ECON5529
Bayesian Theory

Ellis Scharfenaker
Fall 2017, Lecture 5

Intractable Posteriors

- Bayes' Theorem tell us that the posterior distribution is equal to

$$p[\theta \mid x] = \frac{p[x \mid \theta]\, p[\theta]}{p[x]} = (p[x \mid \theta]\, p[\theta]) \Big/ \left(\int p[x \mid \theta]\, p[\theta]\, d\theta \right) \tag{1}$$

- In many cases we have a **closed-form expression** for $p[x \mid \theta]\, p[\theta]$, such as with a natural conjugate model.
- We also saw that it was possible specify the prior on a dense grid of points spanning the range of θ values, and thus the posterior is numerically generated by summing across the discrete values.
- However there are many situations in which neither of these methods will work.
- For example what if our priors about θ in the Binomial model cannot be well represented by a Beta distribution or by any function that yields an analytically solvable posterior function?
- For a single parameter, grid approximation is one approach to addressing such situations, however, for multiparameter models grid approximation becomes forbidding.

In R.

Intractable Posteriors

- In such cases we cannot make proper posterior inference because the posterior defines no standard distribution and solving the integral $\int p[\theta]\, p[x \mid \theta]\, d\theta$ maybe difficult or impossible.
- In these situations it is often possible to *simulate* values that share the same distributional properties as $p[\theta \mid x] = \frac{p[\theta]\, p[x \mid \theta]}{\int p[\theta]\, p[x \mid \theta]\, d\theta}$.
- That is, we can make posterior inferences by using empirical summaries of the simulated values that approximate the true posterior distribution rather than being forced to work with simpler closed form solutions.
- The method we use for simulating posterior distributions is called *Markov Chain Monte Carlo (MCMC).*

In R

Monte Carlo Integration

- Given a function, $f[\theta]$, that is difficult to express or manipulate, but for which we could easily generate samples on an arbitrary support of interest $[a, b]$, Monte Carlo integration allows us to approximate $f[x]$ through simulation. A common quantity of interest is to calculate:

$$I[a, b] = \int_a^b f[\theta]\, g[\theta]\, d\theta$$

- For example if $g[\theta] = \theta$, then $I[a, b]$ simply calculates the mean of θ over the support $[a, b]$.

$$I[a, b] = E[\theta] = \int_a^b f[\theta]\, \theta\, d\theta$$

- If $f[\theta]$ is sufficiently complex so that directly solving this integral is prohibitively challenging, we may be able approximate it through simulation.
- If we have an easy method for producing random θ which are known to be from the correct distribution, then it is possible to use these empirical draws to summarize the unknown integral.
- Instead of analytically calculating the complex integral we could randomly generate n values of θ from $f[\theta]$ and instead calculate:

$$\hat{I}[a, b] = n^{-1} \sum_{i=1}^{n} g[\theta_i]$$

- The idea being that we can replace analytical integration with summation from a large number of simulated values, rejecting values outside of the range of interest, $[a, b]$.
- By the ***strong law of large numbers*** $\hat{I}[a, b]$ will converge with probability one to $I[a, b]$.

Monte Carlo Integration

- Monte Carlo methods are a form of ***stochastic integration*** used to approximate expectations by invoking the law of large numbers.
- In a Bayesian context the difficult function we deal with is the posterior distribution $f[\theta] = p[\theta \mid x]$.
- We can report the posterior with various estimates of some function of the unknown parameter θ with $g[\theta]$ e.g. $g[\theta] = \theta$ is the posterior mean of $p[\theta \mid x]$.
- In general:

$$\mathbb{E}[g[\theta] \mid x] = \int p[\theta \mid x]\, g[\theta]\, d\theta \approx \frac{1}{n} \sum_{i=1}^{n} g[\theta_i]$$

Example

- Let $\theta \sim \mathcal{N}[0, 1]$ such that $f[\theta] = \frac{1}{\sqrt{2\pi}} e^{-\frac{\theta^2}{2}}$. The variance between -2 and 1 is the integral:

$$I[-2, 1] = \int_{-2}^{1} \theta^2 \frac{1}{\sqrt{2\pi}} e^{\frac{-\theta^2}{2}}\, d\theta \approx \frac{1}{n} \Sigma_{i=1}^{n} \theta_i^2$$

```
∫_-2^1 e^2 1/√(2 π) e^(-e^2/2) de // N
```

```
0.468642
```

- This is a difficult integral to solve analytically (requires using polar coordinates). However in R we only need two lines of code.

