MICHAEL CLARK
CENTER FOR SOCIAL RESEARCH
UNIVERSITY OF NOTRE DAME

BAYESIAN BASICS

A CONCEPTUAL INTRODUCTION WITH APPLICATION IN R AND STAN

Contents

Preface

Initial draft posted 2014 Summer.
Current draft November 9, 2014.

The following serves as a practical and applied introduction to Bayesian estimation methods for the uninitiated. The goal is to provide just enough information in a brief format to allow one to feel comfortable exploring Bayesian data analysis for themselves, assuming they have the requisite context to begin with. Roughly the idea is to cover a similar amount of material as one would with one or two afternoons in a standard statistics course in various applied disciplines, where statistics is being introduced in general.

After a conceptual introduction, a fully visible by-hand example is provided using the binomial distribution. After that the document proceeds to introduce fully Bayesian analysis with the standard linear regression model, as that is the basis for most applied statistics courses and is assumed to be most familiar to the reader. Model diagnostics, model enhancements, and additional modeling issues are then explored. Supplemental materials provide more technical detail if desired, and include a maximum likelihood refresher, overview of programming options in Bayesian analysis, the same regression model using BUGS and JAGS, and code for the model using the Metropolis-Hastings and Hamiltonian Monte Carlo algorithms.

Prerequisites include a basic statistical exposure such as what would be covered in typical introductory social or other applied science statistics course. At least some familiarity with R is necessary, and one may go through my introductory handout to acquire enough knowledge in that respect. However, note that for the examples here, at least part of the code will employ some Bayesian-specific programming language (e.g. Stan, BUGS, JAGS). No attempt is made to teach those languages though, as it would be difficult to do so efficiently in this more conceptually oriented setting. As such, it is suggested that one follow the code as best they can, and investigate the respective manuals, relevant texts, etc. further on their own. Between the text and comments within the code it is hoped that what the code is accomplishing will be fairly clear.

This document relies heavily on Gelman et al. (2013), which I highly recommend. Other sources used or particularly pertinent to the material in this document can be found in the references section at the end. It was created using the knitr package within RStudio. The layout used allows for visuals and supplemental information to be placed exactly where they should be relative to the text. For plots, many of them have been made intentionally small to allow for this, but one can zoom in on them without loss of clarity.

Introduction

Bayesian analysis is now fairly common in applied work. It is no longer a surprising thing to see it utilized in non-statistical journals, though it is still fresh enough that many researchers feel they have to put 'Bayesian' in the title of their papers when they implement it. However, one should be clear that one doesn't conduct a Bayesian analysis per se. A Bayesian logistic regression is still just logistic regression. The *Bayesian* part comes into play with the perspective on probability that one uses to interpret the results, and in how the estimates are arrived at.

The Bayesian approach itself is very old at this point. Bayes and Laplace started the whole shebang in the 18th and 19th centuries, and even the modern implementation of it has its foundations in the 30s, 40s and 50s of last century[1]. So while it may still seem somewhat newer to applied researchers, much of the groundwork has long since been hashed out, and there is no more need to justify a Bayesian analysis anymore than there is to use the standard maximum likelihood approach[2]. While there are perhaps many reasons why the Bayesian approach to analysis did not catch on until recently, perhaps the biggest is simply computational power. Bayesian analysis requires an iterative and time-consuming approach that simply wasn't viable for most until modern computers. But nowadays, one can conduct such analysis even on their laptop very easily.

[1] Jeffreys, Metropolis etc.

[2] Though some Bayesians might suggest the latter would need *more*.

The Bayesian approach to data analysis requires a different way of thinking about things, but not too much different in implementation. In fact, as we will see later, it incorporates the very likelihood one uses in traditional statistical techniques. The key difference regards the notion of probability, which, while different than Fisherian or frequentist statistics, is actually more akin to how the average Joe thinks about probability. Furthermore, p-values and intervals will have the interpretation that many applied researchers incorrectly think their current methods provide. On top of this one gets a very flexible toolbox that can handle many complex analyses. In short, the reason to engage in Bayesian analysis is that it has a lot to offer and can potentially handle whatever you throw at it.

As we will see shortly, one must also get used to thinking about distributions rather than fixed points. With Bayesian analysis we are not so much as making guesses about specific values as in the traditional setting, but more so understanding the limits of our knowledge and getting a healthy sense of the uncertainty of those guesses.

Bayesian Probability

This section will have about all the math there is going to be in this handout and will be very minimal even then. The focus will be on the conceptual understanding though, and illustrated with a by-hand example.

Conditional probability & Bayes theorem

Bayes theorem is illustrated in terms of probability as follows:

$$p(\mathcal{A}|\mathcal{B}) = \frac{p(\mathcal{B}|\mathcal{A})p(\mathcal{A})}{p(\mathcal{B})}$$

In short, we are attempting to ascertain the conditional probability of $\mathcal{A}$ given $\mathcal{B}$ based on the conditional probability of $\mathcal{B}$ given $\mathcal{A}$ and the respective probabilities of $\mathcal{A}$ and $\mathcal{B}$. This is perhaps not altogether enlightening in and of itself, so we will frame it in other ways followed by a hands on example. For the next depictions we will ignore the denominator.

$$p(hypothesis|data) \propto p(data|hypothesis)p(hypothesis)$$

The $\propto$ means 'proportional to'.

In the above formulation, we are trying to obtain the probability of an hypothesis given the evidence at hand (data) and our initial (prior) beliefs regarding that hypothesis. We are already able to see at least one key difference between Bayesian and classical statistics, namely that classical statistics is only concerned with $p(data|hypothesis)$, i.e. if some (null) hypothesis is true, what is the probability I would see data such as that experienced? While useful, we are probably more interested in the probability of the hypothesis given the data, which the Bayesian approach provides.

Here is yet another way to consider this:

$$posterior \propto likelihood * prior$$

For this depiction let us consider a standard regression coefficient b. Here we have a prior belief about b expressed as a probability distribution. As a preliminary example we will assume perhaps that the distribution is normal, and is centered on some value μ_b and with some variance σ_b^2. The likelihood here is the exact same one used in classical statistics- if y is our response of interest, then the likelihood is $p(y|b)$ as in the standard regression approach using maximum likelihood estimation. What we end up with in the Bayesian context however is not a specific value of b that would make the data most likely, but a probability distribution for b that serves as a sort of weighted combination of the likelihood and prior. Given that *posterior* distribution for b, we

can then get the mean, median, 95% *credible* interval[3] and technically a host of other statistics of interest that might be of interest to us.

To summarize conceptually, we have some belief about the state of the world, expressed as a mathematical model (such as the linear model used in regression). The Bayesian approach provides an updated belief as a weighted combination of prior beliefs regarding that state and the currently available evidence, with the possibility of the current evidence overwhelming prior beliefs, or prior beliefs remaining largely intact in the face of scant evidence.

[3] More on this later.

updated belief = current evidence * prior belief/evidence

A Hands-on Example

Prior, likelihood, & posterior distributions

The following is an attempt to provide a 'by-hand' example to show the connection between prior, likelihood and posterior. Let's say we want to estimate the probability that someone on the road is texting while driving. We will employ the binomial distribution to model this.

Our goal is to estimate a parameter θ, the probability of that a car's driver is texting. You take a random sample of ten cars while driving home, and note the following:

As you passed three cars, the drivers' heads were staring at their laps.

You note two cars that appear to be driving normally.

Another two cars were swerving into other lanes.

Two more cars appear to be driving normally.

At a stoplight, one car wastes 10 seconds of everyone else's time before realizing the stoplight has turned green[4].

[4] Or more than a minute if you don't happen make the light as a result.

We can represent this in R as follows, as well as setup some other things for later.

```
drive = c('texting','texting','texting','not','not',
          'texting','texting','not','not','texting')

# convert to numeric, arbitrarily picking texting=1, not=0
driveNum = ifelse(drive=='texting', 1, 0)
N = length(drive)                    # sample size
nTexting = sum(drive=='texting')     # number of drivers texting
nNot = sum(drive=='not')             # number of those not
```

Recall the binomial distribution where we specify the number of trials for a particular observation and the probability of an event. Let's

look at the distribution for a couple values for θ equal to .5 and .85 and $N = 10$ observations. We will repeat this 1000 times (histograms not shown).

```
x1 = rbinom(1000, size=10, p=.5)
x2 = rbinom(1000, size=20, p=.85)

mean(x1); hist(x1)
mean(x2); hist(x2)

## [1] 5.043
## [1] 17.09
```

We can see the means are roughly around Np as we expect with the binomial.

Prior

For our current situation, we don't know θ and are trying to estimate it. We will start by supplying some possible values.

```
theta = seq(from=1/(N+1), to = N/(N+1), length=10)
```

For the Bayesian approach we must choose a prior representing our initial beliefs about the estimate. I provide three possibilities and note that any one of them would work just fine for this situation. We'll go with a triangular distribution, which will put most of the weight toward values around .5. While we will talk more about this later, I will go ahead and mention that this is where some specifically have taken issue with Bayesian estimation in the past, because this part of the process is too *subjective* for their tastes. Setting aside the fact that subjectivity is an inherent part of the scientific process, and that ignoring prior information (if explicitly available from prior research) would be blatantly unscientific, the main point to make here is that this 'subjective' choice *is not an arbitrary one*. There are countless distributions we might work with, but some will be better for us than others. Again, we'll revisit this topic later.

Choose the prior that makes most sense to you.

```
### prior distribution
# uniform
# pTheta = dunif(theta)

# triangular as in Kruschke
pTheta = pmin(theta, 1-theta)

# beta prior with mean = .5
# pTheta = dbeta(theta, 10, 10)

pTheta = pTheta/sum(pTheta) # Normalize so sum to 1
```

So given some estimate of θ, we have a probability of that value based on our chosen prior.

Likelihood

Next we will compute the likelihood of the data given some value of θ. The likelihood function for the binomial can be expressed as:

$$p(y|\theta) = \binom{N}{k} \theta^k (1-\theta)^{N-k}$$

where N is the total number of possible times in which the event of interest could occur, and k number of times the event of interest occurs. Our maximum likelihood estimate in this simple setting would simply be the proportion of events witnessed out of the total number of samples[5]. We'll use the formula presented above. Technically, the first term is not required, but it serves to normalize the likelihood as we did with the prior.

```
pDataGivenTheta = choose(N, nTexting) * theta^nTexting * (1-theta)^nNot
```

Posterior

Given the prior and likelihood, we can now compute the posterior distribution via Bayes theorem.

```
# first we calculate the denominator from Bayes theorem; this is the marginal
# probability of y
pData = sum(pDataGivenTheta*pTheta)

pThetaGivenData = pDataGivenTheta*pTheta  / pData  # Bayes theorem
```

Now lets examine what all we've got.

```
round(data.frame(theta, prior=pTheta, likelihood=pDataGivenTheta,
                 posterior=pThetaGivenData), 3)

##    theta prior likelihood posterior
## 1  0.091 0.033      0.000     0.000
## 2  0.182 0.067      0.003     0.002
## 3  0.273 0.100      0.024     0.018
## 4  0.364 0.133      0.080     0.079
## 5  0.455 0.167      0.164     0.203
## 6  0.545 0.167      0.236     0.293
## 7  0.636 0.133      0.244     0.242
## 8  0.727 0.100      0.172     0.128
## 9  0.818 0.067      0.069     0.034
## 10 0.909 0.033      0.008     0.002
```

We can see that we've given most of our prior probability to the middle values with probability tapering off somewhat slowly towards either extreme. The likelihood suggests the data is most likely for θ values .55-.64 (again the maximum likelihood estimate for θ is the proportion for the sample, or .6). Our posterior falls somewhere between the two, and we can see it has shifted the bulk of the probability slightly away from center of the prior towards a θ value of .6.

[5] See for yourself in the Binomial Likelihood Example section in the appendix. Note that if we had covariates as in a regression model, we would have different estimates of theta for each observation, and thus would calculate each observation's likelihood and then take their product (or sum their log values, See the Maximum Likelihood Review for further details.). Even here, if you turn this into binary logistic regression with 10 responses of texting vs. not, the 'intercept only' model would be identical to our results here.

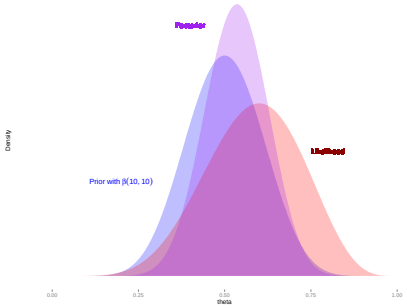
Let's go ahead and see what the mean is:

```
posteriorMean = sum(pThetaGivenData*theta)
posteriorMean

## [1] 0.5624
```

The expected value for a continuous parameter is $\mathrm{E}[X] = \int_{-\infty}^{\infty} xp(x)\mathrm{d}x$, and for a discrete parameter $\mathrm{E}[X] = \sum_{i=1}^{\infty} x_i\, p_i$, i.e. a weighted sum of the possible values times their respective probability of occurrence.

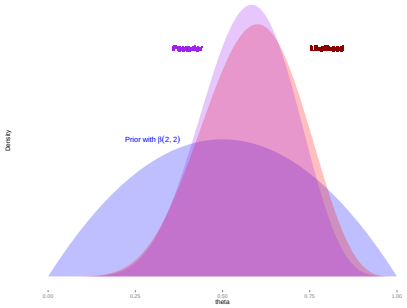
So we start with a prior centered on a value of $\theta = .5$, add data whose ML estimate is $\theta = .6$, and our posterior distribution suggests we end up somewhere in between.

