STA250, Final Project
Christopher Aden
December 9, 2013

1 Introduction

We will revisit the probit regression model we introduced in Homework 4.

The idea is simple: We observe an indicator function y of an unobserved latent variable, z. To assess a relationship between y and some covariates, we suppose that our z's have a linear relationship to the covariates. If we impose normality on the latent variables, we can say that the probability of $y = 1$ is $p = \Phi(X^T\beta)$. Probit regression is not a huge leap of the imagination if one is familiar with logistic regression–the difference is simply the choice of the link function. The goal is to estimate the influence of the covariates β on the probability of observed y being 1 or 0. Being Bayesians, we can also place prior belief on the values for β.

In the fourth homework assignment, we handled this problem through the magic of Gibbs sampling. This approach was very successful and very simple to implement. Since the conditionals could all be arrived at in closed form, standard R functions could be used, which meant sampling was somewhat fast. We also didn't have the problems Metropolis-Hastings has of rejecting outrageous proposals and staying too conservative, and we are not dependent on a smart choice for a proposal distribution.

These niceties are irrelevant if the samples from the Gibbs Sampler are very dependent. The purpose of this analysis will be to see whether the Metropolis-Hastings algorithm produces better samples than the Gibbs Sampler. We will also discuss certain properties of the MH Algorithm that are nicer than the Gibbs sampler.

2 Methodology

This section will detail how to use Metropolis-Hastings to sample from the posterior of a Bayesian probit regression. The standard representation of Bayesian probit regression is the following model:

$$\begin{aligned} &\beta \sim N(\mu_0, \Sigma_0) \\ &y_i|\beta \sim \text{Bin}(1, \Phi(x_i^T\beta)), \qquad i = 1, \ldots, n, \end{aligned}$$

where $\Phi(x)$ is the CDF of the standard normal distribution. For tractability, assume the responses are conditionally independent given β.

Being unable to directly sample the posterior distribution acquired by a multivariate normal prior and a binomial likelihood, we are reduced to more computational methods–using either Gibbs sampling or Metropolis-Hastings. Assuming our likelihood and prior are

specified as above, the posterior is thus

$$f(\underline{\beta}|\underline{y}) \propto f(\underline{\beta}) \cdot \prod_{i=1}^{n} f(y_i|\beta) \quad \text{(by the cond. indep. of } y_i \text{ and Bayes' Theorem)}$$
$$= N(\mu_0, \Sigma_0) \cdot \prod_{i=1}^{n} \text{Bin}(1, \Phi((x_i^T \beta)))$$

This can be simplified further, but there's not much point. This distribution has no closed form in β. We cannot directly sample from it, so we apply Metropolis-Hastings.

0 Specify a starting point for the Markov Chain, β^0.

For each $i = 1, \ldots, niter$:

1. Propose a new value, β^*, centered on the previous one, β^i.
2. Compute the probability of the posterior at the candidate value $p(\beta^*|\underline{y})$, and the posterior at the current value $p(\beta^i|\underline{y})$. In R, just multiplying the prior by the likelihood, `dmvnorm` by `dbinom` is sufficient. For stability reasons, we work on the log scale, so we'll add the log-prior and the log-likelihood to get the log-posterior.
3. Compute $\ln \alpha^* = \ln p(\beta^*|\underline{y}) - \ln p(\beta^i|\underline{y})$.
4. Draw $U \sim U(0,1)$. If $\ln U \leq \ln \alpha^*$, set $\beta^{i+1} = \beta^*$. Else, $\beta^{i+1} = \beta^i$. The log scale will be equivalent with a greatly-reduced concern for numerical inaccuracy compared to the normal version.
5. The vector β^i, $i = 1, \ldots, n$ represents *dependent* draws from $p(\beta|\underline{y})$.

The following summarizes some of the parameters I use to make the algorithm more efficient.

Starting Values: To emulate the posterior mode with a completely uninformative prior, $\beta^0 = \hat{\beta}$, the MLE of the data. It can be shown that as the sample size increases, the effect of the prior diminishes for any non-degenerate prior, so for a large enough sample size and uninformative enough prior, the MLE will be close to the posterior mode. Using the MLE is a sensible choice if the sample size is large enough that the likelihood carries more weight in the posterior than the prior.