In R

Stochastic Processes

- Markov chains represent a class of stochastic processes of great interest for the wide spectrum of practical applications.

- A ***stochastic process*** is a set of random variables $\{X[\alpha]\}$ with $\alpha \in A$ an *ordered set*. Usually we shall take A to be the reals or integers and give this ordered set the interpretation of time.
- A stochastic process is said to be ***time-homogeneous*** (or time invariant) if the ordered set $A \in \mathbb{R}$ and the conditional probabilities satisfy

$$p[x[t] \mid x[t-a]] = p[x[s] \mid x[s-a]] \;\forall\; t, s \in \mathbb{R} \text{ and } \forall\; a > 0$$

- That is, if any sequence of consecutive points $(t-a)$ has the same distribution as any other sequence of consecutive points $(s-a)$.
- The case where $A = \mathbb{Z}$ is similar. This has an attractive (physical) interpretation in terms of statistical causality since it says that if the random variable is known exactly at a particular time then the later probability of the variable is specified uniquely. Note the introduction of an "arrow in time" since we require that $a > 0$.
- A stochastic process is called discrete if $A = \mathbb{Z}$. Often we consider only positive integers ($A = \mathbb{N}$) to allow for an “initial condition” at $t = 1$.

Markov Process

- A ***discrete stochastic process*** is called ***Markov*** if the transition probabilities between different values in the *state space* depend only on the random variable's current state, or immediate previous component:

$$p[x_{t+1} \mid x_t, x_{t-1}, x_{t-2}, \ldots, x_1] = p[x_{t+1} \mid x_t] \tag{2}$$

- The Markov property implies:

$$p[x_{t+1}, x_t, \ldots, x_1] = p[x_{t+1} \mid x_t]\, p[x_t \mid x_{t-1}] \ldots p[x_2 \mid x_1]\, p[x_1] \tag{3}$$

- Because this sequence is now exchangeable, for all $t > 1$ we can equivalently write:

$$p[x_1, x_2, \ldots, x_t, x_{t+1}] = p[x_1]\, p[x_2 \mid x_1]\, p[x_3 \mid x_2] \ldots p[x_{t+1} \mid x_t] \tag{4}$$

- This shows that the initial condition probability, $p[x_1]$, and the conditional probability functions $p[x_t \mid x_{t-1}] \;\forall\; t$ are all that are required to describe a Markov process.
- In the normal time-homogeneous Markov case the conditional probabilities $p[x_t \mid x_{t-1}]$ serve (with the initial conditional probability) to describe the complete set of probability functions for the system. They are referred to as the ***transition functions*** (or matrix when X_t has a countable set of values).
- In a Markov process we are essentially interested in how the set of marginal probability functions evolves in time:

$$p[x_{t+1}] = \sum_{\{x_1, \ldots, x_t\}} p[x_1, x_2, \ldots, x_t, x_{t+1}] \tag{5}$$

Markov Chains

- A ***Markov chain*** refers to a sequence of random variables $\{X\} = \{x_0, \cdots\cdots, x_n\}$ generated by a ***Markov process.*** The analysis of Markov processes is particularly simple in the case where the state space, X, is finite.
- Assume that X has n elements, or, equivalently, that we can observe the system in n distinct states.
- The Markov property implies that the probability laws governing the Markov process can be expressed as a $n \times n$ matrix $P = \{p_{ij}\}$ where $p_{ij} = p[i \mid j]$ is interpreted as the probability that a process at state space j moves to state i in a single step (sometimes written $p_{j \to i}$).

- The matrix P must satisfy the conditions: $p_{ij} \geq 0$ and $\Sigma_{i=1}^{n} p_{ij} = 1$, since the system has to move to some state.
- A matrix with these properties is called a ***stochastic matrix***.
- A ***right*** stochastic matrix is a square matrix, with each ***row*** summing to 1, whereas with a ***left*** stochastic matrix each ***column*** sums to 1. In this lecture all Ps refer to a *left* stochastic matrix.