We can perhaps understand this further via the figures at the right. The first is based on a different prior than just used in our example, and instead employs the one with the beta distribution noted among the possibilities in the code above. While the beta distribution is highly flexible, with shape parameters $\mathcal{A}$ and $\mathcal{B}$ set to 10 and 10 we get a symmetric distribution centered on $\theta = .5$. This would actually be a somewhat stronger prior than we might normally want to use, but serves to illustrate a point. The mean of the beta is $\frac{\mathcal{A}}{\mathcal{A}+\mathcal{B}}$, and thus has a nice interpretation as a prior based on data with sample size equal to $\mathcal{A} + \mathcal{B}$. The posterior distribution that results would have a mean somewhere between the maximum likelihood value and that of the prior.

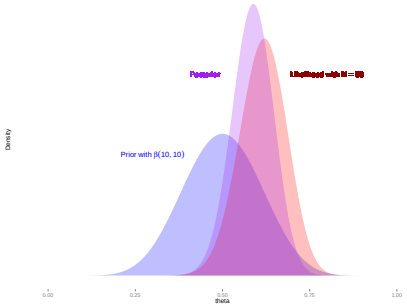


The second utilizes a more diffuse prior of $\beta(2,2)$[6]. The result of using the vague prior is that the likelihood gets more weight with regard to the posterior. In fact, if we used a uniform distribution, *we would be doing the equivalent of maximum likelihood estimation*. As such most of the commonly used methods that implement maximum likelihood can be seen as a special case of a Bayesian approach.

[6] $\beta(1,1)$ is a uniform distribution



The third graph employs the initial $\beta(10,10)$ prior again, but this time we add more observations to the data. Again this serves to give more weight to the likelihood, which is what we want. As scientists, we'd want the evidence, i.e. data, to eventually outweigh our prior beliefs about the state of things the more we have of it.



Posterior predictive distribution

At this point it is hoped you have a better understanding of the process of Bayesian estimation. Conceptually, one starts with prior beliefs about the state of the world and adds evidence to one's understanding, ultimately coming to a conclusion that serves as a combination of evidence and prior belief. More concretely, we have a prior distribution regarding parameters, a distribution regarding the data given those parameters, and finally a posterior distribution that is the weighted combination of the two.

However there is yet another distribution of interest to us- the *posterior predictive distribution*. Stated simply, once we have the posterior

distribution for θ, we can then feed new or unobserved data into the process and get distributions for $\tilde{y}$ [7].

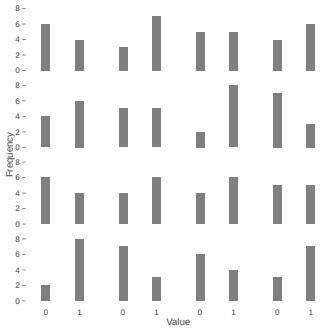
As in Gelman et al. (2013), we can implement the simulation process given the data and model at hand. Where $\tilde{y}$ can regard *any* potential observation, we can distinguish y^{Rep} as the case where we keep to the current data (if a model had explanatory variables X, the explanatory variables are identical for producing y^{Rep} as they were in modeling y, where $\tilde{y}$ might be based on any values $\tilde{X}$). In other words, y^{Rep} is an attempt to replicate the observed data based on the parameters θ. We can then compare our simulated data to the observed data to see how well they match.

When using typical Bayesian programming languages, we could have simulated values of the posterior predictive distribution given the actual values of θ we draw from the posterior. I provide the results of such a process with the graph at right. Each histogram represents a replication of the data, i.e. y^{Rep}, given an estimate of θ.

[7] Mathematically represented as:
$p(\tilde{y}|y) = \int p(\tilde{y}|\theta)p(\theta|y)\mathrm{d}\theta$

We can get a sense of the structure of this process via the following table, taken from Gelman et al. (2013):

Simulation draw	Parameters			Predictive Quantities		
	θ_1	$\cdots$	θ_k	$\tilde{y}_1$	$\cdots$	$\tilde{y}_n$
1	θ_1^1	$\cdots$	θ_k^1	$\tilde{y}_1^1$	$\cdots$	$\tilde{y}_n^1$
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
1	θ_1^S	$\cdots$	θ_k^S	$\tilde{y}_1^S$	$\cdots$	$\tilde{y}_n^S$



Regression Models

Now armed (hopefully) with a conceptual understanding of the Bayesian estimation process, we will actually investigate a regression model using the Bayesian approach. To keep things simple we start with a standard linear model akin to ordinary least squares regression. Later we will show how easy it can be to add changes to the sampling distribution or priors for alternative modeling techniques. But before getting too far you should peruse the Modeling Languages section of the appendix to get a sense of some of the programming approaches available. We will be using the programming language Stan via R and the associated R package rstan.

Linear Regression Model Example

In the following we will have some initial data set up and also run the model using the standard `lm` function for later comparison. I choose simulated data so that not only should you know what to expect from the model, it can easily be modified to enable further understanding. I will also use some matrix operations, and if these techniques are unfamiliar you'll perhaps want to do some refreshing or learning on your own before hand.

Setup

First we need to create the data we'll use here and for most of the other examples in this document.

```
# set seed for replicability
set.seed(8675309)

# create a Nxk matrix of covariates
N = 250
K = 3

covariates = replicate(K, rnorm(n=N))
colnames(covariates) = c('X1', 'X2', 'X3')

# create the model matrix with intercept
X = cbind(Intercept=1, covariates)

# create a normally distributed response that is a function of the covariates
coefs = c(5,.2,-1.5,.9)
mu = X %*% coefs
sigma = 2
y <- rnorm(N, mu, sigma)

# same as
# y = 5 + .2*X1 - 1.5*X2 + .9*X3 + rnorm(N, mean=0, sd=2)

# Run lm for later comparison; but go ahead and examine now if desired
modlm = lm(y~., data=data.frame(X[,-1]))
# summary(modlm)
```

Just to make sure we're on the same page, at this point we have 3 covariates, and a y response that is a normally distributed function of them with standard deviation equal to 2. The population values for the coefficients including the intercept are 5, .2, -1.5, and .9, though with the noise added, the actual estimated values for the sample are slightly different. Now we are ready to set up an R list object of the data for input into Stan, as well as the corresponding Stan code to model this data. I will show all the Stan code, which is implemented in R via a single character string, and then provide some detail on each corresponding model block. However, the goal here isn't to provide a tutorial on Stan, as you might prefer BUGS or JAGS, and related code for this same model is shown in appendix, e.g. BUGS Example for those languages[8]. I don't think there is an easy way to learn these programming languages except by diving in and doing them yourself with models and data you understand. Furthermore, the focus here is on concepts over tools.

[8] In general their modeling syntax is very similar.

The data list for Stan should include any matrix, vector, or value of interest that might be used in the Stan code. For example, along with the data one can include things like sample size, group indicators (e.g. for mixed models) and so forth. Here we can get by with just the N, the number of columns in the model matrix, the response and the

model matrix itself.

```
# Create the data list object for stan inupt
dat = list(N=N, K=ncol(X), y=y, X=X)
```

Next comes the Stan code. In R2OpenBugs or rjags one would call a separate text file with the code, and one can do the same with rstan, but it isn't necessary. The first thing to note is that the model code is one single character string. Next, Stan has programming blocks that have to be called in order. I will have all of the blocks in the code to note their order and discuss each in turn, even though we won't use them all. Anything following a // or between /* */ are comments pertaining to the code. Assignments in Stan are as in R, via **<-**, while distributions are specified with a ~, e.g. `y ~ normal(0, 1)`.

The primary goal here again is to get to the results and beyond, but one should examine the Stan manual for details about the code. In addition, to install rstan one will need to do so outside of the usual CRAN repository process, and the developers provide a quickstart guide to get you going. It does not require a separate installation of Stan itself, but it does take a couple steps and does require a C++ compiler[9]. Once you have rstan installed it is called like any other R package as will see shortly.

[9] You can examine this list for possibilities. Note that you may already have one incidentally. Try the Stan test in their getting started guide before downloading one.

```
# Create the stan model object using Stan's syntax
stanmodelcode = "
data {                          // Data block
  int<lower=1> N;               // Sample size
  int<lower=1> K;               // Dimension of model matrix
  matrix[N, K] X;               // Model Matrix
  vector[N] y;                  // Response
}

/*
transformed data {              // Transformed data block. Not used presently.
}
*/

parameters {                    // Parameters block
  vector[K] beta;               // Coefficient vector
  real<lower=0> sigma;          // Error scale
}

model {                         // Model block
  vector[N] mu;
  mu <- X * beta;               // Creation of linear predictor

  // priors
  beta ~ normal(0, 10);
  sigma ~ cauchy(0, 5);         // With sigma bounded at 0, this is half-cauchy

  // likelihood
  y ~ normal(mu, sigma);
}

/*
generated quantities {          // Generated quantities block. Not used presently.
}
*/
"
```

Stan Code

The first section is the *data block*, where we tell Stan the data it should be expecting from the data list. It is useful to put in bounds as a check on the data input, and that is what is being done between the < > (e.g. we should at least have a sample size of 1). The first two variables declared are N and K, both as integers. Next the code declares the model matrix and response vector respectively. As you'll note here and for the next blocks, we declare the type and dimensions of the variable and then its name. In Stan, everything declared in one block is available to subsequent blocks, but those declared in a block may not be used in earlier blocks. Even within a block, anything declared, such as N and K, can then be used subsequently, as we did to specify dimensions.

The *transformed data block* is where you could do such things as log or center variables and similar, i.e. you can create new data based on the input data or just in general. If you are using R though, it would likely be easier to do those things in R first and just include them in the data list. You can also declare any unmodeled parameters here.

The primary parameters of interest that are to be estimated go in the *parameters block*. As with the data block you can only declare such variables, you cannot make any assignments. Here we note the β and *sigma* to be estimated, with a lower bound of zero on the latter. You might prefer to split out the intercept to be modeled separately as it will often be on a notably different scale than the other coefficients.

The *transformed parameters block* is where optional parameters of interest might be included. What might go here is fairly open, but for efficiency's sake you will typically want to put things only of specific interest that are dependent on the parameters block. These are evaluated along with the parameters, so if not of specific interest you can generate them in the model or generated quantities block to save time.

The *model block* is where your priors and likelihood are specified, along with the declaration of any variables necessary. As an example, the linear predictor is included here, as it will go towards the likelihood[10]. Note that we could have instead put the linear predictor in the transformed parameters section, but this would slow down the process, and again, we're not so interested in those specific values.

I use a normal prior for the coefficients with a zero mean and a very large standard deviation to reflect my notable ignorance here[11]. For the σ estimate I use a Cauchy distribution[12]. Many regression examples using BUGS will use an inverse gamma prior, which is perfectly okay for this model, though it would not work so well for other variance parameters. Had we not specified anything for the prior distribution for the parameters, vague (discussed more in the Choice of Prior),

From the Stan manual, variables and their associated blocks:

Variable Kind	Declaration Block
unmodeled data	data, transformed data
modeled data	data, transformed data
missing data	parameters, transformed parameters
modeled parameters	parameters, transformed parameters
unmodeled parameters	data, transformed data
generated quantities	parameters, transformed parameters, generated quantities

[10] The position within the model block isn't crucial. I tend to like to do all the variable declarations at the start, but others might prefer to have them under the likelihood heading at the point they are actually used.

[11] By setting the prior mean to zero, this will have the effect of shrinking the coefficients toward zero to some extent. In this sense, it is equivalent to penalized regression in the non-Bayesian setting, ridge regression in particular.

[12] Actually a half-Cauchy as it is bounded to be positive.

uniform distributions would be the default. The likelihood is specified in pretty much the same manner as we did with R. BUGS style languages would actually use dnorm as in R, though Stan uses 'normal' for the function name.

Finally, we get to the *generated quantities block*, which is kind of a fun zone. *Anything* you want to calculate can go here- predictions on new data, ratios of parameters, how many times a parameter is greater than x, transformations of parameters for reporting purposes, and so forth. We will demonstrate this later.

Running the Model

Now that we have an idea of what the code is doing, let's put it to work. Bayesian estimation, like maximum likelihood, starts with initial guesses as starting points and then runs in an iterative fashion, producing simulated draws from the posterior distribution at each step, and then correcting those draws until finally getting to some target, or stationary distribution. This part is key and different from classical statistics. We are aiming for a distribution, not a point estimate.

The simulation process is referred to as *Markov Chain Monte Carlo*, or MCMC for short. The specifics of the simulation process are what sets many of the Bayesian programming languages/approaches apart, and something we will cover in more detail in a later section (see Sampling Procedure). In MCMC, all of the simulated draws from the posterior are based on and correlated with the previous[13], as the process moves along the path toward a stationary distribution. Typically we will allow the process to 'warm up', or rather get a bit settled down from the initial starting point, which might be way off, and thus the subsequent estimates will also be way off for the first few iterations. Rest assured, assuming the model and data are otherwise ok, the process will get to where it needs to go. However, as a further check, we will run the whole thing multiple times, i.e. have more than one *chain*. As the chains will start from different places (sometimes only very slightly so), if multiple chains get to the same place in the end we can feel more confident about our results.

[13] In a Markov Chain, θ_t is independent of previous $\theta_{t-2...t_1}$, conditional on θ_{t-1}.

