STA250, Homework 1
Christopher Aden
October 21, 2013

Proof that Gibbs Sampler Will Converge:

Part B

Let $p(\underline{x})$ denote the probability density function of a vector of random variables $x_1, \ldots, x_p$. We will assume that the Markov Chain that results from running the Gibbs Sampler is ergodic. We want to show that the stationary distribution of the p-component Gibbs sampler is indeed this target distribution $p(\underline{x})$.

To avoid having to write a bunch of stuff in LaTeX, let me introduce a couple pieces of notational shorthand. Let x^t denote the t-th iteration of the Gibbs Sampler (**not** x to the power t!). Additionally, $x_{[i:j]} = (x_i, x_{i+1}, \ldots, x_{j-1}, x_j)$, just like the slice operation in Python or the colon function in R.

By the assumption of ergodicity, we know that the Markov Chain formed by the Gibbs Sampler, $\underline{x}^1, \underline{x}^2, \ldots$, converges to a stationary distribution, denoted by $\pi(\underline{x})$ in the notes. In the continuous state space case, it is a property of a stationary Markov Chain that

$$\pi(\underline{y}) = \int_{\mathbb{R}^p} \pi(\underline{x}) p(\underline{x}, \underline{y}) d\underline{x}.$$

That is, we must have that the probability of arriving at state $\underline{y}$ is the integral of the probability of being at state $\underline{x}$ times the probability of moving from state $\underline{x}$ to state $\underline{y}$.

This will require only two steps: deriving a closed form for the transition density $p(\underline{x}, \underline{y})$ and showing that the (multivariate) integral of the stationary distribution times the transition density gives us the stationary distribution. We will assert that one such stationary distribution that satisfies this property is the target distribution, then use the uniqueness of the stationary distribution to claim the target distribution is the unique stationary distribution.

Transition Density: What is the density of $\underline{x}^{t+1}$, given $\underline{x}^t$? For this, by conditional expectation,

$$\begin{aligned}
p(\underline{x}^{t+1}|\underline{x}^t) &= p(x_1^{t+1}, x_2^{t+1}, \ldots, x_p^{t+1}|\underline{x}^t) \\
&= p(x_1^{t+1}|\underline{x}^t) \cdot p(x_2^{t+1}, \ldots, x_p^{t+1}|x_1^{t+1}, \underline{x}^t_{[2:p]}) \\
&= p(x_1^{t+1}|\underline{x}^t) \cdot p(x_2^{t+1}|x_1^{t+1}, \underline{x}^t_{[2:p]}) \cdot p(x_3^{t+1}, \ldots, x_p^{t+1}|x^{t+1}_{[1:2]}, \underline{x}^t_{[3:p]}) \\
&= \ldots (\text{Repeated conditioning, unrolling each term conditioned on all the rest,}) \\
&= p(x_1^{t+1}|\underline{x}^t) \cdot p(x_2^{t+1}|x_1^{t+1}, \underline{x}^t_{[2:p]}) \ldots p(x_p^{t+1}|\underline{x}^{t+1}_{[1:p-1]}, x_p^t)
\end{aligned}$$

Intuitively, you can think of this as similar to what the Gibbs Sampler does. The distribution of the joint is the product of the conditionals of each random variable at the current iteration,

given the variables already drawn in the current iteration and the previous iteration for the variables not yet drawn in the current iteration.

Stationarity: The target distribution $p(\underline{x})$ is stationary iff

$$\begin{aligned} p(\underline{x}^{t+1}) &= \int_{\mathbb{R}^p} p(\underline{x}^t)p(\underline{x}^{t+1}|\underline{x}^t)d\underline{x}^t, \forall \underline{x}^t \in \mathbb{R}^p. \\ &= \int_{\mathbb{R}} \cdots \int_{\mathbb{R}} p(x_1^t, x_2^t, \ldots, x_p^t)p(x_1^{t+1}, x_2^{t+1}, \ldots, x_p^{t+1}|x_1^t, x_2^t, \ldots, x_p^t)dx_1^t dx_2^t \cdots dx_p^t, \end{aligned}$$

for all $x_i^t \in \mathbb{R}$, $i = 1, \ldots, p$. Recall that $f(y) = \int_{\mathbb{R}} f(x, y)dx = \int_{\mathbb{R}} f(x) \cdot f(y|x)dx$ for arbitrary real-valued random variables y and x. The above integral can be seen to be the product of a marginal of $\underline{x}^t$ and the conditional of $\underline{x}^{t+1}$ given $\underline{x}^t$, integrated over the entire parameter space of $\underline{x}^t$. By similar logic to my univariate example earlier in the paragraph, we see that the result of this will be the marginal distribution of $\underline{x}^{t+1}$.

$$\Rightarrow \int_{\mathbb{R}^p} p(\underline{x}^t)p(\underline{x}^{t+1}|\underline{x}^t)d\underline{x}^t = p(\underline{x}^{t+1}).$$

Combining the uniqueness of the stationary distribution with the stationarity of the target distribution, we have that the Gibbs Sampler Markov Chain converges to the target distribution. ■

Part A

First, a story: An engineer and a mathematician shared an apartment. Their kitchen was equipped with an electric stove, and every morning someone had placed a pot of water on the back-right burner so they could make coffee. They both knew what knob turned on this burner. One morning the engineer came into the kitchen and found the pot was on the front-left burner. He got out the stove's schematics and followed the wiring diagram and finally figured out which knob turned on this burner and he then used that knob and made the coffee. The next morning the mathematician came in and also found the pot on the front-left burner. He moved the pot to the back-right burner, thereby reducing the problem to one which he had already solved.