The Transition Kernel

- A Markov chain is defined by a stochastic matrix containing the ***transition probabilities*** (or the ***transition kernel***): $P = p_{ij} = \text{Prob}[j \to i]$.

	"From 1"	"From 2"
"To 1"	p_{11}	p_{12}
"To 2"	p_{21}	p_{22}

- Let $\pi[t] = \{\pi_1[t], \pi_2[t], \ldots, \pi_n[t]\}$ (where $\pi_i[t] = p[x_t = i]$) denote the column vector of the state space probabilities at step t, this is the probability mass function of the system over the n states in period t.
- Then $\pi_i[t+1] = \Sigma_{j=1}^{n} p_{ij} \pi_j[t]$, since if the system is in state i at time $t+1$, it must have gotten there from being in some state j at time t.
- For example, with the 3×3 transition matrix:

	"From 1"	"From 2"	"From 3"
"To 1"	0.2	0.1	0.5
"To 2"	0.4	0.3	0.1
"To 3"	0.4	0.6	0.4

- And initial conditions $\pi[0] = \{0.5, 0.4, 0.1\}$, we can calculate $\pi_i[1]$ for $i = 1, 2, 3$:

$$\begin{aligned}\pi_1[1] &= p_{11}\,\pi_{11}[0] + p_{12}\,\pi_2[0] + p_{13}\,\pi_3[0] \\ &= 0.2\,(0.5) + 0.1\,(0.4) + 0.5\,(0.1) \\ &= 0.19\end{aligned} \tag{6}$$

$$\begin{aligned}\pi_2[1] &= p_{21}\,\pi_1[0] + p_{22}\,\pi_2[0] + p_{23}\,\pi_3[0] \\ &= 0.4\,(0.5) + 0.3\,(0.4) + 0.1\,(0.1) \\ &= 0.33\end{aligned} \tag{7}$$

$$\begin{aligned}\pi_3[1] &= p_{31}\,\pi_1[t] + p_{32}\,\pi_2[t] + p_{33}\,\pi_3[t] \\ &= 0.4\,(0.5) + 0.6\,(0.4) + 0.4\,(0.1) \\ &= 0.48\end{aligned} \tag{8}$$

- Giving us $\pi[1] = \{\pi_1[1], \pi_2[1], \pi_3[1]\} = \{0.19, 0.33, 0.48\}$

The Chapman-Kolmogorov equation

- The probability that the chain has state value i at time $t+1$ is given by the ***Chapman-Kolmogorov equation***, which sums over the probability of being in a particular state (i.e., gives us the marginal probability) at the current step and the transition probability from that state into state i,

$$\begin{aligned}\pi_i[t+1] &= p[x_{t+1} = i] \\ &= \sum_j p[x_{t+1} = i,\, x_t = j] = \sum_j p[x_{t+1} = i \mid x_t = j]\, p[x_t = j] \\ &= \sum_j p[j, i]\, \pi_j[t]\end{aligned} \tag{9}$$

- The Chapman-Kolmogorov equation can be written more compactly by noticing the probability mass function satisfies the *difference equation*:

$$\pi[t+1] = P\, \pi[t] \tag{10}$$

- Successive iteration of the Chapman-Kolmogorov equation describes the evolution of the Markov chain:

$$\pi[t+2] = P\, \pi[t+1] = P\, P\, \pi[t] = P^2\, \pi[t] \tag{11}$$

- In general, after t periods the transition matrix will be P^t.

$$\pi[t] = P^t\, \pi[0] \tag{12}$$

Example

- Continuing with the example above we see

$$\pi[1] = P\, \pi[0] = \begin{pmatrix} 0.2 & 0.1 & 0.5 \\ 0.4 & 0.3 & 0.1 \\ 0.4 & 0.6 & 0.4 \end{pmatrix} \begin{pmatrix} 0.5 \\ 0.4 \\ 0.1 \end{pmatrix} = \begin{pmatrix} 0.19 \\ 0.33 \\ 0.48 \end{pmatrix} \tag{13}$$