Proposal Distribution The proposal distribution is multivariate normal, centered on the previous value, with a covariance matrix $v^2\Sigma$, where v^2 is a tuning parameter that changes to give a suitable acceptance rate. Σ is the estimated covariance matrix using a Frequentist probit regression, which, for large enough sample size, closely models the structure of the posterior covariance. The normal was chosen because it is a symmetric distribution, which simplifies the math. The covariance was chosen to mimic the variance of the posterior with an uninformative prior (or large sample size).

Tuning Process It is possible that the proposal distribution does not have an appropriate variance. Too high a variance and we will propose improbable values, rejecting too often and ending up with lots of redundant draws. If the variance is too small, we will stay in relatively the same spot, not exploring the space effectively, accepting too often–we gain very little additional information on each draw. The solution to this problem is to keep track of our acceptance rates and make sure it stays within a good acceptance rate window. If the acceptance rate fell outside this band, I adjusted the proposal variance by a scalar factor of $v_i^2 = \exp\left(3 \cdot (\text{acceptance} - \text{ideal})\right)$, where v_i^2 is the scalar proposed during this tuning period (the previous tuning period times the scaling factor). This worked for logistic regression, so we try it again here. I chose about 20% as the ideal acceptance rate, based on simulation results on a multivariate normal.

Iterations I spend very little time choosing a decent number of iterations. Millions of samples takes less than 10 minutes on my machine, with a retuning every thousand observations up to the burn-in.

Burn-In Like the number of iterations, the burn-in is somewhat of a non-issue. I chose to start at the MLE of the likelihood, which means my Markov Chain was pointed at the correct position from the very beginning. I could've chosen a very small burn-in if I wanted, but I also needed to adjust the proposal variance, so I set the burn-in to be more than $n/100$. If I could've adjusted the variance on-the-fly *and* kept those draws instead of burning them, I could've made the burn-in period far less.

3 Results

The speed results are staggering, but must be taken with a grain of salt.

First, we start with the run times. The Gibbs sampler was incredibly slow per iteration, compared to Metropolis Hastings. The need to sample the latent variables, z, meant that we had to do a lot more computational work, as well as require large amounts of memory. Metropolis-Hastings nicely avoids this problem by only needing to calculate the normal CDF of the covariates at each iteration of β. This negates needing truncated normal sampling at all!

However... the Metropolis-Hastings suffers from a very high autocorrelation, because of how dependent the samples are (an inherent flaw in the method). While the Gibbs sampler is also correlated to the previous sample, it is not correlated to the same degree. As such, we require far fewer iterations before we are happy with our effective sample size. Looking at the autocorrelations on the first dataset ($n = 10,000$), we can get an idea of how correlated the MH and Gibbs are, comparatively.

β_i	MH	Gibbs
1	0.932	0.599
2	0.910	0.641
3	0.927	0.680
4	0.924	0.613
5	0.930	0.795
6	0.931	0.625
7	0.923	0.673
8	0.927	0.673

Table 1: Lag-1 Autocorrelations for Metropolis-Hastings versus Gibbs, Dataset 2, $n = 10,000$

How about the effective sample sizes for $3,000$ iterations?

	MH	Gibbs
var1	90.16	608.07
var2	98.67	637.55
var3	113.54	388.06
var4	130.65	568.59
var5	95.11	279.89
var6	102.70	570.49
var7	117.18	438.30
var8	119.51	432.54

Table 2: Effective Sample Size for MH and Gibbs samplers, $niter = 3,000$, $n = 10,000$.

So the effective sample sizes range from 3 times better to as much as 7 times better for Gibbs versus Metropolis-Hastings. One suggestion is to just run the MH algorithm 7 times as long as the Gibbs sampler (or maybe a little more–for padding) to achieve a better effective sample size for the worst case variable. For larger data sets, this may make sense–the MH algorithm drastically outperforms Gibbs sampling as the cost of sampling the latent z's increases.

How does convergence look? On the $n = 10,000$ dataset, we can run the Gibbs sampler with 3000 iterations and the Metropolis Hastings with $8 \cdot 3000 = 24,000$ iterations to compare.