Proof: Let $p = 2$. Refer to Part B. ■

Metropolis-Hastings with Simulated 2-dimensional Parameter

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. Assume our likelihood and prior are

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

where $\text{expit}(u) = \exp(u)/(1+\exp(u))$. For tractability, assume the responses are conditionally independent given β.

The posterior is thus

$$
\begin{aligned}
f(\underline{\beta}|\underline{y}) &\propto f(\underline{\beta}) \cdot \prod_{i=1}^{n} f(y_i|\beta) \quad \text{(by the cond. indep. of } y_i \text{ and Bayes' Theorem)} \\
&= N(\mu_0, \Sigma_0) \cdot \prod_{i=1}^{n} \text{Bin}(m_i,\ \text{expit}(x_i^T\beta)) \\
&\propto \exp\left(-\frac{1}{2}(\beta-\mu_0)^T\Sigma_0^{-1}(\beta-\mu_0)\right) \prod_{i=1}^{n} \binom{m_i}{y_i} \text{expit}(x_i^T\beta)^{y_i}(1-\text{expit}(x_i^T\beta))^{m_i-y_i} \\
&\propto \exp\left(-\frac{1}{2}(\beta-\mu_0)^T\Sigma_0^{-1}(\beta-\mu_0)\right) \prod_{i=1}^{n} \frac{\exp(x_i^T\beta)^{y_i}}{1+\exp(x_i^T\beta)^{y_i}} \frac{1}{1+\exp(x_i^T\beta)^{m_i-y_i}} \\
&= \exp\left(-\frac{1}{2}(\beta-\mu_0)^T\Sigma_0^{-1}(\beta-\mu_0)\right) \prod_{i=1}^{n} \frac{\exp(x_i^T\beta)^{y_i}}{1+\exp(x_i^T\beta)^{y_i}} \frac{1+\exp(x_i^T\beta)^{y_i}}{1+\exp(x_i^T\beta)^{m_i}} \\
&= \exp\left(-\frac{1}{2}(\beta-\mu_0)^T\Sigma_0^{-1}(\beta-\mu_0)\right) \prod_{i=1}^{n} \frac{\exp(x_i^T\beta)^{y_i}}{1+\exp(x_i^T\beta)^{m_i}}
\end{aligned}
$$

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 must apply Metropolis-Hastings.

1. Specify a starting point for the Markov Chain, β^0.
2. At each increment i in the Markov Chain, propose a new value, β^*, centered on the previous one, β^i.
3. Compute the probability of the posterior at the candidate value $p(\beta^*|\underline{y})$ (derived up to proportionality above), as well as the posterior at the current value $p(\beta^i|\underline{y})$. In R, we work up to proportionality, so this is as simple as multiplying the prior by the likelihood, `dmvnorm` by `dbinom`. 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.
4. Compute $\alpha^* = \frac{p(\beta^*|\underline{y})}{p(\beta^i|\underline{y})}$.
5. Draw $U \sim U(0,1)$. If $U \leq \alpha^*$, set $\beta^{i+1} = \beta^*$. Else, $\beta^{i+1} = \beta^i$. Again, because of the log-scale, we can compare a log-uniform against the difference of the log-posterior at the candidate to the log-posterior at the current value of the Markov Chain, and the results will be equivalent with a greatly-reduced concern for numerical inaccuracy.
6. The vector $\beta^i, i = 1, \ldots, n$ represents dependent draws from the target distribution $p(\beta|\underline{y})$.

This algorithm has a ton of knobs and switches to play with. The following summarizes some of the parameters I use to make the algorithm more efficient.

Starting Values: $\beta^0 = \hat{\beta}$, the MLE of the data, was chosen. We can think of this as being close to the posterior mode if we had an uninformative prior distribution. It can be shown that as the sample size increases, the effect of the prior diminishes for any non-degenerate prior, so for our sample size, the MLE will be quite 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 does.

Proposal Distribution The proposal is generated from the proposal distribution, which I chose to be a bivariate 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 for samples. Σ is the estimated covariance matrix using a Frequentist logistic regression, which, for large enough sample size, closely models the structure of the posterior covariance. The reason for choosing a bivariate normal as my proposal distribution was out of simplicity. Using the bivariate normal, the probably of going from the candidate value to the current value is the same as going from current to candidate, a property called symmetry. This simplifies the calculation of the Metropolis-Hastings acceptance statistic. The choice of that strange form for the covariance matrix is an attempt at mimicking the variance of the posterior distribution. I acquired this matrix by fitting a logistic regression in R with `glm`, then using its estimate for the covariance as my proxy for the posterior variance.