$$\pi[2] = P\, \pi[1] = \begin{pmatrix} 0.2 & 0.1 & 0.5 \\ 0.4 & 0.3 & 0.1 \\ 0.4 & 0.6 & 0.4 \end{pmatrix} \begin{pmatrix} 0.19 \\ 0.33 \\ 0.48 \end{pmatrix} = \begin{pmatrix} 0.311 \\ 0.233 \\ 0.466 \end{pmatrix}$$

$$= P^2\, \pi[0] = \begin{pmatrix} 0.2 & 0.1 & 0.5 \\ 0.4 & 0.3 & 0.1 \\ 0.4 & 0.6 & 0.4 \end{pmatrix} \begin{pmatrix} 0.2 & 0.1 & 0.5 \\ 0.4 & 0.3 & 0.1 \\ 0.4 & 0.6 & 0.4 \end{pmatrix} \begin{pmatrix} 0.5 \\ 0.4 \\ 0.1 \end{pmatrix} \tag{14}$$

$$= \begin{pmatrix} 0.28 & 0.35 & 0.31 \\ 0.24 & 0.19 & 0.27 \\ 0.48 & 0.46 & 0.42 \end{pmatrix} \begin{pmatrix} 0.5 \\ 0.4 \\ 0.1 \end{pmatrix} = \begin{pmatrix} 0.311 \\ 0.233 \\ 0.466 \end{pmatrix}$$

Properties of Markov Processes

- A fundamental idea in Markov chains is that state i can be reached from some state j if for $t \geq 1$, $p_{ij}^t > 0$.
- In general if some of the transition functions are zero then it will not be possible to pass from a particular state to another in one time step and these sets of states are ***closed***.
- However it may be possible to do so in more than one step i.e. indirectly. This motivates several properties of some Markov processes.
- A Markov process in which it is possible to pass from any state to another in a finite number of time steps is called ***irreducible***. That is, all states ***communicate*** with each other, as one can always go from any state to any other state (although it may take more than one step).

- If the possible path lengths (number of time-steps) of such a transition has a common factor of 1 then the process is further termed ***aperiodic***. That is, the number of steps required to move between two states is not required to be a multiple of some integer. Put another way, the chain is not forced into some cycle of fixed length between certain states.

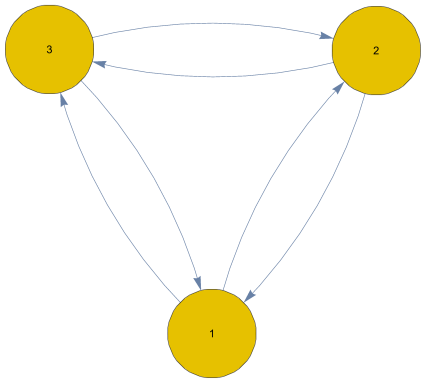
Visualizing a Markov Chain

- One way of visualizing Markov chains is by plotting a graph of the states indicating the communication between states with arrows. For example take a simple 3×3 transition kernel:

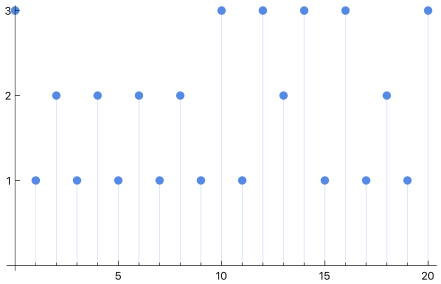
	"From 1"	"From 2"	"From 3"
"To 1"	0	0.5	0.5
"To 2"	0.5	0	0.5
"To 3	0.5	0.5	0

$$P = \begin{pmatrix} 0 & .5 & .5 \\ .5 & 0 & .5 \\ .5 & .5 & 0 \end{pmatrix}$$

- All states communicate so a Markov process defined by this transition kernel is ***irreducible***. This process is also ***aperiodic***. The graph appears as:



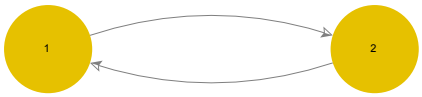
- Simulating this Markov process:



