
mod.mat<-as.matrix(mod.mcmc)
#now a dataframe
mod.dat<-as.data.frame(mod.mat)

#install.packages("mcmcplots")
library(mcmcplots)
```

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

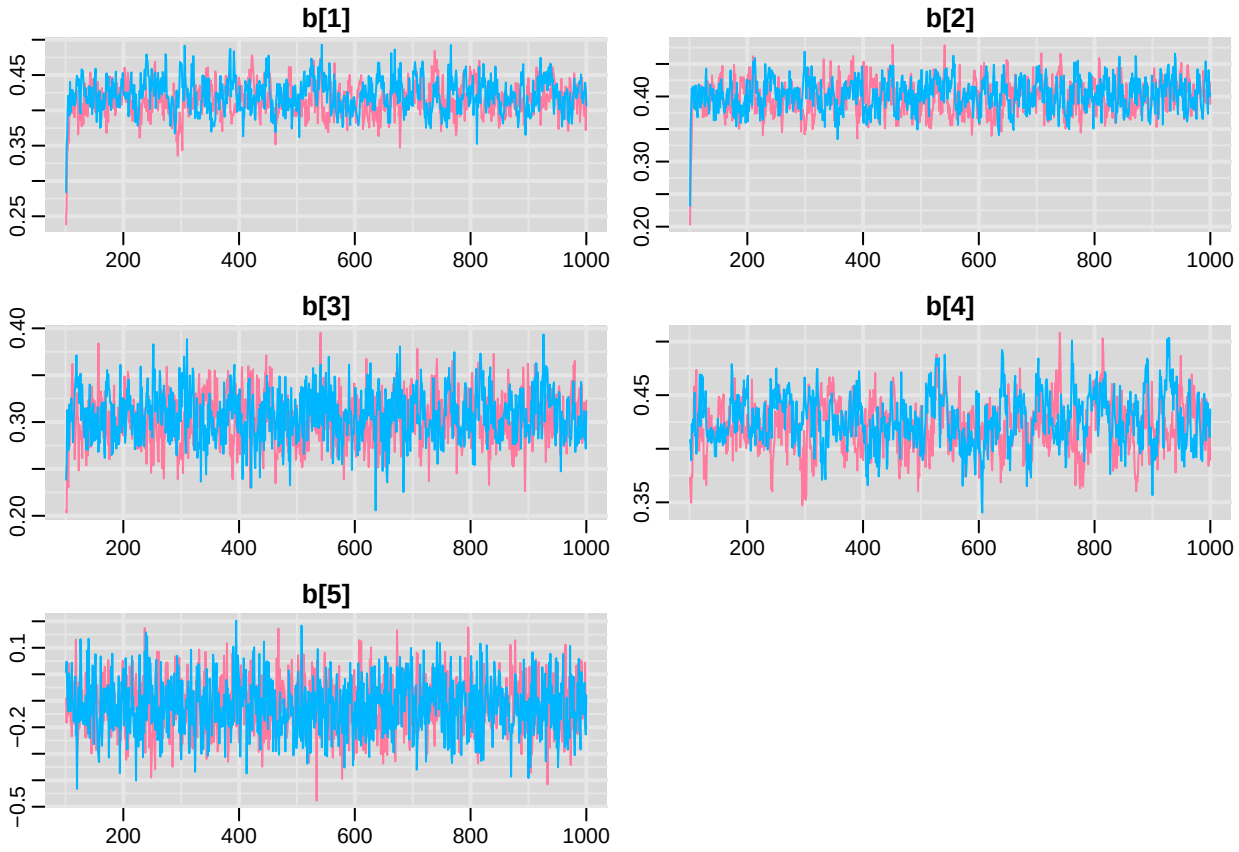
```
denplot(mod.mcmc, parms = c("deviance"))
```




```
denplot(mod.mcmc, parms = c("b"))
```




```
traplot(mod.mcmc, parms = c("b"))
```




```
#traplot(mod.mcmc, parms=c("mobility"))
```

Updating the Model (if needed) If your model has not converged, you can contintue to run the model with the `update` command.