While this process may sound like it might take a long time to complete, for the following you'll note that it will likely take more time for Stan to compile its code to C++ than it will to run the model[14], and on my computer each chain only takes a little over three seconds. However it used to take a very long time even for a standard regression such as this, and that is perhaps the primary reason why Bayesian analysis only caught on in the last couple decades; we simply didn't have the machines to do it efficiently. Even now though, for highly complex models and large data sets it can still take a long time to run, though

How far one wants to go down the rabbit hole regarding MCMC is up to the reader. A great many applied researchers do classical statistical analysis without putting much thought into the actual maximum likelihood estimation process, and I suppose one could do so here as well.

[14] Not usually the case except for simple models with smaller data.

typically not prohibitively so.

In the following code, we note the object that contains the Stan model code, the data list, how many iterations we want (12000), how long we want the process to run before we start to keep any estimates (warmup=2000), how many of the post-warmup draws of the posterior we want to keep (thin=10 means every tenth draw), and the number of chains (3). In the end we will have three chains of 1000 draws from the posterior distribution of the parameters. Stan spits out a lot of output to the R console even with `verbose = FALSE`, and I omit it here, but you will see some initial info about the compiling process, updates as each chain gets through 10% of iterations specified in the `iter` argument, and finally an estimate of the elapsed time. You may also see *informational messages* which, unless they are highly repetitive, can be ignored for the most part[15].

$\frac{12000-2000}{10} = 1000$

[15] The developers are still deciding on the best way to be informative without suggesting something has gone wrong with the process.

```
library(rstan)

### Run the model and examine results ###
fit = stan(model_code=stanmodelcode, data=dat, iter=12000,
           warmup=2000, thin=10, chains=3)
```

With the model run, we can now examine the results. In the following, we specify the digit precision to display, which parameters we want (not necessary here), and which quantiles of the posterior draws we want, which in this case are the median and those that would produce a 95% interval estimate.

```
# summary
print(fit, digits_summary=3, pars=c('beta','sigma'),
      probs = c(.025, .5, .975))

## Inference for Stan model: stanmodelcode.
## 3 chains, each with iter=12000; warmup=2000; thin=10;
## post-warmup draws per chain=1000, total post-warmup draws=3000.
##
##           mean se_mean    sd   2.5%    50%  97.5% n_eff  Rhat
## beta[1]  4.894   0.002 0.132  4.630  4.896  5.144  3000 1.001
## beta[2]  0.085   0.002 0.131 -0.178  0.086  0.340  3000 1.001
## beta[3] -1.471   0.002 0.127 -1.716 -1.471 -1.221  2795 1.000
## beta[4]  0.819   0.002 0.121  0.576  0.820  1.057  3000 0.999
## sigma    2.032   0.002 0.091  1.862  2.029  2.215  2997 0.999
##
## Samples were drawn using NUTS(diag_e) at Mon May 26 10:29:05 2014.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

So far so good. The mean estimates reflect the mean of posterior draws for the parameters of interest, and are the typical coefficients reported in standard regression analysis. The 95% *probability*, or, *credible* intervals are worth noting, because *they are not confidence intervals as you know them*. There is no repeated sampling interpretation here,

i.e. that if we'd done the study exactly the same over and over, and calculated a confidence interval each time, 95% of them would capture the true value, and this is just one of those CIs. The probability interval is more intuitive. It means simply that, based on the results of this model, there is a 95% chance the 'true' value will fall between those two points. The other values printed out I will return to in just a moment.

Comparing the results to those from R's `lm` function, we can see we obtain similar estimates. Had we used uniform priors[16], we would doing essentially the same as what is being done in standard maximum likelihood estimation. Here, we have a decent amount of data for a model that isn't complex, so we would expect the likelihood to notably outweigh the prior, as we demonstrated previously with our binomial example.

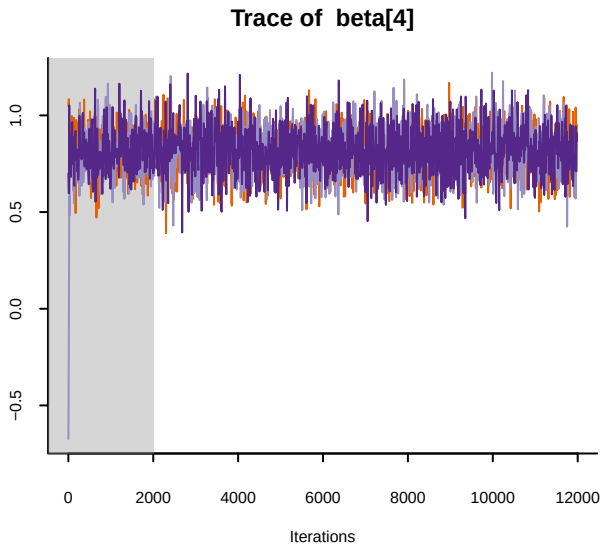
[16] In Stan code this can be done by not explicitly specifying a prior.

```
summary(modlm)

##
## Call:
## lm(formula = y ~ ., data = data.frame(X[, -1]))
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -6.863 -1.470  0.243  1.421  5.041
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   4.8978     0.1284   38.13  < 2e-16 ***
## V1            0.0841     0.1296    0.65     0.52
## V2           -1.4686     0.1261  -11.64  < 2e-16 ***
## V3            0.8196     0.1207    6.79  8.2e-11 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.02 on 246 degrees of freedom
## Multiple R-squared:  0.452,Adjusted R-squared:  0.446
## F-statistic: 67.8 on 3 and 246 DF,  p-value: <2e-16
```

But how would we know if our model was working out okay otherwise? There are several standard diagnostics, and we will talk in more detail about them in the next section, but let's take a look at some presently. In the summary, `se_mean` is the *Monte Carlo error*, and is an estimate of the uncertainty contributed by only having a finite number of posterior draws. `n_eff` is *effective sample size* given all chains and essentially accounts for autocorrelation in the chain, i.e. the correlation of the estimates as we go from one draw to the next. It actually doesn't have to be very large, but if it was small relative to the total number of draws desired there might be cause for concern. `Rhat` is a measure of how well chains mix, and goes to 1 as chains are allowed to run for an infinite number of draws. In this case, n_eff and Rhat suggest we have good convergence, but we can also examine this visually via a

traceplot.

```
# Visualize
traceplot(fit, pars=c('beta[4]'), cex.main=.75, cex.axis=.5, cex.lab=.5)
```



I only show one parameter for the current demonstration, but one should always look at the traceplots for all parameters. What we are looking for after the warmup period is a 'fat hairy caterpillar' or something that might be labeled as 'grassy', and this plot qualifies as such[17]. One can see that the estimates from each chain find their way from the starting point to a more or less steady state quite rapidly. Furthermore, all three chains, each noted by a different color, are mixing well and bouncing around the same conclusion. The statistical measures and traceplot suggest that we are doing okay.

[17] Like all model diagnostics, we aren't dealing with an exact science.

There are many other diagnostics available in the coda package, and Stan model results can be easily converted to work with it. The following code demonstrates how to get started.

```
library(coda)
betas = extract(fit, pars='beta')$beta
betas = as.mcmc(betas)
plot(betas.mcmc)
```

So there you have it. Aside from the initial setup with making a data list and producing the language-specific model code, it doesn't necessarily take much to running a Bayesian regression model relative to standard models[18]. The main thing perhaps is simply employing a different mindset, and interpreting the results from within that new perspective. For the standard models you are familiar with, it probably won't take too long to be as comfortable here as you were with those, and now you will have a flexible tool to take you into new realms with deeper understanding.

[18] Other R packages would allow for regression models to be specified just like you would with the `lm` and `glm` functions. See the `bayesglm` function in the arm package for example.

Model Checking & Diagnostics

As with modeling in traditional frequentist approaches, it is not enough to simply run a model and get some sort of result. One must examine the results to assess model integrity and have faith in what has been produced.

I wonder how many results have been published on models that didn't converge with the standard MLE. People will often ignore warnings as long as they get a result.

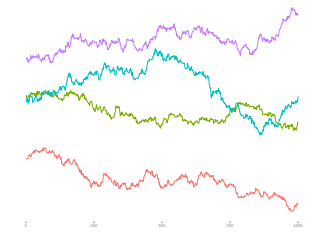
Monitoring Convergence

There are various ways to assess whether or not the model has converged to a target distribution[19], but as with more complex models in MLE, there is nothing that can tell you for sure that you've hit upon *the* solution. As a starting point, the program itself will spit out repeated warnings or errors if something is egregiously wrong, or perhaps take an extraordinary time to complete relative to expectations, if it ever finishes at all. Assuming you've at least gotten past that point, there is more to be done.

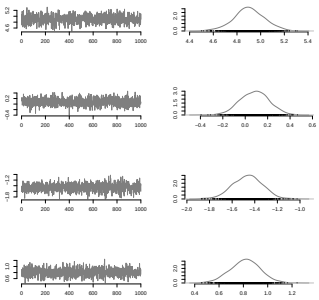
[19] Recall again that we are looking for convergence to a distribution, not a specific parameter.

Visual Inspection: Traceplot & Densities

In the previous model we noted the traceplot for a single parameter, and a visual approach to monitoring convergence is certainly one good method. In general we look for a plot that shows random scatter around a mean value, and our model results suggest that the chains mixed well and the traceplot looked ok. To the right I provide an example where things have gone horribly wrong. The chains never converge nor do they mix with one another. However, one reason for running multiple chains is that any individual chain might converge toward one target, while another chain might converge elsewhere, and this would still be a problem. Also you might see otherwise healthy chains get stuck on occasion over the course of the series, which might suggest more model tweaking or a change in the sampler settings is warranted.



We can examine the mixed chains and density plots of the posterior together with the coda package plot function displayed in the model example. In the Bayesian approach, increasing amounts of data leads to a posterior distribution of the parameter vector approaching multivariate normality. The figure to the right shows the density plots for the regression coefficients along side their respective traceplot for one chain.



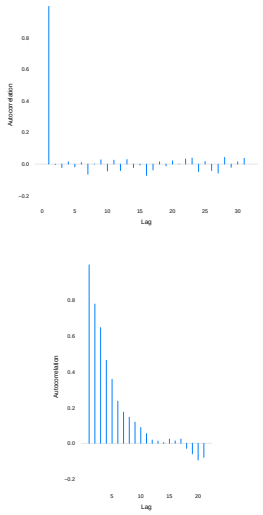
Statistical Measures

To go along with visual inspection, we can examine various statistics that might help our determination of convergence or lack thereof. Gelman and Rubin's *potential scale reduction factor*, $\hat{R}$ provides an estimate of convergence based on the variance of an estimate θ between chains to the variance within a chain. It is interpreted as the factor by which the variance in the estimate might be reduced with longer chains. We are looking for a value near 1 (and at the very least less than 1.1), and it will get there as $N_{sim} \rightarrow \infty$.

The coda package provides other convergence statistics based on Geweke (1992) and Heidelberger and Welch (1983). Along with those statistics, it also has plots for the $\hat{R}$ and Geweke diagnostics.

Autocorrelation

As noted previously, each estimate in the MCMC process is serially correlated with the previous estimates by definition. Furthermore, other aspects of the data, model, and estimation settings may contribute to this. Higher serial correlation typically has the effect of requiring more samples in order to get to a stationary distribution we can feel comfortable with. If inspection of the traceplots look more 'snake-like' than like a fat hairy caterpillar, this might suggest such a situation, and that more samples are required. We can also look at the autocorrelation plot (e.g. in the coda package), in which the chain's correlation with successive lags of the chain are plotted. The first plot to the right is the autocorrelation plot from our model (starting at lag 1). The correlation is low to begin with and then just bounces around zero after. The next plot shows a case of high serial correlation, where the correlation with the first lag is high and the correlation persists even after longer lags. A longer chain with more thinning could help with this.



The effective number of simulation draws is provided as n_{eff} in the Stan output and similarly given in BUGS. We would desire our effect number to equal the number of posterior draws requested. In the presence of essentially no autocorrelation the values would be equal. However this is not a requirement, and technically a low number of draws would still be usable. However a notable discrepancy might suggest there are some issues with the model, or simply that more longer chains could be useful.

Monte Carlo error is an estimate of the uncertainty contributed by only having a finite number of posterior draws. Typically we'd want roughly less than 5% of the posterior standard deviation (reported right next to it in the Stan output), but might as well go for less than 1%. With no autocorrelation it would equal $\sqrt{\frac{var(\theta)}{n_{\text{eff}}}}$[20] and n_{eff} would equal the number of simulation draws requested.

[20] This is the 'naive' estimate the coda package provides in its summary output.

Model Checking

Checking the model for suitability is crucial to the analytical process[21]. Assuming initial diagnostic inspection for convergence has proven satisfactory, we must then see if the model makes sense in its own right. This can be a straightforward process in many respects, and with Bayesian analysis one has a much richer environment in which to do so compared to traditional methods.

[21] Gelman et al. (2013) devotes an entire chapter to this topic to go along with examples of model checking throughout his text. Much of this section follows that outline.

Sensitivity Analysis

Sensitivity analysis entails checks on our model settings to see if changes in them, perhaps even slight ones, result in changes in posterior inferences. This may take the form of comparing models with plausible but different priors, different sampling distributions, or differences in other aspects of the model such as the inclusion or exclusion of explanatory variables. While an exhaustive check is impossible, at least some effort in this area should be made.

Predictive Accuracy & Model Comparison

A standard way to check for model adequacy is simply to investigate whether the predictions on new data are accurate. In general, the measure of predictive accuracy will be specific to the data problem, and involve posterior simulation of the sort covered in the next section. In addition, while oftentimes new data is hard to come by, assuming one has sufficient data to begin with, one could set aside part of it specifically for this purpose. In this manner one trains and tests the model in much the same fashion as machine learning approaches. In fact, one can incorporate the validation process as an inherent part of the modeling process in the Bayesian context just as you would there.

For model comparison of out of sample predictive performance, there are information measures similar to the Akaike Information criterion (AIC), that one could use in the Bayesian environment. One not to use is the so-called Bayesian information criterion, which is not actually Bayesian nor a measure of predictive accuracy. BUGS provides the DIC, or deviance information criterion, as part of its summary output, which is a somewhat Bayesian version of the AIC. More recently developed, the WAIC, or Watanabe-AIC[22], is a more fully Bayesian approach. Under some conditions, the DIC and WAIC measures are asymptotically equivalent to Bayesian leave-one-out cross validation, as the AIC is under the classical setting.

[22] See Gelman et al. (2013) For a review and references. See Vehtari & Gelman (2014) for some more on WAIC as well as Stan code.

Posterior Predictive Checking: Statistical

For an overall assessment of model fit, we can examine how well the model can reproduce the data at hand given the θ draws from the posterior. We discussed earlier the *posterior predictive distribution* for a future observation $\tilde{y}$, $p(\tilde{y}|y) = \int p(\tilde{y}|\theta)p(\theta|y)\mathrm{d}\theta$, and here we'll dive in to using it explicitly. There are two sources of uncertainty in our regression model, the variance σ^2 in y not explained by the model, and posterior uncertainty in the parameters due to having a finite sample size. As $N \rightarrow \infty$, the latter goes to zero, and so we can simulate draws of $\tilde{y} \sim N(\tilde{X}\beta, \sigma^2 I)$ [23]. If $\tilde{X}$ is the model data as in the following, then we will refer to y^{Rep} instead of $\tilde{y}$.

For our model this entails extracting the simulated values from the model object, and taking a random draw from the normal distribution based on the β and σ that are drawn to produce our *replicated data*, y^{Rep}. The following code follows Gelman's example (Gelman et al. (2013, Appendix C)) to obtain this replicated data.

[23] Technically, in the conjugate case the posterior predictive is t-distributed because of the uncertainty in the parameters, though it doesn't take much sample size for simple models to get to approximately normal. Conceptually, with $\hat{\beta}$ being our mean β estimate from the posterior, this can be represented as: $\tilde{y} \sim t(\tilde{X}\hat{\beta}, \sigma^2 + \text{parUnc}, \text{df} = N - k)$

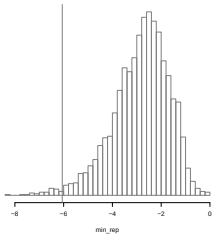
```
# extract the simulated draws from the posterior and note the number for nsims
theta = extract(fit)
betas = theta$beta
sigmas = theta$sigma
nsims = length(theta$sigma)