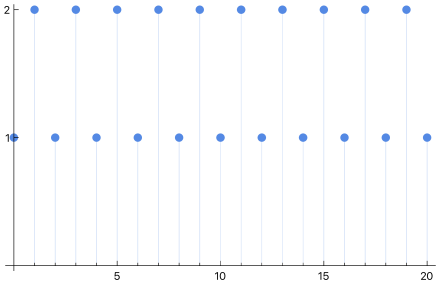
- However, a transition kernel such as:

$$P = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

- Is ***periodic*** and therefore deterministic. The probability of moving to the opposite state is 1. The graph appears as:



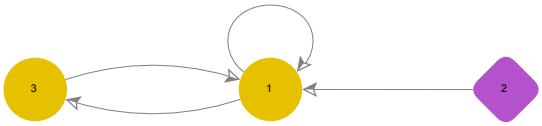
- Simulating this Markov process:



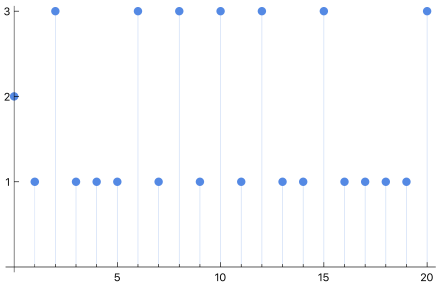
- For the transition kernel:

$$P = \begin{pmatrix} .5 & 1 & 1 \\ 0 & 0 & 0 \\ .5 & 0 & 0 \end{pmatrix}$$

- the corresponding Markov process is ***not irreducible*** because not all states communicate, though it is ***aperiodic***. We can see that entering state 2 from any other state is impossible as the probability is zero. The graph appears as:



- Simulating this Markov process:



- It is also possible that once a system leaves a set of states it can never reach it again. These states are called ***transient***. In the above example, state 2 is transient.

Markov Stationarity

- Often as time increases the probability function of a Markov process will tend to converge. In particular, a time-homogeneous Markov process for which $\pi[t+1] = \pi[t]$ is called ***stationary***.
- A Markov chain may reach a ***stationary distribution*** π^*, where the vector of probabilities of being in any particular given state is independent of the initial condition. The stationary distribution satisfies

$$\pi^* = P\,\pi^* \tag{15}$$

- It is proven in standard texts on Markov processes that if the process is ***irreducible*** and ***aperiodic*** then asymptotically (i.e. as time increases) it converges to a unique ***stationary*** process.

Perron-Frobenius

- This stationary process can be found for the case that there are a finite number of states as the *eigenvector* of the transition matrix with *eigenvalue* one.
- In other words, if P is an *irreducible*, *aperiodic stochastic matrix,* then P is non-negative, and the ***Perron-Frobenius theorem*** tells us that the largest eigenvalue of P is real and corresponds to a non-negative eigenvector.
- Since P is stochastic, the largest eigenvalue must be 1. Since P is irreducible the corresponding eigenvector $\pi^* > 0$. Thus, the solution of the difference equation $\pi[t+1] = P\,\pi[t]$ must converge to π^* *for any initial condition* $\pi[0]$.

Ergodicity

- The probability mass function π^* is called the ***ergodic distribution*** for the matrix P.
- The important idea here is that an irreducible, aperiodic Markov process tends to mix probabilities, so that asymptotically the system "forgets" its initial conditions and converges to the same long-run probabilistic state.

In R

MCMC Principles

- MCMC is essentially a continuous-valued generalization of the discrete Markov chain setup. MCMC produces chains that “wander” around the target posterior distribution, giving full characterization of it.
- MCMC has its (recent) origins in Geman and Geman (1984) who introduced the ***Gibbs sampler*** as a method for obtaining difficult posterior quantities in image reconstruction, though its origins are in statistical physics, see Metropolis (1946) and (1953).
- Given a draw θ_j, one generates a new draw θ_{j+1} from a distribution that may depend on θ_j (but not on earlier draws). The draws are generally serially correlated across j, but eventually their sample distribution function converges to that of the target distribution.
- We have three main goals in generating an MCMC sample from the posterior distribution:
  - 1. The values in the chain must be representative of the posterior distribution. They should not be unduly influenced by the arbitrary initial value of the chain, and they should fully explore the range of the posterior distribution without getting stuck.
  - 2. The chain should be of sufficient size so that estimates are accurate and stable. In particular, the estimates of the central tendency (such as median or mode), and the limits of the 95% HPD, should not be much different if the MCMC analysis is run again (using different seed states for the pseudorandom number generators).
  - 3. The chain should be generated efficiently, with as few steps as possible, so not to exceed our patience or computing power.