```
mod.upd<-update(mod, n.iter=1000)

print(mod.upd$BUGSoutput$summary[1:7,])
```

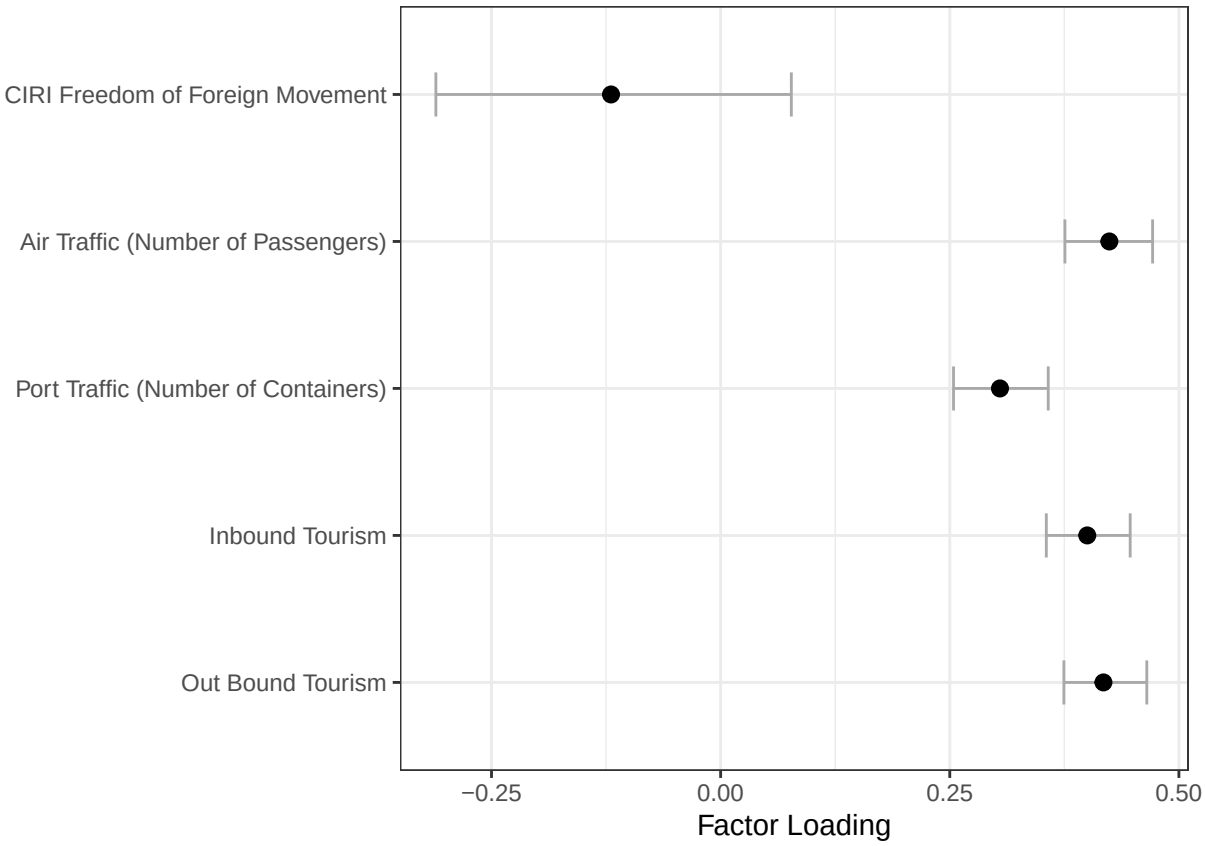
```
##                    mean          sd         2.5%          25%          50%
## b[1]          0.4174842  0.02266433    0.3735167    0.4025854    0.4176325
## b[2]          0.4015677  0.02342996    0.3563452    0.3856191    0.4014088
## b[3]          0.3036428  0.02498255    0.2548683    0.2869355    0.3030473
## b[4]          0.4240031  0.02331837    0.3801311    0.4082734    0.4239206
## b[5]         -0.1227425  0.10188551   -0.3167457   -0.1939473   -0.1211011
## deviance  2168.3363363 32.47495291 2106.1348026 2145.5294137 2167.7604698
## mobility[1]  -0.4044231  0.36033989   -1.1328878   -0.6415673   -0.4049934
##                    75%        97.5%     Rhat n.eff
## b[1]          0.4330461 4.618195e-01 1.003648   510
## b[2]          0.4167100 4.481768e-01 1.010367   890
## b[3]          0.3201094 3.533696e-01 1.000867  2000