# produce the replications and inspect
yRep = sapply(1:nsims, function(s) rnorm(250, X%*%betas[s,], sigmas[s]))
str(yRep)

##  num [1:250, 1:3000] 6.68 7.72 3.47 1.17 7.77 ...
```

With the y^{Rep} in hand we can calculate all manner of statistics that might be of interest.

As a starting point, we can check our minimum value among the replicated data sets versus that observed in the data.

```
min_rep = apply(yRep, 2, min)
min_y = min(y)
hist(min_rep, main=''); abline(v=min_y)
c(mean(min_rep), min_y)

## [1] -2.834284 -6.056495

prop.table(table(min_rep>min_y))

##
##      FALSE       TRUE
## 0.01166667 0.98833333

sort(y)[1:5]

## [1] -6.0564952 -3.2320527 -2.6358579 -2.1649084 -0.8366149
```

These results suggest we may be having difficulty picking up the lower tail of the response, as our average minimum is notably higher



than that seen in the data, and almost all our minimums are greater than the observed minimum ($p = .99$). While in this case we know that assuming the normal distribution for our sampling distribution is appropriate, this might otherwise have given us pause for further consideration. A possible solution would be to assume a t distribution for the sampling distribution, which would have fatter tails and thus possibly be better able to handle extreme values. We'll show an example of this later. In this case it is just that by chance one of the response values is extreme relative to the others.

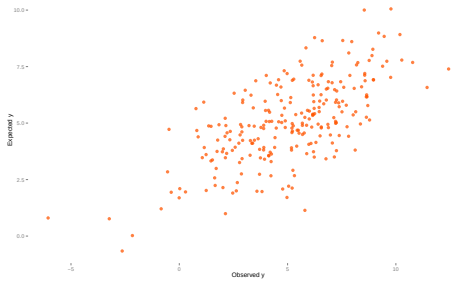
In general we can devise a test statistic, T_{Rep}, and associated p-value to check any particular result of interest based on the simulated data. The p-value in this context is simply the percentage of times the statistic of interest is equal to or more extreme than the statistic, T_y, calculated for the original data. Thus p-values near 0 or 1 are indicative of areas where the model is falling short in some fashion. Sometimes T_y may be based on the θ parameters being estimated, and thus you'd have a T_y for every posterior draw. In such a case one might examine the scatterplot of T_{Rep} vs. T_y, or examine a density plot of the difference between the two. In short, this is an area where one can get creative. However, it must be stressed that we are not trying to accept or reject a model hypothesis as in the frequentist setting- we're not going to throw a model out because of an extreme p-value in our posterior predictive check. We are merely trying to understand the model's shortcomings as best we can, and make appropriate adjustments if applicable. It is often the case that the model will still be good enough for practical purposes.

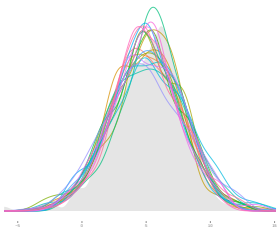
Posterior Predictive Checking: Graphical

As there are any number of ways to do statistical posterior predictive checks, we have many options for graphical inspection as well. As a starting point I show a graph of our average fitted value versus the observed data. The average is over all posterior draws of θ.

Next, I show density plots for a random sample of 20 of the replicated data sets along with that of the original data (shaded). In general it looks like we're doing pretty well here. The subsequent figure displays the density plot for individual predictions for a single value of y from our data. While it looks like some predictions were low for that value, in general the model captures this particular value of the data decently.

We can also examine residual plots of $y - E(y|X, \theta)$ as with standard analysis, shown as the final two figures to the right. The first shows such *realized* residuals, so-called as they are based on a posterior draw of θ rather than point estimation of the parameters, versus





the expected values from the model. The next plot shows the average residual against the average fitted value. No discernible patterns are present, so we may conclude that the model is not entirely inadequate.

The two plots directly above replicate the figures in 6.11 in Gelman et al. (2013).

Summary

As with standard approaches, every model should be checked to see whether it holds up under scrutiny. The previous suggests only a few ways one might go about checking whether the model is worthwhile, but this is a very flexible area where one can answer questions beyond model adequacy and well beyond what traditional models can tell us. Not only is this phase of analysis a necessity, one can use it to explore a vast array of potential questions the data presents, and maybe even answer a few.

Model Enhancements

Enhancing and making adjustments to a model can often be straightforward in the Bayesian context, depending on what one wants to accomplish. In other cases there are some things one can do that aren't even available with standard likelihood approaches in the glm/traditional setting. The following shows a few brief examples to give an idea of the possibilities.

Generating New Variables of Interest

We have already seen one way to get at new statistics of interest in the predictive model checking section. Here I show how to do so as part of the modeling process itself. In Stan we can accomplish this via the generated quantities section.

A typical part of linear regression output is R^2, the amount of variance accounted for by the model. To get this in Stan we just have to create the code necessary for the calculations, and place it within the generated quantities section. I only show this part of the model code; everything we had before would remain the same. For comparison I show the corresponding R code. There are a couple of ways to go about this, and I use some of Stan's matrix operations as one approach.

```
stanmodelcodeRsq = "
.
.
.

generated quantities{
  real rss;
  real totalss;
  real R2;
  vector[N] mu;

  mu <- X * beta;
  rss <- dot_self(y-mu);
  totalss <- dot_self(y-mean(y));
  R2 <- 1 - rss/totalss;
}
"
```

Using the results from the model using `lm`, we do the same calculations for `rss` and `totalss`, and note the result is identical to what you'd see in the summary of the model.

```
rss = crossprod(resid(modlm))
totalss = crossprod(y-mean(y))
1-rss/totalss; summary(modlm)$r.squared

##           [,1]
## [1,] 0.4524289
## [1] 0.4524289

# 1-var(resid(modlm))/var(y) # alternatives
# var(fitted(modlm))/var(y)
```

Now we can run the model with added R^2. Note that as before we do not just get a point estimate, but a whole distribution of simulated values for R^2. First the results.

```
print(fitRsq, digits=3, par=c('beta','sigma','R2'), prob=c(.025,.5,.975))

## Inference for Stan model: stanmodelcodeRsq.
## 3 chains, each with iter=12000; warmup=2000; thin=10;
## post-warmup draws per chain=1000, total post-warmup draws=3000.
##
##           mean se_mean    sd   2.5%    50%  97.5% n_eff  Rhat
## beta[1]  4.895   0.002 0.129  4.639  4.897  5.144  3000 1.000
## beta[2]  0.087   0.003 0.131 -0.169  0.086  0.342  2751 1.000
## beta[3] -1.466   0.002 0.125 -1.712 -1.469 -1.219  2826 0.999
## beta[4]  0.821   0.002 0.123  0.584  0.820  1.063  3000 0.999
## sigma    2.028   0.002 0.091  1.858  2.025  2.212  2945 1.000
## R2       0.443   0.000 0.006  0.427  0.445  0.451  2932 1.000
##
## Samples were drawn using NUTS(diag_e) at Sat May 24 13:10:08 2014.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

The nice thing here is that our R^2 incorporates the additional uncertainty in estimating the model parameters, and thus acts like an *adjusted* R^2 [24]. The following is the classical regression adjusted R^2.

[24] See Gelman & Pardoe (2006), Bayesian Measures of Explained Variance.

```
summary(modlm)$adj

## [1] 0.4457512
```

Furthermore, in the Bayesian context we get an interval estimate and everything else we typically get as with other quantities of interest, and the same goes for anything we calculate along the way (e.g. the mu values). In addition, had we wanted, it would be trivial to calculate something like the actual adjusted R^2, the probability that the value is greater than .5, and other things of that nature.

Robust Regression

If we were perhaps concerned about outliers or otherwise more extreme values in our data, we could change the sampling distribution to one that had a little more probability in the tails. This is very easy to do in this situation, as we just change likelihood portion of our code to employ say, a t-distribution. In Stan, the t-distribution has parameters mean and sigma as with the normal distribution, but we also have the added parameter for degrees of freedom. Thus our code might look like the following:

```
stanmodelcodeT = "
.
.
.

model {
  vector[N] mu;
  mu <- X * beta;

  // priors
  beta ~ normal(0, 10);
  sigma ~ cauchy(0, 5);

  // likelihood
  // y ~ normal(mu, sigma);            // previously used normal
  y ~ student_t(10, mu, sigma)         // t with df=10
}
"
```

In this case we set the degrees of freedom at 10[25], but how would you know in advance what to set it as? It might be better to place a prior (with lower bound of one) for that value and estimate it as part of the modeling process. One should note that there are many distributions available in Stan (e.g. others might be useful for skewed data, truncated etc.), and more will be added in the future.

[25] Alternatively, we could add a value 'df' to the data list and data block.

Generalized Linear Model

Expanding from standard linear model, we can move very easily to generalized linear models, of which the standard regression is a special case. The key components are use of a link function that links the linear predictor to the response, and an appropriate sampling distribution for the likelihood.

Let's consider a count model using the Poisson distribution. We can specify the model as follows:

$$y \sim Pois(\lambda)$$
$$g(\lambda) = X\beta$$

where $g(.)$ is the link function, the canonical link function being the natural logarithm. In Stan this can be expressed via the inverse link function, where we exponentiate the linear predictor. Aside from that we simply specify y as distributed Poisson in the same way we used the normal and t-distribution in earlier efforts.

```
stanmodelcodePoisson = "
.
.
.

model {
  vector[N] lambda;
  vector[N] eta;

  eta <- X * beta;
  lambda <- exp(eta)

  // priors
  beta ~ normal(0, 10);


  // likelihood
  y ~ poisson(lambda)
}
"
```

And that's all there is to that[26]. We just saw that we are not limited to the exponential family distributions of glm(s), though that covers a lot of ground, and so at this point you have a lot of the tools covered in standard applied statistics course, and a few beyond.

[26] Note that some link/inverse-link functions in Stan cannot be applied to vectors, only scalars. As such you would have to loop over the values of y, `for(n in 1:N) ...`

Issues

This section highlights generally some things to think about as well as questions that would naturally arise for the applied researcher who might now be ready to start in on their first Bayesian analysis. It pro-

vides merely a taste regarding some select issues, and at this point one should be consulting Bayesian analysis texts directly.

Debugging

An essential part of Bayesian analysis is debugging to see if your code and model are doing what it should be doing[27], and this especially holds for more complex models. For many models and common settings for the number of simulations, Bayesian analysis can still take several minutes on standard computers or laptops. With large data and/or complex models, some might take *days*. In either case, it is a complete waste of time to let broken code/models run unnecessarily.