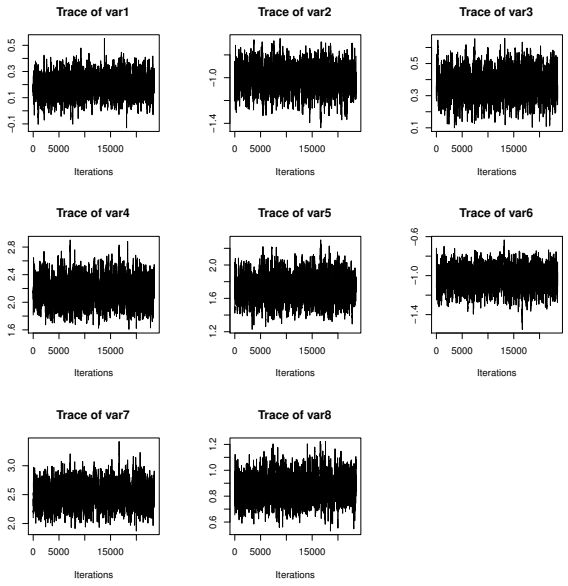

Figure 1: Trace plots for Metropois-Hastings, $niter = 24,000$, $burnin = 500$, $n = 10,000$

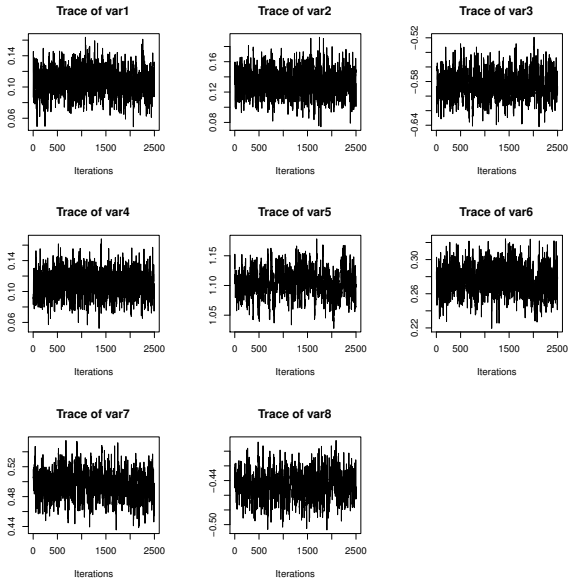

Figure 2: Trace plots for Gibbs Sampler, $niter = 3,000$, $burnin = 500$, $n = 10,000$

The convergence looks okay on both models, so let's go with these iterations. Now for a runtime comparison on the five datasets.

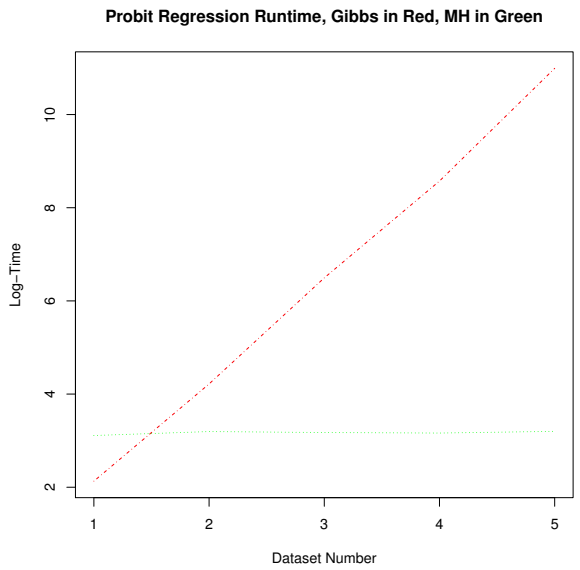

Figure 3: Runtimes on the five datasets, $3,000$ Gibbs samples, $24,000$ MH samples

The Metropolis-Hastings algorithm handily dominates Gibbs in all but the trivial case, despite having to generate eight times as many observations! Given that the Gibbs Sampler is sampling truncated normals–a step the MH algorithm is not doing–this shouldn't be too hard to believe. What is hard to believe are the actual run-times.