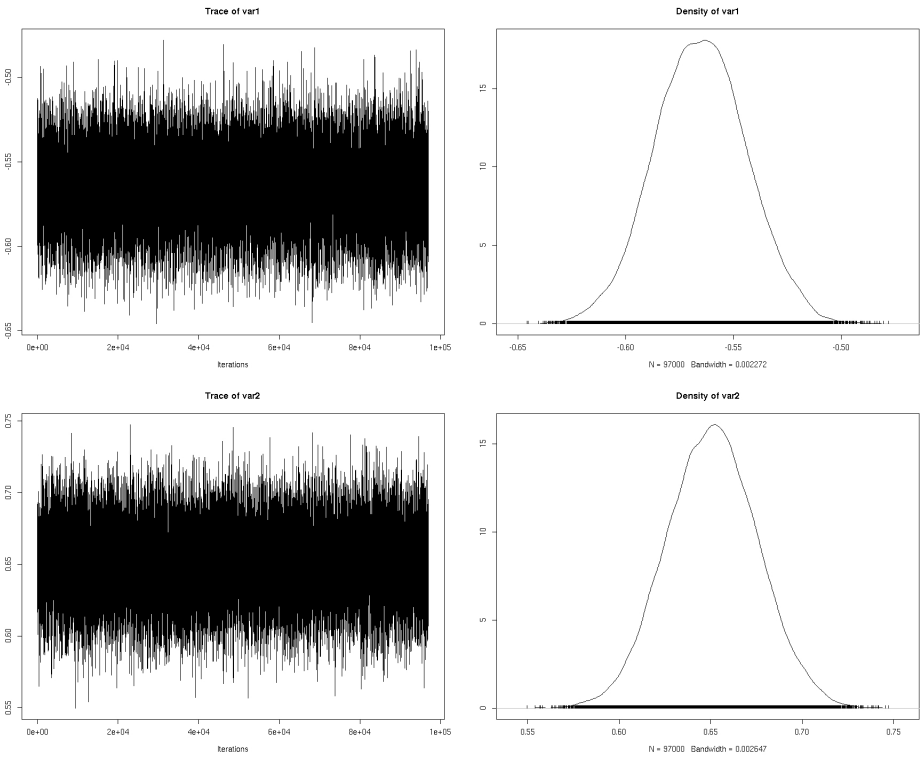
Tuning Process Every so often, we will go through a retuning process. It is possible that the proposal distribution does not have an appropriate variance. Too high a variance and we will propose bad values, rejecting too often and ending up with lots of redundant draws. If the variance is too small, we will stay in relatively the same spot, not exploring the space effectively, accepting too often. In both cases, we end up with a very small effective sample size, and gaining 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. I chose 20% to 50% for this 2-dimensional example. This comes based on heuristic moreso than any analytical reason. This ended up being pretty reasonable, so I stuck with it. 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). Acceptance is the acceptance ratio, and ideal is the ideal accept rate. For the two-dimensional case, I chose 30%, which is a reasonable balance between accepting and rejecting. If the acceptance rate is higher than 30%, this scaling will increase the proposal variance (to make more adventurous jumps). If the acceptance is less, the scaling will decrease the variance. The coefficient of 3 was chosen as a factor of how much to scale by each time and was chosen by hand-tuning heuristically.

Number of Iterations Computing power is inexpensive in 2013. As long as I have other things to do, I would rather have the computer spend two hours running my code than have to spend 30 minutes of my time optimizing the program to make it run quicker.

Consistent with that philosophy, I spend very little time choosing a decent number of iterations. A couple million samples takes less than 10 minutes on my Linux machine, with a retuning every thousand observations up to the burn-in.

Burn-In Period Like the number of iterations, the burn-in is somewhat of a non-issue. I cleverly 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 around $10,000$. 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. I hit stationarity after less than a thousand draws most times.

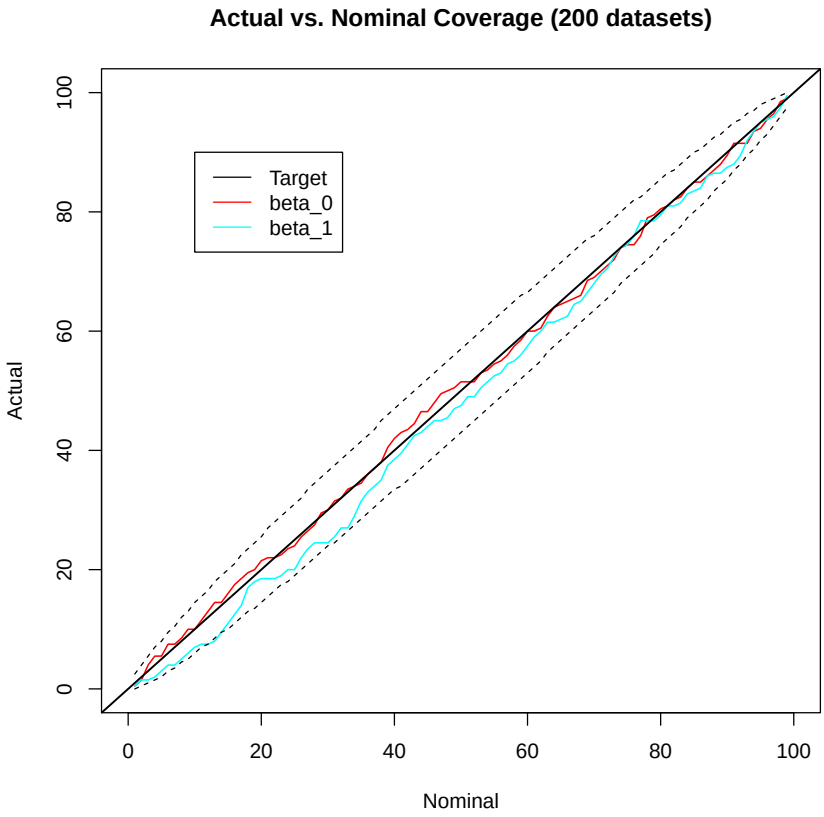
For this simple 2-dimensional problem, Metropolis-Hastings makes short work of sampling the posterior. While the effective sample size is only about 17% of the total sample size, this is good enough if we run enough iterations of the algorithm. We can thin the data (taking only every k^{th} iteration) if we are worried about autocorrelation between samples, but I didn't bother for only two dimensions. Looking at the traceplots, we see that stationarity has been attained.