[27] It really should be a part of most analysis.

The idea with debugging is that, once you think you have everything set up the way you like, run very short attempts to see if A, the code runs at all, and B, whether it runs appropriately. As such, you will only want to set your warm-up and iterations to some small number to begin with, e.g. maybe 200 iterations for warm-up, 1000 post warm-up, and no more than two chains. Sometimes it will be obvious what a problem is, such as a typo resulting in the program of choice not being able to locate the parameter of interest. Others may be fairly subtle, for example, when it comes to prior specification.

Along with shorter runs, one should consider simpler models first, and perhaps using only a subset of the data. Especially for complex models, it helps to build the model up, debugging and checking for problems along the way. As a not too complicated example, consider a mixed model for logistic regression. One could even start with a standard linear model ignoring the binary nature of the response. Getting a sense of things from that and just making sure that inputs etc. are in place, one can supply the inverse logit link and change the sampling distribution to Bernoulli (or binomial of size = 1). Now you can think about adding the random effect, other explanatory variables of interest, and any other complexities that had not been included yet.

As you identify issues, you fix any problems that arise and tinker with other settings. Once you are satisfied, *then* try for the big run. Even then, you might spot new issues with a longer chain, so you can rinse and repeat at that point. BUGS, JAGS, and Stan more or less have this capacity built in with model upgrade functions. For example, in Stan you can feed the previous setup of a model in to the main `stan` function. Use one for your initial runs, then when you're ready, supply the model object as input to the 'fit' argument, perhaps with adjustments to the Monte Carlo settings.

Choice of Prior

Selection of prior distributions might be a bit daunting for the new user of applied Bayesian analysis, but in many cases, and especially for standard models, there are more or less widely adopted choices. Even so, we will discuss the options from a general point of view.

Noninformative, Weakly Informative, Informative

We can begin with *noninformative* priors, which might also be referred to as *vague*, *flat*, *reference*, *objective*, or *diffuse*. The idea is to use something that allows for Bayesian inference but puts all the premium on the data, and/or soi-disant 'objectivity'. As we have alluded to elsewhere, if we put a prior uniform distribution on the regression coefficients (and e.g. the log of σ), this would be a noninformative approach that would essentially be akin to maximum likelihood estimation. One might wonder at this point why we wouldn't just use vague priors all the time and not worry about overly influencing the analysis by the choice of prior.

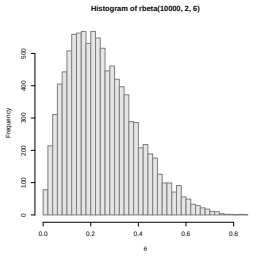
As an example, let's assume a uniform distribution for some parameter θ. A uniform, 'flat' prior is *improper*, i.e. the probability distribution does not integrate to 1. While the posterior distribution may be proper, it is left the the researcher to determine this. One also has to choose a suitable range, something which may not be easy to ascertain. In addition, the distribution may not be uniform on some transformation of the parameter, say θ^2. A *Jeffreys' prior* could be used to overcome this particular issue, but is more difficult for multiparameter settings.

In general there are several issues with using a noninformative or reference prior. For many models there may be no clear choice of what to use. In any case, if the data are sufficient, the prior can't matter, so establishing some reference to be used automatically isn't exactly in keeping with Bayesian thinking. Furthermore, if you had clear prior information from previous research, one should use it. Furthermore, such choices can still have unintended effects on the results. In reality, any prior could be said to be *weakly informative*.

So instead of being completely ignorant, we can choose instead to be mostly ignorant, vague but not too vague. As an example, consider our earlier binomial distribution example (A Hands-on Example). Perhaps a reasonable guess as to the proportion of those texting while driving was .25. With that as a basis, we could choose a Beta distribution that would have roughly 80% of its probability between .1 and .5. We know that lower values for the parameters of a beta distribution represent a less informed state of mind, and the mean of the distribution is A/(A+B) so we could just fiddle with some values to see what

we can turn up. The following code suggests a $\mathcal{B}(2, 6)$ would probably be our best bet. One can examine the distribution to the right.

```
diff(pbeta(c(.1, .5), 1, 3))
diff(pbeta(c(.1, .5), 2, 6))
diff(pbeta(c(.1, .5), 3, 9))

## [1] 0.604
## [1] 0.7878056
## [1] 0.8777233
```



With our regression model we were dealing with standardized predictors, so even choosing a N(0, 10) might be overly vague, though it would be near flat from -1 to 1. The nice part about setting the prior mean on zero is that it has a regularizing effect that can help avoid overfitting with smaller samples.

Thus weakly informative priors can be based on perfectly reasonable settings, and this probably makes more sense than claiming complete ignorance. Just some casual thought in many settings will often reveal that one isn't completely ignorant. Furthermore if we have clear prior information, in the form of prior research for example, we can then use *informative* priors based on those results. This again would be preferable to a completely noninformative approach.

Conjugacy

Another consideration in the choice of prior is *conjugacy*. Consider using the beta distribution as a prior for the binomial setting as we have done previously. It turns out that using a $\beta(\mathcal{A}, \mathcal{B})$ results in the following posterior:

$$p(\theta|y, n) \propto \beta(y + \mathcal{A}, n - y + \mathcal{B})$$

Thus the posterior has the same parametric form as the prior, i.e. the beta distribution is *congugate* for the binomial likelihood. In this sense the prior has the interpretation as additional data points. In our regression model, using a normal distribution for the predictor coefficients and an inverse gamma for σ^2 is the conjugate setting. In the case of exponential family distributions of generalized linear models, there are natural conjugate prior distributions.

While there can be practical advantages to using a conjugate prior, it is not required, and for many more complex models, may not even be possible. However it might help to consider a known conjugate prior as a starting point if nothing else.

Sensitivity Analysis

As a reminder, we pointed out in the Sensitivity Analysis section of the discussion on model checking, one may perform checks on settings

for the model to see if changes to them results in gross changes of inference from the posterior. Part of that check should include the choice of prior, whether different versions of the same distribution or different distributions altogether. Doing such a check will give you more confidence in the final selection.

Summary

The BUGS book has many examples for a wide variety of applications. The Stan github page has Stan examples for each of those BUGS examples.

It will not take long with a couple Bayesian texts or research articles that employ Bayesian methods to get a feel for how to go about choosing priors. One should also remember that in the face of a lot of data, the likelihood will overwhelm the prior, rendering the choice effectively moot. While the choice might be 'subjective' in some respects, it is not arbitrary, and there are standard choices for common models and guidelines for more complex ones to help the researcher in their choice.

Sampling Procedure

There are many ways in which one might sample from the posterior. Bayesian analysis is highly flexible and can solve a great many statistical models in theory. In practice things can be more difficult. As more complex models are attempted, new approaches are undertaken to deal with the problems in estimation that inevitably arise. In an attempt to dissolve at least some of the mystery, a brief description follows.

Metropolis

We have mentioned that BUGS and JAGS use Gibbs sampling, which is a special case of the *Metropolis-Hastings* (MH) algorithm[28] a very general approach encompassing a wide variety of techniques. The Metropolis algorithm can be briefly described in the following steps:

[28] Originally developed in physics in the 50s, it slowly made its way across to other fields.

1. Start with initial values for the parameters θ^0

   For $t = 1, 2...N_{sim}$:

2. Sample from some proposal distribution a potential candidate θ^*, given θ^{t-1}
3. Calculate the ratio r of the posterior densities $\frac{p(\theta^*|y)}{p(\theta^{t-1}|y)}$[29]
4. Set $\theta^t = \theta^*$ with probability $min(r, 1)$, else $\theta^t = \theta^{t-1}$

[29] In practice we can take the difference in the log values.

Conceptually, if the proposal increases the posterior density, $\theta^t = \theta^*$. If it decreases the proposal density, set $\theta^t = \theta^*$ with probability r, else

it is θ^{t-1}. The MH algorithm generalizes the Metropolis to use asymmetric proposal distributions and uses an r to correct for asymmetry[30].

Let's look at this in generic R code for additional clarity:

```
nsim = numberSimulatedDraws
theta0 = initValue
theta = c(theta0, rep(NA, nsim))

for (t in 2:nsim){
  thetaStar = rnorm(1, theta[-1], sd)
  u = runif(1)
  r = exp(logPosterior_thetaStar - logPosterior_theta0)
  theta[t] = ifelse(u<=r, thetaStar, theta[-1])
}
```

One can see the Metropolis Hastings Example to see the Metropolis algorithm applied to our regression problem.

Gibbs

The *Gibbs* sampler takes an alternating approach for multiparameter problems, sampling one parameter given the values of the others, and thus reducing a potentially high dimensional problem to lower dimensional conditional densities. We can describe its steps generally as follows.

Start with initial values for some ordering of the parameters $\theta_1^0, \theta_2^0, ..., \theta_p^0$

For $t = 1, 2..., N_{sim}$:

At iteration t, for $p = 1, 2..., P$:

1. Generate $\theta_1^t \sim p(\theta_1^t | \theta_2^{t-1}, \theta_3^{t-1}, ..., \theta_p^{t-1})$
2. Generate $\theta_2^t \sim p(\theta_2^t | \theta_1^t, \theta_3^{t-1}, ..., \theta_p^{t-1})$

$\vdots$

p. Generate $\theta_p^t \sim p(\theta_p^t | \theta_1^t, \theta_2^t, ..., \theta_{p-1}^t)$

Again some generic code may clarify:

```
for (t in 1:nsim){
  for (p in 1:P){
    thetaNew[p] = rDistribution(1, theta[t,-p])
  }
  theta[t,] = thetaNew
}
```

[30] Given a proposal/jumping distribution $\mathcal{J}_t$,
$r = \frac{p(\theta^*|y)/\mathcal{J}_t(\theta^*|\theta^{t-1})}{p(\theta^{t-1}|y)/\mathcal{J}_t(\theta^{t-1}|\theta^*)}$

Hamiltonian Monte Carlo

Stan uses *Hamiltonian Monte Carlo*, another variant of MH. It takes the parameters θ as collectively denoting the position of a particle in some space with momentum ϕ (of same dimension as θ). Both θ and ϕ are updated at each Metropolis step and jointly estimated, though we are only interested in θ. We can describe the basic steps as follows.

1. At iteration t, take a random draw of momentum ϕ from its posterior distribution
2. Update the position vector θ given current momentum, update ϕ given the gradient of θ
3. Calculate $r = \frac{p(\theta^*|y)p(\phi^*)}{p(\theta^{t-1})p(\phi^{t-1})}$
4. Set $\theta^t = \theta^*$ with probability $min(r, 1)$, else $\theta^t = \theta^{t-1}$

The overall process allows it to move quite rapidly through the parameter space, and can work well where other approaches such as Gibbs might be very slow. An example using HMC on the regression model data can be found in the Hamiltonian Monte Carlo Example.

Other Variations and Approximate Methods

Within these MH approaches there are variations such as slice sampling, reversible jump, particle filtering, etc. Also, one can reparameterize the model to help overcome some convergence issues if applicable. In addition, there exist many approximate methods such as Variational Bayes, INLA, Approximate Bayesian Computation, etc. The main thing is just to be familiar with what's out there in case it might be useful. Any particular method might be particularly well suited to certain models (e.g. INLA for spatial regression models), those that are notably complex, or they may just be convenient for a particular case.

Number of draws, thinning, warm-up

Whatever program we use, the typical inputs that will need to be set regard the number of simulated draws from the posterior, the number of warm-up draws, and the amount of thinning. Only the draws that remain after warm-up and thinning will be used for inference. However, there certainly is no default that would work from one situation to the next, and there is no limit.

Recall that we are looking for convergence to a distribution, and this isn't determined by the number of draws alone. The fact is that one only needs a few draws for accurate inference. Even something as low

as n_{eff} of 10 for each chain would actually be ok assuming everything else seemed in order, though typically we want more than that so that our values don't bounce around from one model run to the next. To feel confident about convergence, i.e. get $\hat{R}$ of around 1, plots looking right, etc., we will usually want in the thousands for the number of total draws. We might need quite a few more for increasing model complexity.

A conservative approach to the number of warm-up draws is half the number of runs, but this is fairly arbitrary. Thinning isn't specifically necessary for inference if approximate convergence is achieved, but is useful with increasing model complexity to reduce autocorrelation among the estimates.

For myself, I typically run models such that the results are based on roughly $n_{\text{eff}} = 1000$ estimates per chain, simply because 1000 is a nice round number and is enough to make graphical display nice. For a regression model as we have been running, that could be setting the number of simulations at 12000, the warm-up at 2000, and thinning at 10. Other models might make due with 100000, 50000, 50 respectively. Just do what you feel you need to.

Model Complexity

One of the great things about the Bayesian approach is its ability to handle extremely complex models with oodles of parameters. In addition, it will often work better (or at all) in simpler settings where the data under consideration are problematic (e.g. collinearity, separation in the logistic regression setting). While it can be quite an undertaking to set things correctly and debug, re-run etc. and generally go through the trial and error process typically associated with highly complex models, it's definitely nice to know that you can. It will take some work, but you will learn a great deal along the way. Furthermore, there are typically tips and tricks that can potentially help just about any model run a little more smoothly.

Summary

Hopefully this document has provided a path toward easing into Bayesian analysis for those that are interested but might not have had the confidence or particular skill set that many texts and courses assume. Conceptually, Bayesian inference can be fairly straightforward, and inferentially is more akin to the ways people naturally think about probability. Many of the steps taken in classical statistical analysis are

still present, but have been enriched via the incorporation of prior information, a more flexible modeling scheme, and the ability to enrich even standard analyses with new means of investigation.