	Gibbs Times	MH Times
1	8.42	22.43
2	67.84	24.42
3	661.15	23.91
4	5285.45	23.64
5	59380.66	24.52

Table 3: Run times for data sets of size $n = 10^3, \ldots, 10^7$.

It's as though the Metropolis-Hastings algorithm takes almost constant time, while the Gibbs sampler takes exponential time in n. The fact that MH took 24 seconds to calculate equivalent posterior draws from a dataset with $n = 10,000,000$ and the Gibbs sampler took 17 hours should be enough information for us to strongly prefer using Metropolis-Hastings as the sample size increases.

4 Conclusions

The Metropolis-Hastings algorithm handily manages to beat the runtime of the Gibbs sampler, even considering the very large handicap we pay for high autocorrelation. It's on this basis that makes a lot of sense to prefer MH over Gibbs when doing a Bayesian probit regression.

We pay a bit of an initial cost for having to generate so many more samples than the Gibbs sampler, as is manifested in the first dataset, but we *very* quickly manage to overcome that initial cost, and for any non-trivially-sized data set, the MH algorithm is far faster. There is an exponential speedup for MH versus Gibbs if we're willing to draw more samples in exchange for a higher autocorrelation.

5 Possible Extensions

Writing the MH algorithm in C would've been a more appropriate comparison against the Gibbs sampler which had the advantage of using R constructs solely with very little overhead. Implementing my MH algorithm in C would've made for a more apt comparison.

From a more algorithmic side, Gibbs involved sampling a latent variable, and that was expensive. If the low autocorrelation of Gibbs could be combined with speed of Metropolis-Hastings, this would be an ideal. This could be achieved either through a less-dependent version of MH, or a version of Gibbs sampling that didn't involve having to sample from a truncated Normal distribution for the latent variables.

6 Self-Criticisms

The Metropolis-Hastings could've been written on the GPU very easily. It is a far more tractable GPU problem than the Gibbs sampler which involves generating truncated normals. For the MH algorithm, all we have to do is calculate a normal CDF at each iteration, as well as evaluate a binomial PDF. This is an "embarassingly parallel" problem. I could imagine the run time would be phenomenal compared to a pure-CPU implementation.

I wish I had the time to play around with alternative proposal distributions. The Metropolis algorithm was thoroughly bothered by high autocorrelation. Potentially, a smarter proposal could've made for more efficient sampling. It's also possible that alternative MCMC algorithms would've been good to research. If Gibbs was more efficient, it could've been a smart choice (it was very easy to write).