## b[4]          0.4389911 4.707178e-01 1.002546   730
## b[5]         -0.0531003 6.879724e-02 1.000859  2000
## deviance  2189.7252469 2.233159e+03 1.002511   750
## mobility[1]  -0.1672115 3.240541e-01 1.000652  2000
```

Presenting Model Results

Now, we have our latent variable model. However, displaying those model results can be challenging. In the following section, I demonstrate a couple of useful figures to displaying the results of a latent variable model.

The first figure is a coefficient plot for the factor loadings. The following code demonstrates the preparation of the model and the actual figure code. For this code, I use the `ggplot2` library.

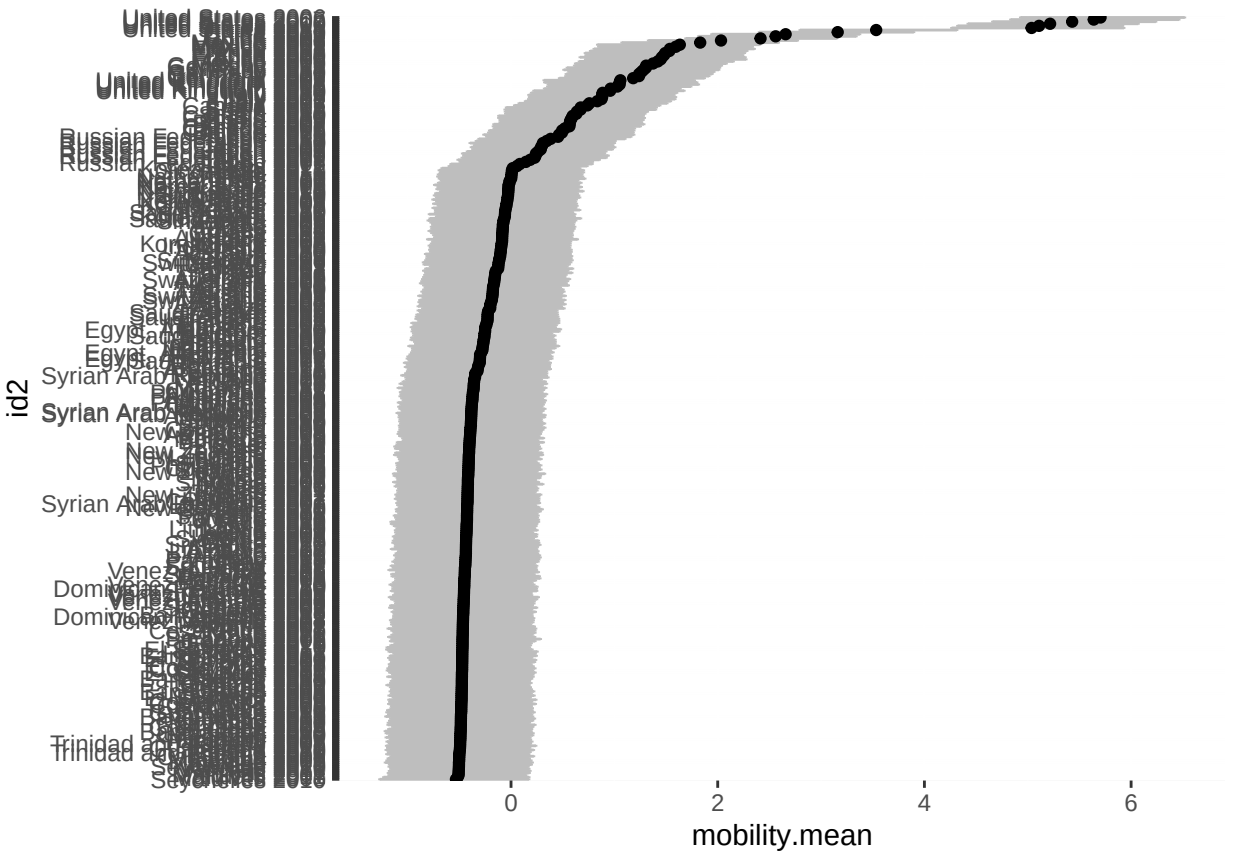
```
#Preparing the coefficients
bs<-mod.dat[,grep("b[", colnames(mod.dat), fixed=T)]
b.mean<-apply(bs, 2, mean)
b.lower<-apply(bs, 2, function(x) quantile(x, probs= c(0.025)))
b.upper<-apply(bs, 2, function(x) quantile(x, probs=c(.975)))
coefs<-colnames(bs)