Trace and density plots for (β_1, β_2), $n = 10^5$, burn = 3000, retune = 100.

A similar picture can be seen on the other 200 datasets generated on the compute cluster. On the topic of coverage probabilities, you may take a look at the coverage summaries.

Things turned out okay. Keep in mind that there is randomness associated with the Markov Chain. Running the simulation another time changes the coverage probabilities.



Coverage Line Plot for (β_1, β_2), $n = 5 \cdot 10^5$, burn = 10^4, retune = 10^3.

	β_0	β_1
p_01	0.01	0.01
p_05	0.06	0.03
p_10	0.10	0.07
p_25	0.24	0.20
p_50	0.52	0.47
p_75	0.74	0.74
p_90	0.90	0.88
p_95	0.94	0.95
p_99	0.99	0.99

Coverage Probabilities for (β_0, β_1).

Metropolis-Hastings with 11-dimensional Parameter on Real Data

The previous approach did not work very well on this particular problem (wait for the punchline after the table). I tried do it in the same fashion, with 100,000 iterations, a burn-in of 20,000, an acceptance range between 10% and 30% to accommodate the more frequent number of rejections, the algorithm finished quickly, but the effective sample size for the Texture variable was not much larger than 5. In the interest of not re-writing my Metropolis-Hastings code from Problem 2, I ramped up the iterations to 5,000,000 (*five million!*). The effective sample size of the Texture variable (X11) was still only 10.

The dimension of the parameter space is partially to blame. It's difficult to propose plausible values in 11-dimensional space and have them be more likely than another previously-selected value. But the "Curse of Dimensionality" cannot fully explain the difficulty of sampling the βs in this problem. Looking at the covariance matrix of the 11 parameters, we can see where we might have trouble.

β_1	β_2	β_3	β_4	β_5	β_6	β_7	β_8	β_9	β_{10}	β_{11}
165.19	0.0003	413.82	813.91	65.94	7319.94	0.26	13.81	1021.12	113.01	0.004

Variances for MLEs $\hat{\beta}_i$, $i = 1, \ldots, 11$.

The scale is simply ridiculous! Variables X2 (area), X7 (perimeter), and X11 (texture) are on the order of tenths to thousandths, while X6 (fracdim) and X9 (smoothness) are in the thousands. Between the smallest-varied and largest-varied variables, we're looking at a six order of magnitude difference! This makes it very hard to tune our proposal distribution. However, a clever choice of proposal distribution solves a number of problems. I was initially unable to get the MH algorithm to converge correctly with this complicated space. Using the estimated covariance matrix created from a logistic regression solved a large part of the problem, but was still not enough until about 10pm sitting in my office, I realized I had been using the wrong prior ($\frac{1}{1000} \cdot I_{11} \neq 1000 \cdot I_{11}$)! Changing the code drastically increased my effective sample size to the point that Metropolis-Hastings became a viable option for sampling an 11-dimensional parameter space. The convergence was not as nice as the 2-dimensional case, but people who don't want to spend extra time re-factoring optimized code can't be choosers. Running the chain a long time handily solves this problem anyway. Let's talk about the autocorrelation. Using built-in R function `acf()`, computing the lag-1 autocorrelation is trivial.

β_1	β_2	β_3	β_4	β_5	β_6	β_7	β_8	β_9	β_{10}	β_{11}
0.95	0.96	0.94	0.93	0.96	0.83	0.96	0.96	0.93	0.96	0.96

The Lag-1 autocorrelation for each β_i.

Terrible. Each draw is very correlated to the previous draw. One simple and slightly naive

way to handle this problem is to thin the samples, removing every k^{th} observation. Thinning will reduce the effective sample size, but since I ran the chain for half a million iterations, I can afford to lose 98% of my Markov Chain iterations to cut down the autocorrelation.