7 Appendix

R Source Code

Helper_functions is a script containing the majority of the algorithms I used to do Bayesian probit regression. It contains basic functions, as well as my Metropolis Hastings implementation and Gibbs sampler implementation.

```
#Compute posterior of MVN prior and Binom likelihood with probit link. Return log posterior.
posterior.prob = function(Beta, y, beta.0, sigma.0, X){
  #Computes sum of Pr(Beta)~N(beta.0, sigma) )
   logprior = dmvnorm(Beta, mean=beta.0, sigma=sigma.0, log=TRUE)
   loglikelihood = sum(dbinom(y, size=1, prob=as.numeric(pnorm( Beta %*% t(X))), log=TRUE))
   logprior+loglikelihood
}
posterior.prob = cmpfun(posterior.prob)

MH = function(current, candidate, uniform, y, sigma.0, beta.0, X){
  #Computes the Metropolis-Hastings criterion. If the criterion exceeds a uniform (everything logged, btw)
      ,
  #return the new value. Otherwise, return the old one.
  r.top = posterior.prob(candidate, y, beta.0, sigma.0, X)
  r.bottom = posterior.prob(current, y, beta.0, sigma.0, X)
  r = r.top - r.bottom
  if (uniform < r) return(list(sample=candidate, accept=TRUE))
  else return( list(sample=current, accept=FALSE) )
}
MH = cmpfun(MH)

mh.probit <- function(y, X, beta.0=NA, sigma.0=NA, niter=1E5, burnin=1000, print.every=1000, retune=100,
    dataset=NA, verbose=FALSE)
{
  if( !is.na(dataset) ){
      Data = read.table(dataset, header=TRUE)
      y=as.numeric(Data[,1])
      X=as.matrix(Data[,2:ncol(Data)])
    }
  p = ncol(X)
  n = nrow(X)
  #X = scale(X, scale=FALSE) #Rescale the covariates, for stability.
  if (all(is.na(beta.0))) beta.0 = rep(0, p)
  if (all(is.na(sigma.0))) sigma.0 = 1E8 * diag(p)

  beta.samples = matrix(NA, nrow=p, ncol=niter)
  acceptances = rep(NA, niter)
  acceptances[1] = TRUE

  uniform = log(runif(niter)) #Generate all uniforms at once, to utilize R's vectorization.
  probit_reg = glm(y~X-1, family=binomial(link="probit")) #Run a probit regression on the data.
  beta.samples[,1] = probit_reg$coefficients #Start at the Probit Regression MLE
  covar = vcov(probit_reg) #Covariance of Frequentist probit regression. To be scaled by a constant later.

  #Used to adjust the variance in the burn-in period. Jump less if space is large.
  var.scale = ifelse(p > 3, .01, 1)

  #Generate a lot of proposal jumps at once. If Z ~ N(0, S), then Z + b ~ N(b, S) for const b!
  #Vectorization makes things fast, yo.
  perturbations = matrix(nrow=p, ncol=niter)
  perturbations[,1:retune] = t(rmvnorm(retune, mean=rep(0,p), sigma=var.scale * covar))

  #If parameter space is big (p>3), I would expect to be rejected more.
  low   = ifelse(p>3, .10, .20)
  high  = ifelse(p>3, .30, .50)
  ideal = ifelse(p>3, .20, .30)

  for(l in 1:(niter-1)) {
    #Retune loop:
    if (l %% retune == 0 & l <= burnin) {
      acceptance = mean(acceptances[(l-retune):(l-1)])
```

```
60
61        if (acceptance <= low | acceptance >= high) {
62          scaling = exp(3 * (acceptance-ideal))
63          var.scale = var.scale * scaling #If acceptance is greater than ideal, scale it down.
64          if (verbose) {
65            print(paste("Acceptance rate in iterations", l-retune+1, "through", l, "was", round(acceptance
                  ,3), sep=" "))
66            print(paste("Scaling variance by:", round(scaling, 3), sep=" "))
67          }
68        }
69        perturbations[,(l+1):(burnin+1)] = t(rmvnorm( burnin  , mean=rep(0,p), sigma=var.scale * covar))
70      }
71
72      #First legit obs loop:
73      if ( l == (burnin+1) ) {
74        #Generate all proposal perturbations by the final variance, then proceed as usual.
75        if (verbose) print(paste("Starting the first non-burned observation at l=", l, sep=""))
76        perturbations[ , l:niter] = t(rmvnorm( niter-l+1 , mean=rep(0,p), sigma=var.scale * covar))
77      }
78
79      current = beta.samples[,l]
80      candidate = current+perturbations[,l]
81      foo = MH(current, candidate, uniform[l], y, sigma.0, beta.0, X)
82      beta.samples[,l+1] = foo$sample
83      acceptances[l+1] = foo$accept
84
85      if (l %% print.every == 0) print(paste("Current Metropolis iteration:", l, sep=" "))
86    }
87    beta.samples.burned = t(beta.samples[,(burnin+1):niter])
88    return ( mcmc(beta.samples.burned) )
89  }
90  mh.probit = cmpfun(mh.probit)
91
92  #Does Gibbs Sampling for Bayesian Probit Regression.
93  gibbs_probit = function(y, X, beta_0 = NA, Sigma_0_inv=NA, niter=1E5, burnin=1E3, print.every=100,
        Datapath=NULL)
94  { #Noninformative prior--don't need to specify prior variance or mean.
95    if( !is.na(Datapath) ){
96      Data = read.table(Datapath, header=TRUE)
97      y=as.numeric(Data[,1])
98      X=as.matrix(Data[,2:ncol(Data)])
99    }
100   p = ncol(X)
101   n = nrow(X)
102   if (all(is.na(beta_0))) beta_0 = rep(0, p)
103   if (all(is.na(Sigma_0_inv))) Sigma_0_inv = matrix(0, p, p)
104   z = numeric(p)
105   beta = matrix(NA, nrow=niter, ncol=p)
106   beta[1, ] = lm.fit(X,y)$coefficients #Start with OLS.
107
108   uppers = ifelse(y==1, Inf, 0) #If y=1, then z>0, so trunc points were (0, Inf)
109   lowers = ifelse(y==1, 0, -Inf) #If y=0, z<=0, so trunc points were (-Inf, 0)
110
111   #If Flat Prior, posterior variance is (X'X)^-1 .
112   beta_var = solve(Sigma_0_inv + t(X) %*% X)
113
114   for (i in 1:(niter-1)) {
115     if((i %% print.every) == 0) print(paste("Current Gibbs iteration:", i, sep=""))
116     #Sample (z|y,beta)
117     z_mean = X %*% beta[i,]
118     z = rtnorm(n, mean=z_mean, sd=rep(1, n), lower=lowers, upper=uppers)
119
120     #Sample (beta | z, y)
121     if (all(Sigma_0_inv==0)) { beta_mean = lm.fit(X, z)$coefficients } else{
122       beta_mean = beta_var %*% (Sigma_0_inv %*% beta_0 + t(X) %*% z)}
123     beta[i+1, ] = rmvnorm(1, mean=beta_mean, sigma=beta_var)
124   }
125   return(mcmc(beta[(burnin+1):niter, ]))
126 }
127 gibbs_probit = cmpfun(gibbs_probit)
```