coef.plot.dat<-data.frame(b.mean, b.lower, b.upper, coefs)


#The figure


library(ggplot2)

ggplot(coef.plot.dat, aes(x=b.mean, y=coefs, xmin=b.lower, xmax=b.upper))+
  geom_errorbar(col="darkgrey", width=.3)+ scale_y_discrete(labels=
      c("Out Bound Tourism", "Inbound Tourism",
        "Port Traffic (Number of Containers)",
        "Air Traffic (Number of Passengers)",
        "CIRI Freedom of Foreign Movement"))+
  geom_point(size=2.5)+ylab("")+xlab("Factor Loading")+theme_bw()
```



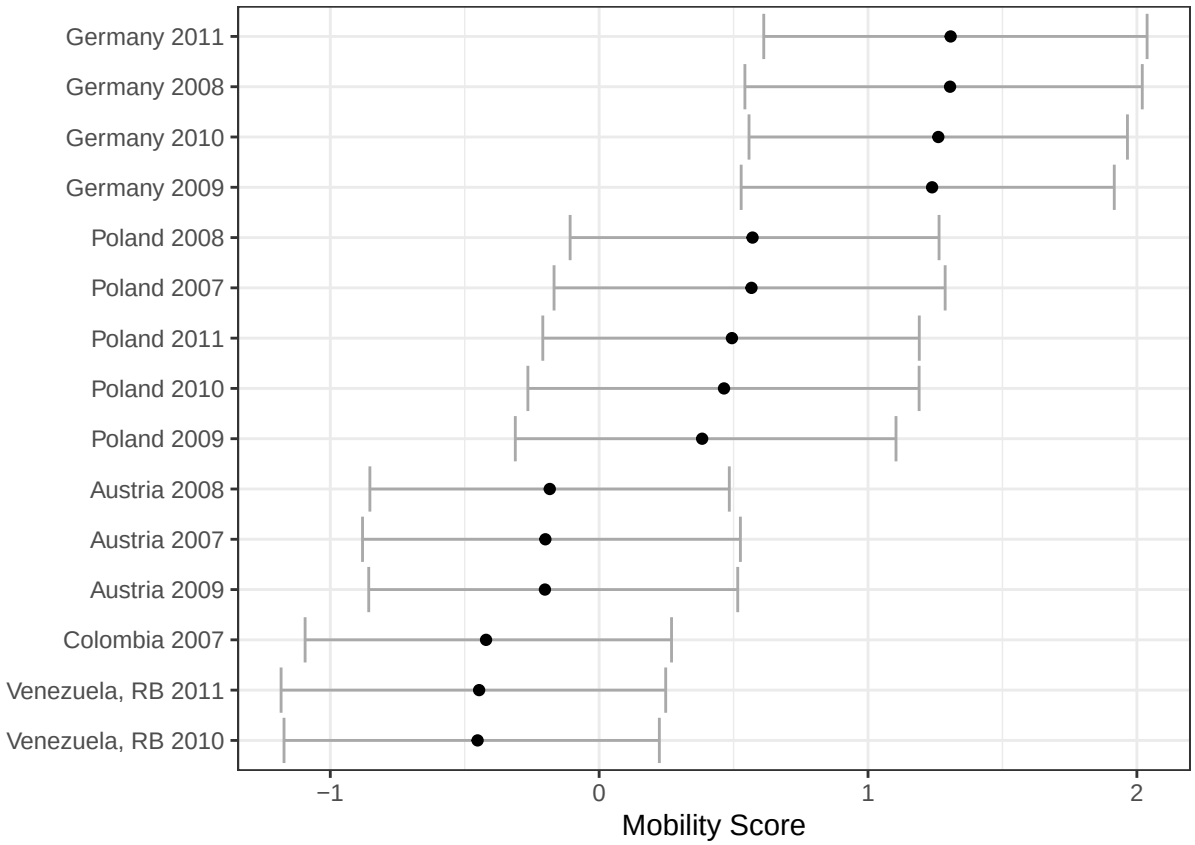
The next part of the presentation is, of course, the reason we are all here, the latent variable values. Typically, we present those as a dot plot. The code below shows how to make this figure.