β_1	β_2	β_3	β_4	β_5	β_6	β_7	β_8	β_9	β_{10}	β_{11}
0.16	0.19	0.12	0.05	0.14	0.05	0.20	0.18	0.01	0.20	0.14

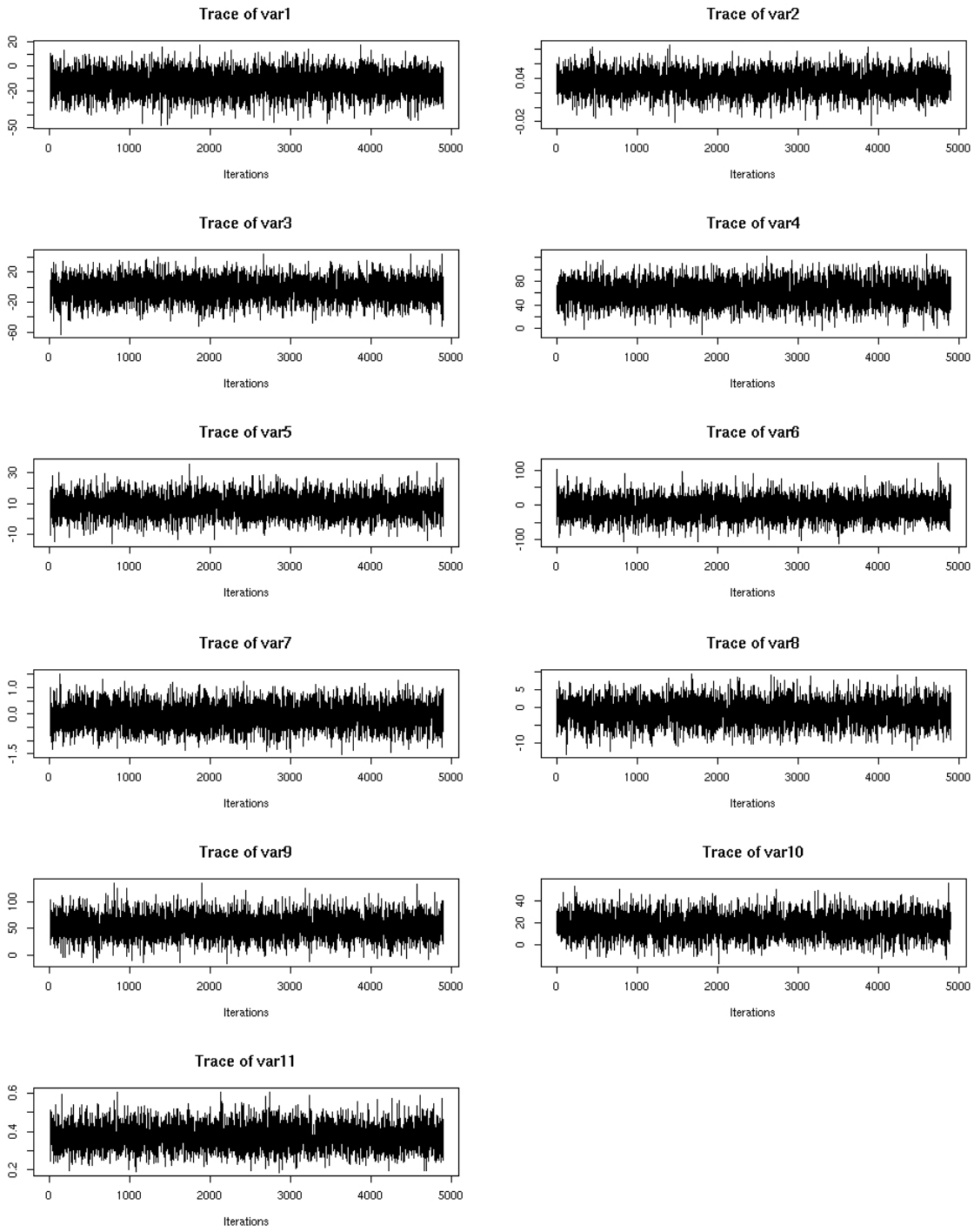
The Lag-1 autocorrelation for each β_i using 50-thinning.

β_1	β_2	β_3	β_4	β_5	β_6	β_7	β_8	β_9	β_{10}	β_{11}
0.08	0.05	-0.04	-0.02	0.08	-0.05	0.02	0.02	-0.01	0.08	-0.01

The Lag-1 autocorrelation for each β_i using 100-thinning.

The 100-thinned version is basically uncorrelated at this point. The effective sample size demonstrated this, as most variables had effective sample sizes that were very close to actual number of thinned samples. For my large iteration choice, this yielded 5,000 uncorrelated samples from the posterior, which was good enough for me. In hindsight, I could've been fine running 50,000 iterations and still had 500 uncorrelated samples.

Convince yourself from my traceplots that the chain has good convergence.



Trace and density plots for $(\beta_1, \ldots, \beta_{11})$, $n = 5 \cdot 10^5$, burn = 10^4, retune = 100.

Next, we consider the problem of variable selection. Bayesian variable selection works in a different way from Frequentist variable selection, since we don't have the same kind of

Neyman-Pearson formulation that we did before–the concept of controlling a Type 1 error for a model isn't as meaningful. One approach is simply to look at the posterior credible intervals and see which ones contain zero, because while the magnitude of each posterior parameter does depend on the scaling of each variable, whether the posterior intervals contain zero is only very weakly dependent on the scaling. From the table, it seems the most important variables (the ones that don't contain zero) are β_4, β_9, and β_{11} (concave points, smoothness, and texture, respectively).