Of course it will not necessarily be easy, particularly for complex models, though such models might actually be relatively easier in the classical framework. While not necessary for all models, oftentimes the process will involve a more hands-on approach. However this allows for more understanding of the model and its results, and gets easier with practice just like anything else.

You certainly don't have to abandon classical and other methods either. Scientific research involves applying the best tool for the job, and in some cases the Bayesian approach may not be the best fit for a particular problem. But when it is, it's hoped you'll be willing to take the plunge, and know there are many tools and a great community of folks to help you along the way.

Best of luck with your research!

Appendix

Maximum Likelihood Review

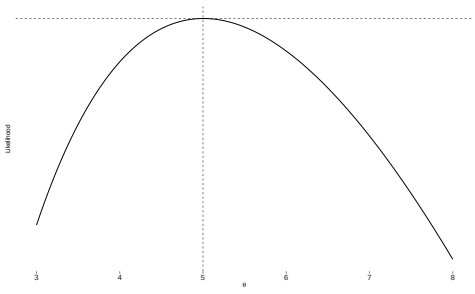
This is a very brief refresher on maximum likelihood estimation using a standard regression approach as an example, and more or less assumes one hasn't tried to roll their own such function in a programming environment before. Given the likelihood's role in Bayesian estimation and statistics in general, and the ties between specific Bayesian results and maximum likelihood estimates one typically comes across, I figure one should be comfortable with some basic likelihood estimation.

In the standard model setting we attempt to find parameters θ that will maximize the probability of the data we actually observe[31]. We'll start with an observed random response vector y with $i...N$ independent and identically distributed observations and some data-generating process underlying it $f(\cdot|\theta)$. We are interested in estimating the model parameter(s), θ, that would make the data most likely to have occurred. The probability density function for y given some particular estimate for the parameters can be noted as $f(y_i|\theta)$. The joint probability distribution of the (independent) observations given those parameters, $f(y_i|\theta)$, is the product of the individual densities, and is our *likelihood function*. We can write it out generally as:

[31] The principle of maximum likelihood.

$$\mathcal{L}(\theta) = \prod_{i=1}^{N} f(y_i|\theta)$$

Thus the *likelihood* for one set of parameter estimates given a fixed set of data y, is equal to the probability of the data given those (fixed) estimates. Furthermore we can compare one set, $\mathcal{L}(\theta_A)$, to that of another, $\mathcal{L}(\theta_B)$, and whichever produces the greater likelihood would be the preferred set of estimates. We can get a sense of this with the graph to the right, based on a single parameter, Poisson distributed variable. The data is drawn from a variable with mean $\theta = 5$. We note the calculated likelihood increases as we estimate values for θ closer to 5.



For computational reasons we instead work with the sum of the natural log probabilities[32], and thus the log likelihood:

$$\ln \mathcal{L}(\theta) = \sum_{i=1}^{N} \ln[f(y_i|\theta)]$$

[32] Math refresher on logs: log(A*B) = log(A)+log(B). So summing the log probabilities will result in the same values for θ, but won't result in extremely small values that will break our computer.

Concretely, we calculate a log likelihood for each observation and then sum them for the total likelihood for parameter(s) θ.

The likelihood function incorporates our assumption about the sampling distribution of the data given some estimate for the parameters.

It can take on many forms and be notably complex depending on the model in question, but once specified we can use any number of optimization approaches to find the estimates of the parameter that make the data most likely. As an example, for a normally distributed variable of interest we can write the log likelihood as follows:

$$\ln \mathcal{L}(\theta) = \sum_{i=1}^{N} \ln[\frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(y-\mu)^2}{2\sigma^2})]$$

Example

In the following we will demonstrate the maximum likelihood approach to estimation for a simple setting incorporating a normal distribution where we estimate the mean and variance/sd for a set of responses[33]. First the data is created, and then we create the function that will compute the log likelihood. Using the built in R distributions[34] makes it fairly straightforward to create our own likelihood function and feed it into an optimization function to find the best parameters. We will set things up to work with the bbmle package, which has some nice summary functionality and other features. However, one should take a glance at `optim` and the other underlying functions that do the work.

[33] Of course we could just use the sample estimates, but this is for demonstration.

[34] `Type ?Distributions` at the console for some of the basic R distributions available.

```
# for replication
set.seed(1234)

# create the data
y = rnorm(1000, mean=5, sd=2)
startvals = c(0, 1)

# the log likelihood function
LL = function(mu=startvals[1], sigma=startvals[2]){
  ll = -sum(dnorm(y, mean=mu, sd=sigma, log=T))
  est = c(mu, sigma, ll)
  message(paste(est[1], est[2], est[3]))
  ll
}
```

The `LL` function takes starting points for the parameters as arguments, in this case we call them μ and σ, which will be set to 0 and 1 respectively. Only the first line (ll = -sum...) is actually necessary, and we use `dnorm` to get the density for each point[35]. Since this optimizer is by default minimization, we reverse the sign of the sum so as to minimize the negative log likelihood, which is the same as maximizing the likelihood. Note that the bit of other code just allows you to see the estimates as the optimization procedure searches for the best values. I do not show that here but you'll see it in your console.

[35] Much more straightforward than writing the likelihood function as above.

We are now ready to obtain maximum likelihood estimates for the parameters. For the `mle2` function we will need the function we've

created, plus other inputs related to that function or the underlying optimizing function used (by default `optim`). In this case we will use an optimization procedure that will allow us to set a lower bound for σ. This isn't strictly necessary, but otherwise you would get get warnings and possibly lack of convergence if negative estimates for σ were allowed[36].

[36] An alternative approach would be to work with the log of σ which can take on negative values, and then convert it back to the original scale.

```
library(bbmle)
# using optim, and L-BFGS-B so as to contrain sigma to be
# positive by setting the lower bound at zero
mlnorm =  mle2(LL, method="L-BFGS-B", lower=c(sigma=0))
mlnorm

##
## Call:
## mle2(minuslogl = LL, method = "L-BFGS-B", lower = c(sigma = 0))
##
## Coefficients:
##    mu sigma
## 4.947 1.994
##
## Log-likelihood: -2109

# compare to intercept only regression model
summary(lm(y~1))

##
## Call:
## lm(formula = y ~ 1)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -6.739 -1.293 -0.026  1.285  6.445
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   4.9468     0.0631    78.4   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.99 on 999 degrees of freedom
```

We can see that the ML estimates are the same[37] as the intercept only model estimates, which given the sample size are close to the true values.

[37] Actually there is a difference between the sigma estimates in that OLS estimates are based on a variance estimate divided by $N-1$ while the MLE estimate has a divisor of N.

In terms of the parameters we estimate, in the typical case of two or more parameters we can think of a *likelihood surface* that represents the possible likelihood values given any particular set of estimates. Given some starting point, the optimization procedure then travels along the surface looking for a minimum/maximum point[38]. For simpler settings such as this, we can visualize the likelihood surface and its minimum point. The optimizer travels along this surface until it finds a minimum. I also plot the the path of the optimizer from a top down view. The large blue dot noted represents the minimum negative log

[38] Which is equivalent to finding the point where the slope of the tangent line to some function, i.e. the derivative, to the surface is zero. The derivative, or gradient in the case of multiple parameters, of the likelihood function with respect to the parameters is known as the *score function*.

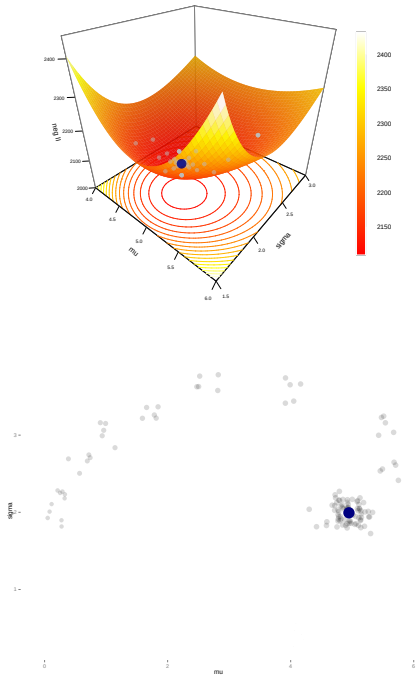
likelihood.

Please note that there are many other considerations in optimization completely ignored here, but for our purposes and the audience for which this is intended, we do not want to lose sight of the forest for the trees. We now move next to a slightly more complicated regression example.

Linear Model

In the regression context, our expected value for the response comes from our linear predictor, i.e. the weighted combination of our explanatory variables, and we estimate the regression weights/coefficients and possibly other relevant parameters. We can expand our previous example to the standard linear model without too much change. In this case we estimate a mean for each observation, but otherwise assume the variance is constant across observations. Again we first construct some data so that we know exactly what to expect, then write out the likelihood function with starting parameters. As we need to estimate our intercept and coefficient for the X predictor (collectively referred to as β), we can can think of our likelihood explicitly as before:

$$\ln \mathcal{L}(\beta, \sigma^2) = \sum_{i=1}^{N} \ln[\frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(y - X\beta)^2}{2\sigma^2})]$$



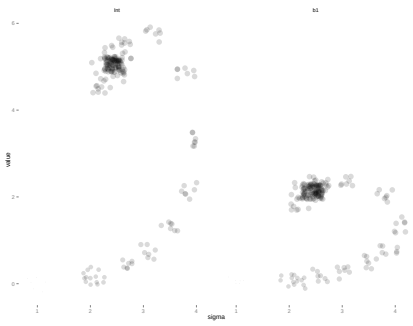
A bit of jitter was added to the points to better see what's going on.

```
# for replication
set.seed(1234)

# predictor
X = rnorm(1000)

# coefficients for intercept and predictor
theta = c(5,2)

# add intercept to X and create y with some noise
y = cbind(1,X)%*%theta + rnorm(1000, sd=2.5)

regLL = function(sigma=1, Int=0, b1=0){
  coefs = c(Int, b1)
  mu = cbind(1,X)%*%coefs

  ll = -sum(dnorm(y, mean=mu, sd=sigma, log=T))
  est = c(sigma, Int, b1, ll)
  message(paste(est[1], est[2], est[3], est[4]))
  ll
}

library(bbmle)
mlopt =  mle2(regLL, method="L-BFGS-B", lower=c(sigma=0))
summary(mlopt)

## Maximum likelihood estimation
```

```
##
## Call:
## mle2(minuslogl = regLL, method = "L-BFGS-B", lower = c(sigma = 0))
##
## Coefficients:
##       Estimate Std. Error z value  Pr(z)
## sigma   2.4478     0.0547    44.7 <2e-16 ***
## Int     5.0400     0.0774    65.1 <2e-16 ***
## b1      2.1393     0.0777    27.6 <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## -2 log L: 4628

# plot(profile(mlopt), absVal=F)

modlm = lm(y~X)
summary(modlm)

##
## Call:
## lm(formula = y ~ X)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -7.915 -1.610  0.036  1.634  7.671
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   5.0400     0.0775    65.0   <2e-16 ***
## X             2.1393     0.0777    27.5   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.45 on 998 degrees of freedom
## Multiple R-squared:  0.431,Adjusted R-squared:  0.431
## F-statistic:  757 on 1 and 998 DF,  p-value: <2e-16

-2*logLik(modlm)

## 'log Lik.' 4628 (df=3)
```

As before, our estimates and final log likelihood value are about where they should be, and reflect the lm output. The visualization becomes more difficult, but we can examine slices similar to the previous plot.

To move to generalized linear models, very little changes of the process outside of the distribution assumed and that we are typically modeling a function of the response (e.g. $log(y) = X\beta$).



Binomial Likelihood Example

This regards the example seen in the early part of the document with the hands-on example.

```
x1 = rbinom(1000, size=10, p=.5)
x2 = rbinom(1000, size=20, p=.85)

binomLL = function(theta, x) {
  -sum(dbinom(x, size=10, p=theta, log=T))
}

optimize(binomLL, x=x1, lower=0, upper=1); mean(x1)

## $minimum
## [1] 0.5043
##
## $objective
## [1] 1903
## [1] 5.043

optimize(binomLL, x=x2, lower=0, upper=1); mean(x2)

## $minimum
## [1] 0.9999
##
## $objective
## [1] Inf
## [1] 17.09
```

Modeling Languages

I will talk only briefly about the modeling language options available, as you will have to make your own choice among many.

Bugs

BUGS Lunn et al. (2012) (*B*ayesian inference *U*sing *G*ibbs *S*ampling) is perhaps the most widely known and used Bayesian modeling language, as it has been around for 25 years at this point. It is implemented via OpenBUGS and freely available for download[39]. It even has a GUI interface if such a thing is desired.

[39] You might come across a previous incarnation, WinBugs, but it is no longer being developed.

JAGS

JAGS (*J*ust *A*nother *G*ibbs *S*ampler) is a more recent dialect of the BUGS language, and is also free to download. It offers some technical and modeling advantages to OpenBUGs, but much of the code translates directly from one to the other.

Stan

Stan is a relative newcomer to Bayesian modeling languages, having only been out a couple years now. It uses a different estimation procedure than the BUGS language and this makes it more flexible and perhaps better behaved for many types of models. It actually compiles Stan code to C++, and so can be very fast as well. I personally prefer it as I find it more clear in its expression, but your mileage may vary.