```
#Prepare the data

#This line selects only the mobility latent scores from the results
mob.dat<-mod.dat[,grep("mobility[", colnames(mod.dat), fixed=T)]

#These lines reshape and aggregate the iterations of the mobility score
#this gets one line per country year observation
mobility.mean<-apply(mob.dat, 2, mean)
mobility.lower<-apply(mob.dat, 2, function(x) quantile(x, probs= c(0.025)))
mobility.upper<-apply(mob.dat, 2, function(x) quantile(x, probs=c(.975)))
cntry<-colnames(mob.dat)


plot.dat<-data.frame(mobility.mean, mobility.lower, mobility.upper, cntry)
plot.dat<-plot.dat[order(plot.dat$cntry), ]



#This line of code gets ride of the mobility[] from the names of the columns.
plot.dat$cntry2<- gsub('\\D', '', plot.dat$cntry)

#This code merges the latent mobility scores with the rest of the data

mob$id2<-as.factor(mob$id)
```

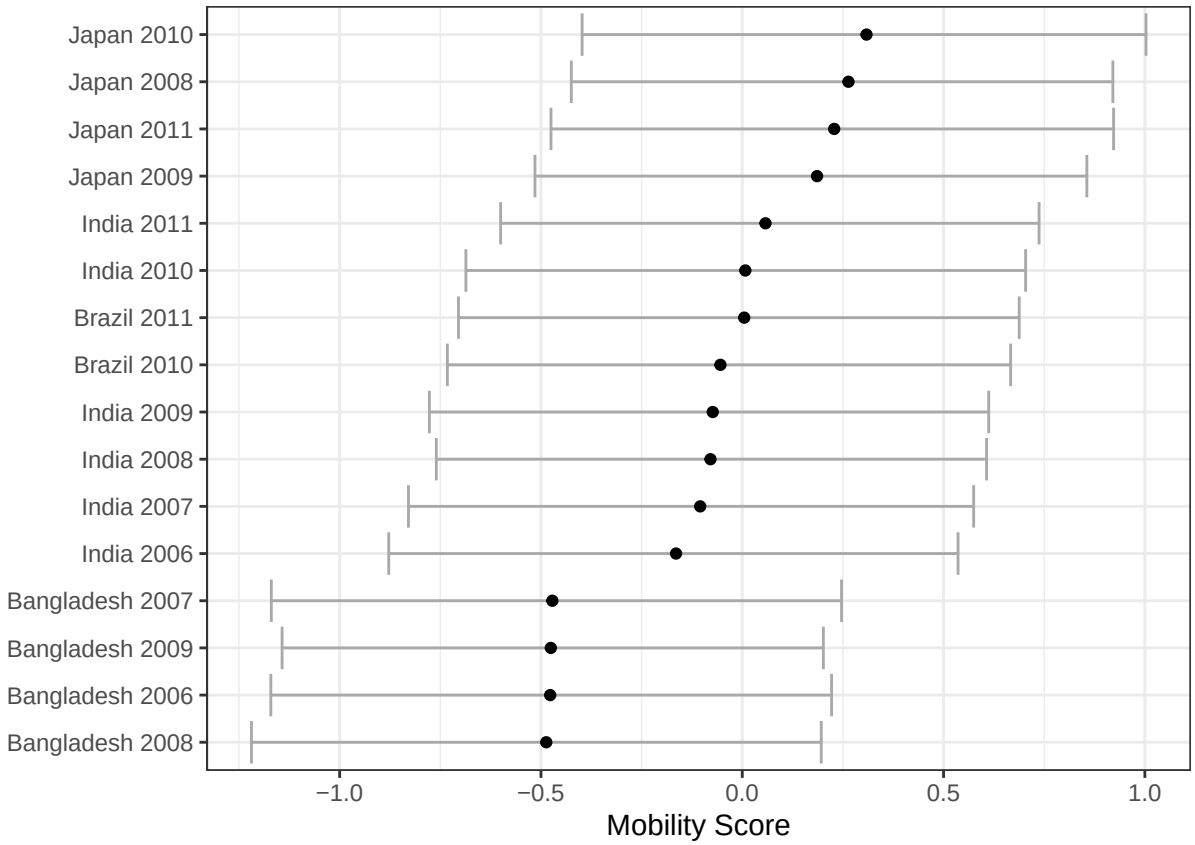
```
better.plot<- plot.dat %>% dplyr::left_join(mob, by=c("cntry2"="id2"))

#Creating a Country-year label
better.plot$Ctry.year<-paste(better.plot$country, better.plot$year, sep=" ")


##Select Mobility Dot Plot

better.plot$id2<- reorder(better.plot$Ctry.year, better.plot$mobility.mean)