Gibbs Sampling

- The Gibbs sampler is the most widely used MCMC technique. All that the Gibbs sampler requires is the specific knowledge about the conditional relationship between variables of interest.

- The basic idea is that if it is possible to express each of the coefficients of interest as conditional on all others, they by cycling through these conditional distributions we approximate the true joint distribution of interest.

The Gibbs Sampling Algorithm

- The Gibbs sampler is a Markov transition kernel created by a series of full conditional distributions that is a Markovian updating scheme based on conditional probability statements.
- The set of full conditional distributions for $\boldsymbol{\theta}$ are denoted $\boldsymbol{\Theta}$ defined by $p[\boldsymbol{\Theta}] = p[\theta_i \mid \boldsymbol{\theta}_{-i}]$ for $i = 1, \ldots, n$. The Gibbs sampler requires a analytically definable full conditional distribution for each coefficient in the $\boldsymbol{\theta}$ vector.
- Let $\boldsymbol{\theta} = \{\theta_1, \theta_2, \ldots, \theta_n\}$. Suppose we know the conditional distributions of all $p\left[\theta_j \mid \theta_k\right] \; \forall \; k \neq j$. In order to obtain samples $\boldsymbol{\theta}^{(i)}$ from the joint distribution $p[\theta_1, \theta_2, \ldots, \theta_n]$ we do the following:
- Initialize $\boldsymbol{\theta}^{(0)} = \left\{\theta_1^{(0)}, \theta_2^{(0)}, \ldots, \theta_n^{(0)}\right\}$ and with $i = 0$ repeatedly:

  Sample $\theta_1^{(i+1)} \sim p\left[\theta_1 \mid \theta_2^{(i)}, \theta_3^{(i)}, \ldots, \theta_n^{(i)}\right]$

  Sample $\theta_2^{(i+1)} \sim p\left[\theta_2 \mid \theta_1^{(i+1)}, \theta_3^{(i)}, \theta_4^{(i)}, \ldots, \theta_n^{(i)}\right]$

  Sample $\theta_3^{(i+1)} \sim p\left[\theta_3 \mid \theta_1^{(i+1)}, \theta_2^{(i+1)}, \theta_4^{(i)}, \ldots, \theta_n^{(i)}\right]$

  ...

  ...

  ...

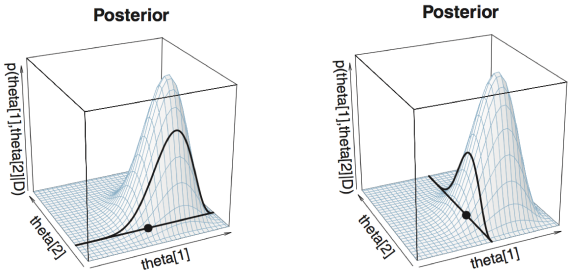
  Sample $\theta_n^{(i+1)} \sim p\left[\theta_n \mid \theta_1^{(i+1)}, \theta_2^{(i+1)}, \ldots, \theta_{n-1}^{(i+1)}\right]$
- If the Gibbs sampler is run sufficiently long, further full cycles of the algorithm produce a complete sampling from the coefficients joint distribution.
- For $\theta = \{\theta_1, \theta_2\}$ initialize $\boldsymbol{\theta}^{(0)} = \left\{\theta_1^{(0)}, \theta_2^{(0)}\right\}$ and with $i = 0$ repeatedly:

  Sample $\theta_1^{(1)} \sim p\left[\theta_1 \mid \theta_2^{(0)}\right]$

  Sample $\theta_2^{(1)} \sim p\left[\theta_2 \mid \theta_1^{(1)}\right]$

  Sample $\theta_1^{(2)} \sim p\left[\theta_1 \mid \theta_2^{(1)}\right]$

  Sample $\theta_1^{(2)} \sim p\left[\theta_1 \mid \theta_2^{(2)}\right]$...