Probit_mcmc handles the work of calling the simulated data sets and running the Gibbs and MH samplers over them. It also deals with saving and plotting.

```
#setwd("~/Dropbox/sta250/Assignments/Final")
library(msm)
library(coda)
library(compiler)
library(mvtnorm)
# source("sim_probit.R") #Generate the data
source("helper_functions.R")

nds = 5 #Number of datasets
MH_times = Gibbs_times = numeric(nds)
filenames = paste0("data_0", 1:nds, ".txt")
MH_samples = Gibbs_samples = list()
length(MH_samples) = length(Gibbs_samples) = nds

for (i in 1:nds){ #Times saved seperately so I could write the paper asyncronously.
    Gibbs_times[i] = system.time({Gibbs_samples[[i]] = gibbs_probit(Sigma_0_inv=diag(8)/1E8, niter=3000,
        burnin=500, Datapath=filenames[i])})["elapsed"]
    MH_times[i] = system.time({MH_samples[[i]] = mh.probit(sigma.0=1E8 * diag(8), niter=24000, burnin=500,
        dataset=filenames[1])})["elapsed"]
}

save(Gibbs_times, file="Gibbs_times.Rdata")
save(MH_times, file="MH_times.Rdata")

#Convergence plots.
MH_obj = MH_samples[[2]]
Gibbs_obj = Gibbs_samples[[2]]

setEPS()
postscript('MH_Trace.eps')
plot(MH_obj, density=FALSE)
dev.off()

setEPS()
postscript('Gibbs_Trace.eps')
plot(Gibbs_obj, density=FALSE)
dev.off()

library(xtable)
(effectiveSize(MH_samples[[2]]) / effectiveSize(Gibbs_samples[[2]])) ^-1
xtable(cbind(MH = autocorr.diag(MH_obj)[2,] , Gibbs = autocorr.diag(Gibbs_obj)[2,]), digits=3)
xtable(cbind(MH_Eff = effectiveSize(MH_samples[[2]]), Gibbs_Eff = effectiveSize(Gibbs_samples[[2]])))
xtable(cbind(Gibbs=Gibbs_times, MH=MH_times))

setEPS()
postscript('Timings.eps')
plot(1:nds, log(Gibbs_times), xlab="Dataset Number", ylab="Log-Time", main="Probit Regression Runtime,
    Gibbs in Red, MH in Green", type='l', lty=4, col='red',
     ylim=c(min(log(c(Gibbs_times, MH_times))), max(log(c(Gibbs_times, MH_times))) ) )
lines(1:nds, log(MH_times), lty=3, col='green')
dev.off()
```