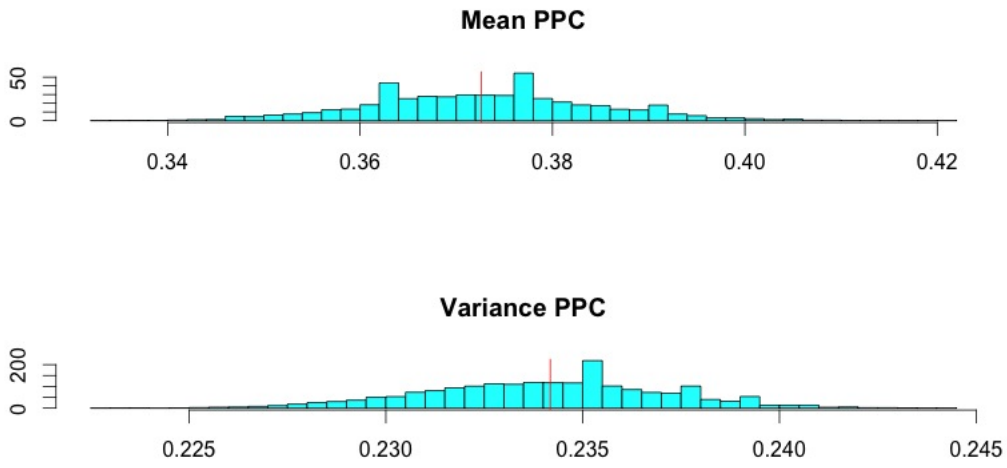
	β_1	β_2	β_3	β_4	β_5	β_6	β_7	β_8	β_9	β_{10}	β_{11}
2.5%	-36.31	0.00	-30.85	19.69	-6.33	-67.32	-0.85	-8.15	15.52	-0.66	0.26
97.5%	4.10	0.06	22.50	98.34	23.22	51.30	0.84	4.49	87.96	36.64	0.49

95% Central Credible Interval

Finally, I will talk about checking our model. Model checking in Bayesian inference is done through posterior predictive checking. This procedure involves drawing parameters $\underline{\beta}$ from the posterior distribution, then generating simulated data from the assumed likelihood, given those posterior values as the parameters. If the model is an accurate representation of the data, the posterior predictive distributions should not be all that different from the true data, which we can corroborate using summary statistics (mean, median, variance, and the like). Our algorithm is as follows:

1. Draw $\underline{\beta}^i$ from the posterior, $p(\underline{\beta}|\underline{y})$, for $i = 1, \ldots, M$.
2. Simulate M datasets $\tilde{y}^i$, where $\tilde{y}^i \sim \text{Bin}(m_i, \text{expit}(X^T\beta^i))$.
3. Compute a summary statistic on each dataset, $T(\tilde{y}^i)$, then create a histogram with the M summary statistics.

If the model is correct, the true summary statistic should sit somewhere near the center of the histogram (or at least not in the far tails of the histogram). Since the likelihood is Bernoulli data, minimum and maximum statistics are not useful. I went with the mean and the variance, and we can see from the plots that the binomial likelihood model is quite sensible in this case.



Posterior Predictive Checks for the Mean and Variance

The red line indicates the true mean and variance. The mean was chosen as a measure of center, and the variance was the choice for measuring dispersion. There is an inflexible relationship between the mean and the variance with the binomial, so if another model was more appropriate, checking both variance and mean could have given some insight. As it turns out, the lines sit very closely to the middle of both histograms, leading us to believe that the model is quite sound as a way of modeling both the mean and the variance. It is possible that the model is not a good fit through some other summary statistic. A more thorough method would be to look at a number of quantiles as the summary statistics. This provides a very accurate picture of both the tails and the center of the model. Similar to the QQ Plot, the posterior predictive quantiles ought to match up against the true corresponding quantiles if the model is valid (think of this as as many histograms as there are quantiles). In the interest of parsimony, I chose only a measure of center and a measure of dispersion.