#for all observations
ggplot(data=better.plot, aes(x=mobility.mean, y=id2, xmin=mobility.lower,
                             xmax=mobility.upper))+
  geom_errorbar(col="grey")+geom_point(col="black")
```



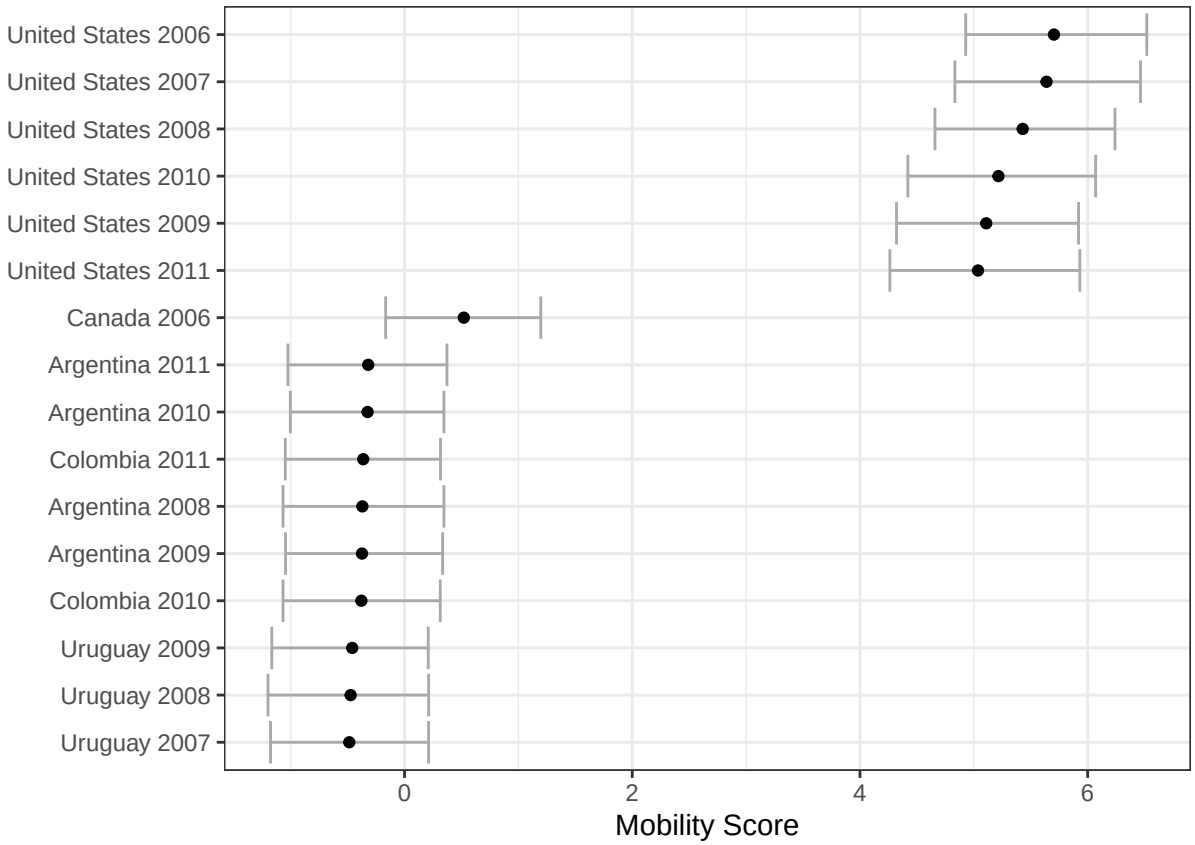

```
#In small batches
ggplot(data=better.plot[1:15,], aes(x=mobility.mean, y=id2,
                                    xmin=mobility.lower, xmax=mobility.upper))+
  geom_errorbar(col="darkgrey")+geom_point(col="black")+theme_bw()+
  ylab("")+xlab("Mobility Score")
```




```
ggplot(data=better.plot[200:215,], aes(x=mobility.mean, y=id2,
                                       xmin=mobility.lower, xmax=mobility.upper))+
  geom_errorbar(col="darkgrey")+geom_point(col="black")+theme_bw()+ylab("")+
  xlab("Mobility Score")
```




```
ggplot(data=better.plot[300:315,], aes(x=mobility.mean, y=id2,
                                       xmin=mobility.lower, xmax=mobility.upper))+
  geom_errorbar(col="darkgrey")+geom_point(col="black")+theme_bw()+ylab("")+
  xlab("Mobility Score")
```


United States 2006
United States 2007
United States 2008
United States 2010
United States 2009
United States 2011
Canada 2006
Argentina 2011
Argentina 2010
Colombia 2011
Argentina 2008
Argentina 2009
Colombia 2010
Uruguay 2009
Uruguay 2008
Uruguay 2007
0
2
4
6
Mobility Score