R

R has many modeling packages devoted to Bayesian analysis such that there is a Task View specific to the topic. Most of them are even specific to the implementation of a certain type of analysis[40]. What's more, R has interfaces to the previous language engines via the packages R2OpenBUGS and BRugs, rjags, and rstan. So not only can you do everything within R and take advantage of the power of those languages, you can then use Bayesian specific R packages on the results.

[40] Many of these packages, if not all of them will be less flexible in model specification compared to implementing languages the aforementioned languages directly or using the R interface to those languages.

General Statistical Package Implementations

The general statistical languages such as SAS, SPSS, and Stata were very late to the Bayesian game, even for implementations of Bayesian versions of commonly used models. SAS started a few years ago (roughly 2006) with experimental and extremely limited capability, and Stata only very recently. SPSS doesn't seem to have much if any

capability (and no, Amos doesn't count). Others still seem to be lacking as well. In general, I wouldn't recommend these packages except as an interface to one of the Bayesian specific languages, assuming they have the capability (e.g. Stata can do this).

Other Programming Languages

Python has functionality via such modules as PyMC, and Stan has a Python implementation, PyStan. Julia has already has some functionality similar in implementation to Matlab's, which one may also consider. And with any programming language that you might use for statistical analysis you could certainly do a lot of it by hand if you have the time.

Summary

In short, you have plenty of options, and the list above barely scratches the surface. I would suggest starting with a Bayesian programming language or using that language within your chosen statistical environment or package. This gives you the most modeling flexibility, choice, and opportunity to learn.

BUGS Example

The following provides a BUGS example of the primary model used in the document. The applicable code for the data set up is in the Linear Regression Model Example section of the document. The model matrix X must be a matrix class object. Next we setup a bugs data list as we did with Stan, and create a text file that contains the model code. Note that the data list comprises simple characters which are used to look for objects of those names that are in the environment. Also, I use `cat` with `sink` so that I don't have to go to a separate text editor to create the file.

One of the big differences between BUGS and other languages is its use of the precision parameter $\frac{1}{\sigma^2}$, the inverse variance, usually denoted as τ. While there were some computational niceties to be had in doing so, even the authors admit this was not a good decision in retrospect. But it's too late to change it now, so prepare to have that come up from time to time when you inevitably forget. Comments and assignments are the same as R, and distributions noted with $\sim$.

```
##################
### BUGS setup ###
##################

bugsdat = list('y', 'X', 'N', 'K')

# This will create a file, lmbugs.txt that will subsequently be called
sink('data/lmbugs.txt')
cat(
'model {
  for (n in 1:N){
    mu[n] <- beta[1]*X[n,1] + beta[2]*X[n,2] + beta[3]*X[n,3] + beta[4]*X[n,4]
    y[n] ~ dnorm(mu[n], inv.sigma.sq)
  }
  for (k in 1:K){
    beta[k] ~ dnorm(0, .001) # prior for reg coefs
  }
  # Half-cauchy as in Gelman 2006
  # Scale parameter is 5, so precision of z = 1/5^2 = 0.04
  sigma.y <- abs(z)/sqrt(chSq) # prior for sigma; cauchy = normal/sqrt(chi^2)
  z ~ dnorm(0, .04)I(0,)
  chSq ~ dgamma(0.5, 0.5) # chi^2 with 1 d.f.
  inv.sigma.sq <- pow(sigma.y, -2) # precision
  # sigma.y ~ dgamma(.001,.001) # prior for sigma; a typical approach used.
}'
)
sink()


# explicitly provided initial values not necessary, but one can specify them as
# follows, and you may have problems with variance parameters if you don't; Note
# also that sigma.y is unnecesary if using the half-cauchy approach as it is
# defined based on other values.

# inits <- list(list(beta=rep(0,4), sigma.y=runif(1,0,10)),
#               list(beta=rep(0,4), sigma.y=runif(1,0,10)),
```

```
#               list(beta=rep(0,4), sigma.y=runif(1,0,10)))
# parameters <- c('beta', 'sigma.y')
```

Now we are ready to run the model. You'll want to examine the help file for the bugs function for more information. In addition, depending on your setup you may need to set the working directory and other options. Note that `n.thin` argument is used differently than other packages. One specifies the n posterior draws (per chain) you to keep want as n.iter-n.burnin. The thinned samples aren't stored. Compare this to other packages where n.iter is the total before thinning and including burn-in, and n.keep is (n.iter-n.burnin)/n.thin. With the function used here, n.keep is the same, but as far as arguments your you'll want to think of n.iter as the number of posterior draws *after* thinning. So the following all produce 1000 posterior draws in R2OpenBUGS:

n.iter=3000	n.thin=1	n.burnin=2000
n.iter=3000	n.thin=10	n.burnin=2000
n.iter=3000	n.thin=100	n.burnin=2000

In other packages, with those arguments you'd end up with 1000, 300, 30 posterior draws.

```
######################
### Run the model ###
######################
library(R2OpenBUGS)
lmbugs <- bugs(bugsdat, inits=NULL, parameters=c('beta', 'sigma.y'),
               model.file='lmbugs.txt', n.chains=3, n.iter=3000, n.thin=10,
               n.burnin=2000)
# save.image('data/lmbugsResults.RData')
```

Now we are ready for the results, which will be the same as what we saw with Stan. In addition to the usual output, you get the *deviance information criterion* as a potential means for model comparison. This would need to be explicitly calculated either via the 'generated quantities' block in the Stan model code, or after import of the results into R.

```
## print(lmbugs, digits=3)

##              mean    sd     2.5%      50%    97.5%  Rhat n.eff
## beta[1]     4.900 0.127    4.648    4.903    5.143 1.001  2400
## beta[2]     0.084 0.130   -0.166    0.084    0.336 1.001  3000
## beta[3]    -1.468 0.125   -1.721   -1.470   -1.224 1.001  2100
## beta[4]     0.824 0.121    0.587    0.827    1.053 1.001  3000
## sigma.y     2.028 0.092    1.860    2.024    2.218 1.001  3000
## deviance 1063.611 3.148 1059.000 1063.000 1071.000 1.001  3000
```

The usual model diagnostics are available with conversion of the results to an object the coda package can work with. Figures are not shown, but they are the typical traceplots and density plots.

```
lmbugscoda = as.mcmc.list(lmbugs)
traceplot(lmbugscoda)
densityplot(lmbugscoda)
plot(lmbugscoda)
# par(mar=c(5, 4, 4, 2) + 0.1) # reset margins
```

JAGS Example

The following shows how to run the regression model earlier in the document via Jags. Once you have the data set up as before, the data list is done in the same fashion as with BUGS. The code itself is mostly identical, save for the use of T instead of I for truncation. JAGS, being a BUGS dialect, also uses the precision parameter in lieu of the variance.

```
jagsdat = list('y'=y, 'X'=X, 'N'=N, 'K'=K)

sink('data/lmjags.txt')
cat(
'model {
  for (n in 1:N){
    mu[n] <- beta[1]*X[n,1] + beta[2]*X[n,2] + beta[3]*X[n,3] + beta[4]*X[n,4]
    y[n] ~ dnorm(mu[n], inv.sigma.sq)
  }

  for (k in 1:K){
    beta[k] ~ dnorm(0, .001)
  }

  # Half-cauchy as in Gelman 2006
  # Scale parameter is 5, so precision of z = 1/5^2 = 0.04
  sigma.y <- z/sqrt(chSq)
  z ~ dnorm(0, .04)T(0,)
  chSq ~ dgamma(0.5, 0.5)
  inv.sigma.sq <- pow(sigma.y, -2)
}'
)
sink()


# explicitly provided initial values not necessary, but can specify as follows
# inits <- function(){
#   list(beta=rep(0,4), sigma.y=runif(1,0,10) )
# }
parameters <- c('beta', 'sigma.y')
```

With everything set, we can now run the model. With JAGS, we have what might be called an initialization stage that sets the model up and runs through the warm-up period, after which we can then flexibly sample from the posterior via the `coda.samples` function.

```
library(rjags); library(coda)
lmjagsmod <- jags.model(file='data/lmjags.txt', data=jagsdat, # inits=inits
# update(lmjagsmod, 10000) # alternative approache
lmjags = coda.samples(lmjagsmod, c('beta', 'sigma.y'), n.iter=10000,
                      thin=10, n.chains=3)
```

Now we have a model identical to the others, and can summarize the posterior distribution in similar fashion.

```
summary(lmjags)

##
## Iterations = 2010:12000
## Thinning interval = 10
## Number of chains = 3
## Sample size per chain = 1000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##            Mean    SD Naive SE Time-series SE
## beta[1]  4.8950 0.128  0.00234        0.00234
## beta[2]  0.0812 0.131  0.00239        0.00226
## beta[3] -1.4693 0.125  0.00229        0.00229
## beta[4]  0.8147 0.123  0.00225        0.00225
## sigma.y  2.0280 0.094  0.00172        0.00172
##
## 2. Quantiles for each variable:
##
##           2.5%      25%     50%    75%  97.5%
## beta[1]  4.644  4.81091  4.8933  4.983  5.149
## beta[2] -0.174 -0.00821  0.0815  0.170  0.336
## beta[3] -1.713 -1.55325 -1.4697 -1.385 -1.220
## beta[4]  0.575  0.73297  0.8173  0.896  1.055
## sigma.y  1.857  1.96250  2.0237  2.089  2.219

effectiveSize(lmjags)

## beta[1] beta[2] beta[3] beta[4] sigma.y
##    3000    3453    3000    3000    3000
```

```
# visualize
library(coda); library(scales); library(ggthemes)
traceplot(lmjags, col=alpha(gg_color_hue(3), .5))
densityplot(lmjags, col=alpha(gg_color_hue(3), .5))
plot(lmjags, col=alpha(gg_color_hue(3), .25))
corrplot:::corrplot(cor(lmjags[[2]]))  # noticeably better than levelplot
par(mar=c(5, 4, 4, 2) + 0.1) # reset margins
```

Metropolis Hastings Example

Next depicted is a random walk Metropolis-Hastings algorithm using the the data and model from prior sections of the document. I had several texts open while cobbling together this code such as Gelman et al. (2013), and some oriented towards teh social sciences by Gill (2008), Jackman (2009), and Lynch (2007) etc. Some parts of the code reflect information and code examples found therein, and follows Lynch's code a bit more.

The primary functions that we need to specify regard the posterior distribution[41], an update step for beta coefficients, and an update step for the variance estimate.

[41] Assuming normal for β coefficients, inverse gamma on σ^2.

```
### posterior function
post = function(x, y, b, s2){
  # Args: X is the model matrix; y the response vector; b and s2 the parameters
  # to be esitmated

  beta = b
  sigma = sqrt(s2)
  sigma2 = s2
  mu = X %*% beta

  # priors are b0 ~ N(0, sd=10), sigma2 ~ invGamma(.001, .001)
  priorbvarinv = diag(1/100,4)
  prioralpha = priorbeta = .001

  if(is.nan(sigma) | sigma<=0){     # scale parameter must be positive
    return(-Inf)
  }
  # Note that you will not find the exact same presentation across texts and
  # other media for the log posterior in this conjugate setting.  In the end
  # they are conceputally still (log) prior + (log) likelihood
  else {
    -(nrow(X)/2)*log(sigma2) - (1/(2*sigma2) *(crossprod(y-mu))) +
      -(ncol(X)/2)*log(sigma2) - (1/(2*sigma2) * (t(beta)%*%priorbvarinv%*%beta)) +
      -(prioralpha+1)*log(sigma2) + log(sigma2) - priorbeta/sigma2
  }
}

### update step for regression coefficients
updatereg = function(i, x, y, b, s2){
  # Args are the same as above but with additional i iterator argument.
  require(MASS)
  b[i,] = mvrnorm(1, mu=b[i-1,], Sigma=bvarscale)  # proposal/jumping distribution

  # Compare to past- does it increase the posterior probability?
  postdiff = post(x=x, y=y, b=b[i,],   s2=s2[i-1]) -
             post(x=x, y=y, b=b[i-1,], s2=s2[i-1])

  # Acceptance phase
  unidraw = runif(1)
  accept = unidraw < min(exp(postdiff), 1)  # accept if so
  if(accept) {b[i,]}
  else {b[i-1,]}
}
```

```
# update step for sigma2
updates2 = function(i, x, y, b, s2){
  s2candidate =  rnorm(1, s2[i-1], sd=sigmascale)
  if(s2candidate < 0) {
    accept = FALSE
  }
  else {
    s2diff = post(x=x, y=y, b=b[i,], s2=s2candidate) -
             post(x=x, y=y, b=b[i,], s2=s2[i-1])
    unidraw = runif(1)
    accept = unidraw < min(exp(s2diff), 1)
  }

  ifelse(accept, s2candidate, s2[i-1])
}
```

Now we can set things up for the MCMC chain[42]. Aside from the typical MCMC setup and initializing the parameter matrices to hold the draws from the posterior, we also require scale parameters to use for the jumping/proposal distribution.