R Source Code

Helper_functions is a script containing the majority of the algorithms I used to do Bayesian Logistic regression. It contains basic functions, as well as my Metropolis Hastings implementation, computation of quantiles, and posterior predictive checks.

```
expit = function(X, Beta){
  exp(X %*% Beta) / {1+exp(X %*% Beta)}
}
expit = cmpfun(expit)

#Compute posterior of MVN prior and Binom likelihood. Return log posterior.
posterior.prob = function(Beta, y, beta.0, sigma.0, m, 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=m, prob=expit(X,Beta), log=TRUE))
  logprior+loglikelihood
}
posterior.prob = cmpfun(posterior.prob)

MH = function(current, candidate, uniform, m, 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, m, X)
  r.bottom = posterior.prob(current, y, beta.0, sigma.0, m, 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)

"bayes.logreg.samples" <- function(m, y, X, beta.0=NA, sigma.0=NA, niter=10000, burnin=1000, print.every=1000, retune=100, verbose=TRUE)
{
  p = ncol(X)
  n = nrow(X)
  if (all(is.na(beta.0))) beta.0 = rep(0, p)
  if (all(is.na(sigma.0))) sigma.0 = diag(p)
  Sigma.0.inv = solve(sigma.0) #Compute the inverse of precision--covariance. Speeds up inner loop

  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.
  logist_reg = glm(cbind(y, m-y) ~ X-1, family="binomial") #Run a logistic regression on the data.
  beta.samples[,1] = logist_reg$coefficients #Start at the Logistic Regression MLE
  covar = vcov(logist_reg) #Covariance of Frequentist logistic 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)])

      if (acceptance <= low | acceptance >= high) {
        scaling = exp(3 * (acceptance-ideal))
        var.scale = var.scale * scaling #If acceptance is greater than ideal, scale it down.
        if (verbose) {
          print(paste("Acceptance rate in iterations", l-retune+1, "through", l, "was", round(acceptance,3), sep=" "))
          print(paste("Scaling variance by:", round(scaling, 3), sep=" "))
        }
      }
      perturbations[,(l+1):(burnin+1)] = t(rmvnorm( burnin  , mean=rep(0,p), sigma=var.scale * covar))
    }

    #First legit obs loop:
    if ( l == (burnin+1) ) {
      #Generate all proposal perturbations by the final variance, then proceed as usual.
      if (verbose) print(paste("Starting the first non-burned observation at l=", l, sep=""))
      perturbations[ , l:niter] = t(rmvnorm( niter-l+1 , mean=rep(0,p), sigma=var.scale * covar))
    }

    current = beta.samples[,l]
```

```
    candidate = current+perturbations[,l]
    foo = MH(current, candidate, uniform[l], m, y, sigma.0, beta.0, X)
    beta.samples[,l+1] = foo$sample
    acceptances[l+1] = foo$accept

    if (l %% print.every == 0) print(paste("Current iteration:", l, sep=" "))
  }
  beta.samples.burned = t(beta.samples[,(burnin+1):niter])
}
bayes.logreg.samples = cmpfun(bayes.logreg.samples)


"bayes.logreg" <- function(m, y, X, beta.0=NA, sigma.0.inv=NA, niter=10000, burnin=1000, print.every=1000,
    retune=100, verbose=TRUE, thin=1) {
  # m: Vector containing the number of trials for each observation (of length n)
  # y: Vector containing the number of successes for each observation (of length n)
  # X: Design matrix (of dimension n p)
  # beta.0: Prior mean for beta (of length p)
  # Sigma.0.inv: Prior precision (inverse covariance) matrix for beta (of dimension p p)
  # niter: Number of iterations to run the MCMC after the burnin period
  # burnin: Number of iterations for the burnin period (draws will not be saved)
  # print.every: Print an update to the user after every period of this many iterations
  # retune: Retune the proposal parameters every return iterations. No tuning should be done after the
      burnin period is completed
  # verbose: If TRUE then print lots of debugging output, else be silent
  # thin: integer-- Take every "thin"^th observation from the chain.
  if (all(!is.na(sigma.0.inv))) {sigma.0 = solve(sigma.0.inv)}
  samples = bayes.logreg.samples(m, y, X, beta.0, sigma.0, niter, burnin, print.every, retune, verbose)
  Posterior_Draws = samples[seq.int(1, (niter-burnin), by=thin), ]

  cbind(quantile(Posterior_Draws[,1], probs=seq(.01, .99, length=99)),
        quantile(Posterior_Draws[,2], probs=seq(.01,.99, length=99)))
}

PPC.Graphs = function(Y, X, m=1E5, mi=rep(1,length(Y)), posterior=chain_thinned){
  #Y: Response variable
  #X: Design matrix (explanatory variables)
  #m: Number of PPC datasets to generate.
  #mi: Number of trials for each level of the design matrix.
  #posterior: MCMC object or matrix (n x p) of posterior draws.

  n = length(Y)
  niter = nrow(posterior) #Number of MCMC draws from posterior

  ppc.datasets <- matrix(nrow=m, ncol=n)
  rownames(ppc.datasets) <- paste("PPC_",1:m,sep="")

  index = sample.int(niter, m, replace=TRUE) #Sample the posteriors we want.

  #Calculate p_i using posterior draw.
  #Rows: Probability at each X_r; Columns: PP dataset m_i.
  p.i = sapply(index, function(i) expit(X, posterior[i,]))

  PPC_Data = sapply(1:m, function(i) rbinom(n, mi, p.i[,i]))
  PPC_mean = apply(PPC_Data, 2, mean) #Apply the mean to each PPC Dataset
  PPC_var = apply(PPC_Data, 2, var) #And the variance

  true_mean = mean(Y)
  true_var = var(Y)

  par(mfrow=c(2,2))
  truehist(PPC_mean,main="Posterior Predictive Check on Mean", xlab="", ylab="")
  abline(v=true_mean, col="red")

  truehist(PPC_var,main="Posterior Predictive Check on Variance", xlab="", ylab="")
  abline(v=true_var, col="red")

  NULL
}
PPC.Graphs = cmpfun(PPC.Graphs)
```

The BLR_Fit file contains the code to run the simulations when the true value was known. It outputs a matrix containing the quantiles for each of the two parameters.

```r
library(mvtnorm)
library(coda)
library(compiler)
suppressWarnings(library(mvtnorm))
suppressWarnings(library(coda))
suppressWarnings(library(compiler))

setwd("/home/caden11/sta250/Explorations-into-Computational-Statistics/HW1/BayesLogit/")

################################################################################################
################################################################################################
## Handle batch job arguments:
args <- commandArgs(TRUE) #1-indexed version is used now.

cat(paste0("Command-line arguments:\n"))
print(args)

####
# sim_start ==> Lowest simulation number to be analyzed by this particular batch job
###

########################
sim_start <- 1000
length.datasets <- 200
beta.0=c(0,0)
sigma.0.inv = diag(2)
########################

if (length(args)==0){
  sinkit <- FALSE
  sim_num <- sim_start + 1
  set.seed(1330931)
} else {
  # Sink output to file?
  sinkit <- TRUE
  # Decide on the job number, usually start at 1000:
  sim_num <- sim_start + as.numeric(args[1])
  # Set a different random seed for every job number!!!
  set.seed(762*sim_num + 1330931)
}

# Simulation datasets numbered 1001-1200

################################################################################################
################################################################################################

# Read data corresponding to appropriate sim_num:
Data = read.csv(paste("./data/blr_data_", sim_num, ".csv", sep=""))
pars = read.csv(paste("./data/blr_pars_", sim_num, ".csv", sep=""))

# Extract X and y:
X = as.matrix(Data[,3:4], ncol=2)
y = Data$y
m = Data$n

################################################
# Set up the specifications:
p = ncol(X)
beta.0 <- matrix(rep(0,p))
sigma.0 <- sigma.0.inv <- diag(p)

niter <- 5E5
burnin = 1E4
print.every = niter/10
retune = 1E3
verbose = TRUE
local = FALSE #If running a cluster job, don't the piece to get raw samples--just quantiles.
thin = 100 #Take only every 100 observations to generate the quantiles.
################################################

# Fit the Bayesian model:
source("./helper_functions.R")
if (local) {
```

```
Posterior_Draws = bayes.logreg.samples(m, y, X, beta.0, sigma.0, niter, burnin, print.every, retune,
    verbose)
chain = mcmc(Posterior_Draws) #Markov Chain objects, to make plots and ESS easy.
effectiveSize(chain) #Effective Sample Size for each variable.
plot(chain) #Convergence plots
}

# Extract posterior quantiles...
posterior_quantiles = bayes.logreg(m, y, X, beta.0, sigma.0.inv, niter, burnin, print.every, retune,
    verbose, thin)

# Write results to a (99 x p) csv file...
write.table(posterior_quantiles, file=paste(getwd(), "/results/blr_res_", sim_num, ".csv", sep=""), row.
    names=FALSE, col.names=FALSE, sep=",")

# Go celebrate.
#stop("You can't tell me what to do--this is America!")
```

The Cancer file handles the import and cleaning of the cancer data, along with the creation of graphs and the MCMC for the data set. At the end, it produces effective sample size, convergence plots, and does a little bit of model checking using posterior predictive checks.

```
library(mvtnorm) #Generate MV Normals
library(coda) #MCMC datatype and Eff Samp Size
library(compiler) #Bytecode Compiler--speeds up for loops
library(MASS) #For Truehist
library(xtable) #To display Latex for variable selection

if (Sys.info()['sysname'] %in% c("Linux", "Darwin")) {
  setwd("~/Dropbox/sta250/Assignments/HW1/BayesLogit/")
} else {
  stop("You're using a different architecture than I am. setwd() so that the code is in the pwd!")
}

# Read data corresponding to appropriate sim_num:
cancer = read.table("breast_cancer.txt", sep="", header=TRUE)


X = model.matrix(diagnosis ~ ., data=cancer) #Making the design matrix
p = ncol(X)
y = as.numeric(cancer$diagnosis)-1 #Benign=0, Malignant=1;
m = rep(1, length(y))

#####################################################
# Set up the specifications:
beta.0 <- matrix(rep(0,p))
sigma.0 <- diag(p)*1000
niter <- 5E5
burnin = 1E4
print.every = niter/10
retune = 1E2
verbose = TRUE
#####################################################

# Fit the Bayesian model:
source("./helper_functions.R")
Posterior_Draws = bayes.logreg.samples(m, y, X, beta.0, sigma.0, niter, burnin, print.every, retune,
    verbose)

chain = mcmc(Posterior_Draws)
ESS = effectiveSize(chain)
corrs = acf(chain, lag.max=1, plot=FALSE) #Compute lag-1 autocorrelations
diag_acf = numeric(p)
for (i in 1:p) { diag_acf[i] = corrs$acf[2,i,i] } #Take only the non cross-autocorrs.

#Let's thin the draws to reduce the autocorrelation.
chain_thinned = mcmc(Posterior_Draws[seq.int(1, (niter-burnin), by=100), ])
ESS_thinned = effectiveSize(chain_thinned)
corrs_thinned = acf(chain_thinned, lag.max=1, plot=FALSE) #Compute lag-1 autocorrelations
diag_acf_thinned = numeric(p)
```

```
for (i in 1:p) { diag_acf_thinned[i] = corrs_thinned$acf[2,i,i] } #Take only the non cross-autocorrs.

#Plotting the thinned, final chain.
plot(chain_thinned, density=FALSE, ask=FALSE)
Posterior_Quantiles = apply(chain_thinned, 2, quantile, probs=c(.025, .975))
xtable(Posterior_Quantiles, caption="95% Central Credible Interval", align="|ccccccccccc|")

#Making the Posterior Predictive Check graph.
PPC.Graphs(Y, X)
```