[42] This code regards only one chain, though a simple loop or any number of other approaches would easily extend it to two or more.

```
### Setup, starting values etc. ###
nsim = 12000
burnin = 2000
thin = 10

b = matrix(0, nsim, ncol(X))         # initialize beta update matrix
s2 = rep(1, nsim)                    # initialize sigma vector


# For the following this c term comes from BDA3 12.2 and will produce an
# acceptance rate of .44 in 1 dimension and declining from there to about
# .23 in high dimensions. For the sigmascale, the magic number comes from
# starting with a value of one and fiddling from there to get around .44.
c = 2.4/sqrt(ncol(b))
bvar = vcov(lm(y~., data.frame(X[,-1])))
bvarscale = bvar * c^2
sigmascale = .9
```

We can now run and summarize the model with tools from the coda package.

```
### Run ###
for(i in 2:nsim){
  b[i,] = updatereg(i=i, y=y, x=X, b=b, s2=s2)
  s2[i] = updates2(i=i, y=y, x=X, b=b, s2=s2)
}

# calculate acceptance rates;
baccrate = mean(diff(b[(burnin+1):nsim,]) != 0)
s2accrate = mean(diff(s2[(burnin+1):nsim]) != 0)
baccrate

## [1] 0.2970297

s2accrate

## [1] 0.4288429
```

```
# get final chain
library(coda)
bfinal = as.mcmc(b[seq(burnin+1, nsim, by=thin),])
s2final = as.mcmc(s2[seq(burnin+1, nsim, by=thin)])

# get summaries; compare to lm and stan
summary(bfinal); summary(s2final)

##
## Iterations = 1:1000
## Thinning interval = 1
## Number of chains = 1
## Sample size per chain = 1000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##          Mean     SD Naive SE Time-series SE
## [1,]  4.89475 0.1252 0.003958       0.005070
## [2,]  0.08252 0.1299 0.004109       0.004922
## [3,] -1.46055 0.1202 0.003801       0.004599
## [4,]  0.82669 0.1221 0.003861       0.004666
##
## 2. Quantiles for each variable:
##
##         2.5%       25%     50%     75%   97.5%
## var1  4.6543  4.809621  4.8940  4.9764  5.1497
## var2 -0.1662 -0.007982  0.0776  0.1723  0.3337
## var3 -1.6851 -1.545646 -1.4612 -1.3806 -1.2203
## var4  0.6107  0.741037  0.8217  0.9085  1.0756
##
## Iterations = 1:1000
## Thinning interval = 1
## Number of chains = 1
## Sample size per chain = 1000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##           Mean             SD       Naive SE Time-series SE
##        4.07995        0.37404         0.01183        0.01183
##
## 2. Quantiles for each variable:
##
##  2.5%   25%   50%   75% 97.5%
## 3.382 3.827 4.060 4.308 4.904

round(c(coef(modlm), summary(modlm)$sigma^2), 3)

## (Intercept)          X1          X2          X3
##       4.898       0.084      -1.469       0.820       4.084
```

Here is the previous Stan fit for comparison.

```
print(fit, digits=3, prob=c(.025,.5,.975))

## Inference for Stan model: stanmodelcode.
## 3 chains, each with iter=12000; warmup=2000; thin=10;
## post-warmup draws per chain=1000, total post-warmup draws=3000.
##
##             mean se_mean    sd     2.5%      50%    97.5% n_eff  Rhat
```

```
## beta[1]    4.894   0.002 0.132    4.630    4.896    5.144  3000 1.001
## beta[2]    0.085   0.002 0.131   -0.178    0.086    0.340  3000 1.001
## beta[3]   -1.471   0.002 0.127   -1.716   -1.471   -1.221  2795 1.000
## beta[4]    0.819   0.002 0.121    0.576    0.820    1.057  3000 0.999
## sigma      2.032   0.002 0.091    1.862    2.029    2.215  2997 0.999
## lp__    -301.008   0.029 1.579 -304.855 -300.700 -298.853  3000 1.000
##
## Samples were drawn using NUTS(diag_e) at Sun May 18 14:01:52 2014.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

Hamiltonian Monte Carlo Example

The following demonstrates Hamiltonian Monte Carlo, the technique that Stan uses, and which is different than BUGS/JAGS. It still assumes the data we used in this document, and is largely based on the code in the appendix of Gelman et al. (2013).

First we start with the functions.

```
### Log posterior
log_p_th = function(X, y, th){
  # Args: X is the model matrix; y the response vector; th is the current
  # parameter estimates.
  beta = th[-length(th)]            # reg coefs to be estimated
  sigma = th[length(th)]            # sigma to be estimated
  sigma2 = sigma^2
  mu = X %*% beta

  # priors are b0 ~ N(0, sd=10), sigma2 ~ invGamma(.001, .001)
  priorbvarinv = diag(1/100, 4)
  prioralpha = priorbeta = .001

  if(is.nan(sigma) | sigma<=0){     # scale parameter must be positive, so post
    return(-Inf)                    # density is zero if it jumps below zero
  }
  # Note that you will not find the exact same presentation across texts and
  # other media for the log posterior in this conjugate setting.  In the end
  # they are conceputally still (log) prior + (log) likelihood
  else {
    -(nrow(X)/2)*log(sigma2) - (1/(2*sigma2) *(crossprod(y-mu))) +
      -(ncol(X)/2)*log(sigma2) - (1/(2*sigma2) *(t(beta)%*%priorbvarinv%*%beta)) +
      -(prioralpha+1)*log(sigma2) + log(sigma2) - priorbeta/sigma2
  }
}


### numerical gradient as given in BDA3 p. 602; same args as posterior
gradient_th_numerical = function(X, y, th){
  d = length(th)
  e = .0001
  diffs = numeric(5)
  for(k in 1:d){
    th_hi = th
    th_lo = th
    th_hi[k] = th[k] + e
    th_lo[k] = th[k] - e
    diffs[k] = (log_p_th(X, y, th_hi) - log_p_th(X, y, th_lo)) / (2*e)
  }
  return(diffs)
}


### single HMC iteration
hmc_iteration = function(X, y, th, epsilon, L, M){
  # Args: epsilon is the stepsize; L is the number of leapfrog steps; epsilon
  # and L are drawn randomly at each iteration to explore other areas of the
  # posterior (starting with epsilon0 and L0); M is a diagonal mass matrix
  # (expressed as a vector), a bit of a magic number in this setting. It regards
  # the mass of a particle whose position is represented by theta, and momentum
  # by phi. See the sampling section of chapter 1 in the Stan manual for more
```

```
  # detail.

  M_inv = 1/M
  d = length(th)
  phi = rnorm(d, 0, sqrt(M))
  th_old = th
  log_p_old = log_p_th(X, y, th) - .5*sum(M_inv*phi^2)
  phi = phi + .5*epsilon*gradient_th_numerical(X, y, th)

  for (l in 1:L){
    th = th + epsilon*M_inv*phi
    phi = phi + ifelse(l==L, .5, 1)*epsilon*gradient_th_numerical(X, y, th)
  }

  # here we get into standard MCMC stuff, jump or not based on a draw from a
  # proposal distribution
  phi = -phi
  log_p_star = log_p_th(X, y, th) - .5*sum(M_inv*phi^2)
  r = exp(log_p_star - log_p_old)
  if (is.nan(r)) r = 0
  p_jump = min(r, 1)
  th_new = if(runif(1) < p_jump) th else th_old
  return(list(th=th_new, p_jump=p_jump))  # returns estimates and acceptance rate
}


### main HMC function
hmc_run = function(starts, iter, warmup, epsilon_0, L_0, M, X, y){
  # Args: starts are starting values; iter is total number of simulations for
  # each chain (note chain is based on the dimension of starts); warmup
  # determines which of the initial iterations will be ignored for inference
  # purposes; edepsilon0 is the baseline stepsize; L0 is the baseline number
  # of leapfrog steps; M is the mass vector
  chains = nrow(starts)
  d = ncol(starts)
  sims = array(NA, c(iter, chains, d),
               dimnames=list(NULL, NULL, colnames(starts)))
  p_jump = matrix(NA, iter, chains)

  for(j in 1:chains){
    th = starts[j,]
    for(t in 1:iter){
      epsilon = runif(1, 0, 2*epsilon_0)
      L = ceiling(2*L_0*runif(1))
      temp = hmc_iteration(X, y, th, epsilon, L, M)
      p_jump[t,j] = temp$p_jump
      sims[t,j,] = temp$th
      th = temp$th
    }
  }

  rstan::monitor(sims, warmup, digits_summary=3)
  acc = round(colMeans(p_jump[(warmup+1):iter,],), 3)  # acceptance rate
  message('Avg acceptance probability for each chain: ',
          paste0(acc[1],', ',acc[2]), '\n')
  return(list(sims=sims, p_jump=p_jump))
}
```

With the primary functions in place, we set the starting values and choose other settings for for the HMC process. The coefficient starting

values are based on random draws from a uniform distribution, while σ is set to a value of one in each case. As in the other examples we'll have 12000 total draws with warm-up set to 2000. I don't have any thinning option but that could be added or simply done as part of the coda package preparation.

```
### Starting values and mcmc settings
parnames = c(paste0('beta[',1:4,']'), 'sigma')
d = length(parnames)
chains = 2

thetastart = t(replicate(chains, c(runif(d-1, -1, 1), 1)))
colnames(thetastart) = parnames
nsim = 12000
wu = 2000

# fiddle with these to get a desirable acceptance rate of around .80. The
# following work well with the document data.
stepsize = .08
nLeap = 10
vars = rep(1, 5)
mass_vector = 1/vars
```

We are now ready to run the model. On my machine and with the above settings, it took about two minutes. Once complete we can use the coda package if desired as we have done before.

```
### Run the model
M1 = hmc_run(starts=thetastart, iter=nsim, warmup=wu, epsilon_0=stepsize,
             L_0=nLeap, M=mass_vector, X=X, y=y)

## Inference for the input samples (2 chains: each with iter=12000; warmup=2000):
##
##           mean se_mean    sd   2.5%    25%    50%    75%  97.5% n_eff Rhat
## beta[1]  4.900   0.001 0.129  4.648  4.812  4.900  4.987  5.154 12982    1
## beta[2]  0.085   0.001 0.130 -0.167 -0.003  0.084  0.172  0.343 12493    1
## beta[3] -1.468   0.001 0.126 -1.718 -1.550 -1.469 -1.384 -1.223 12577    1
## beta[4]  0.820   0.001 0.121  0.585  0.739  0.821  0.902  1.053 12958    1
## sigma    2.017   0.001 0.093  1.848  1.953  2.013  2.077  2.210 11460    1
##
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).

## Avg acceptance probability for each chain:  0.816, 0.822

# str(M1, 1)

# use coda if desired
library(coda)

theta = as.mcmc.list(list(as.mcmc(M1$sims[(wu+1):nsim,1,]),
                          as.mcmc(M1$sims[(wu+1):nsim,2,])))
# summary(theta)
finalest =  summary(theta)$statistics[,'Mean']
b = finalest[1:4]
sig = finalest[5]
log_p_th(X, y, finalest)

##           [,1]
## [1,] -301.7267
```

Our estimates look pretty good, and inspection of the diagnostics would show good mixing and convergence as well. At this point we can compare it to the Stan output. For the following, I modified the previous Stan code to use the same inverse gamma prior and tweaked the control options for a little bit more similarity, but that's not necessary.

```
## Inference for Stan model: stanmodelcodeIG.
## 2 chains, each with iter=12000; warmup=2000; thin=1;
## post-warmup draws per chain=10000, total post-warmup draws=20000.
##
##             mean se_mean    sd     2.5%      50%    97.5% n_eff Rhat
## beta[1]    4.894   0.001 0.128    4.641    4.895    5.144 15354    1
## beta[2]    0.083   0.001 0.130   -0.172    0.083    0.339 14592    1
## beta[3]   -1.469   0.001 0.127   -1.717   -1.469   -1.219 13756    1
## beta[4]    0.819   0.001 0.121    0.583    0.819    1.055 15600    1
## sigma      2.027   0.001 0.092    1.856    2.023    2.219 13883    1
## lp__    -301.532   0.018 1.584 -305.432 -301.213 -299.425  8079    1
##
## Samples were drawn using NUTS(diag_e) at Sun Jun 01 21:53:10 2014.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

Aside from Gelman's BDA3 which served as the primary source for the document content, the following list includes others that were either used as additional references, for code examples, etc., or those that might be useful or interesting to the audience for which this document is intended. Not even a remote attempt is made at a list of essential Bayesian references.

References

Albert, J. (2009). *Bayesian Computation with R*. Springer, New York, second edition.

Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A., and Rubin, D. B. (2013). *Bayesian Data Analysis*. CRC Press, third edition.

Gelman, A. and Hill, J. (2006). *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Cambridge University Press.

Gill, J. (2008). *Bayesian methods : a social and behavioral sciences approach*. Chapman & Hall/CRC, Boca Raton, second edition.

Jackman, S. (2009). *Bayesian analysis for the social sciences*. Wiley, Chichester, UK.

Kruschke, J. (2010). *Doing Bayesian Data Analysis: A Tutorial Introduction with R*. Academic Press.

Lunn, D., Jackson, C., Best, N., Thomas, A., and Spiegelhalter, D. (2012). *The BUGS Book: A Practical Introduction to Bayesian Analysis*. Chapman and Hall/CRC, Boca Raton, FL.

Lynch, S. M. (2007). *Introduction to applied Bayesian statistics and estimation for social scientists*. Springer, New York.

McGrayne, S. B. (2012). *The Theory That Would Not Die: How Bayes' Rule Cracked the Enigma Code, Hunted Down Russian Submarines, and Emerged Triumphant from Two Centuries of Controversy*. Yale University Press, New Haven Conn., reprint edition.

I might also mention the Stan users group as an excellent resource for ideas and assistance for using Stan.