Gibbs Sampling Example: The Drunk Banker

- Suppose a banker is responsible for finding out the probability of default, θ, on a new kind of mortgage. The bank has issued 20 mortgages. The banker in question is given the mortgages and decides to do his calculation while drunk. He accidentally looses one mortgage before observing whether or not it was in default. He does observe that of the 19 mortgages he has seen 14 have defaulted.
- One option is to ignore the missing values and bias the result. In this case our data $X = \{x_1, \ldots, x_{19}\}$, $x_i \in \{0, 1\}$, where $x_i = 1$ means default, is distributed according to the binomial PDF:

$$p[X \mid \theta] = \binom{n}{k} \theta^k (1-\theta)^{n-k}$$

- With $k = \sum_{i=1}^{19} x_i$ and $n = 19$ and θ is the unknown probability of default. The banker however does not want to bias the estimate and knows that the missing value $x_{20} \sim \text{Bin}[x \mid \theta]$. Noting that if he were able to treat x_{20} as known then the distribution of θ (with $n = 20$) would be:

$$p[\theta \mid k, x_{20}] = \binom{n}{k + x_{20}} \theta^{k+x_{20}} (1-\theta)^{n-(k+x_{20})}$$

- Recall that we can write the Binomial distribution as a Beta distribution by adding one to the parameters.

$$p[\theta \mid k, x_{20}] = \text{Beta}\left[k + x_{20}^* + 1, n - \left(k + x_{20}^*\right) + 1\right]$$

- Now the banker reasons that he has two fully specified conditional distributions and two unknowns: θ^* and x_{20}^*:

$$p\left[\theta \mid x_{20}^*\right] \sim \text{Beta}\left[k + x_{20}^* + 1, n - \left(k + x_{20}^*\right) + 1\right]$$

$$p\left[x_{20} \mid X, \theta^*\right] \sim \text{Bernoulli}\left[\theta^*\right]$$

- Thus ,we have everything we need to set up a Gibbs sampler.

In R

Gibbs Sampling Posterior Check

- Typically, we consider some fraction of the first iterations as “burn-in” that is to be discarded. That is, these are the pre-convergence values of the Markov chain wandering away from its initial condition towards its limiting distribution.
- We can plot the ***traceplot*** and histogram with the HPD. A traceplot shows the time-series of the Markov chain path and give a good visual diagnostic for the convergence of the Markov chain to its stationary distribution.
- To see the convergence of the Gibbs sampler look the iterated values of θ.
- The posterior mean of the “burned-in” sample is 0.71 with variance 0.009 while the biased estimate $\frac{k}{n} = \frac{14}{19} = 0.737$. The posterior mean of the missing mortgage is 0.71 meaning it is more likely to be a one than a zero, although the posterior variance of 0.2 suggests some caution in this claim. Nevertheless, our estimate of θ remains unbiased.

Accept-Reject Algorithm

- The accept-reject method is an indirect method in which we generate a candidate random variable and only accept it subject to passing a test. Accept–Reject methods only require us to know the functional form of a density of interest (called the *target density*) up to a multiplicative constant.
- This is ideal for Bayesian analysis as the posterior $p[\theta \mid x] \propto p[x \mid \theta]\, p[\theta]$ is easily ascertainable while the multiplicative normalizing constant $\int p[x \mid \theta]\, p[\theta]\, d\theta$ is often intractable. For the accept-reject method, all we need is a *candidate* distribution over the same support.
- If X is from an unknown density function $X \sim f$ we can simulate it by generating Y from a known candidate density function $Y \sim g$ and then generating from the uniform distribution $U \sim U_{[0,1]}$ and if $U \leq \frac{1}{M}\frac{f[Y]}{g[Y]}$ set $X = Y$, where M is a constant with $\frac{f(x)}{g(x)} \leq M$ for all x. If the inequality is not satisfied discard Y and U and start again.
  - 1. Generate $Y \sim g,\ U \sim U_{[0,1]}$
  - 2. Accept $X = Y$ if $U \leq \frac{f[x]}{M\, g[y]}$
  - 3. Return to step 1 otherwise
- In R, where *randg* is a function that delivers generations from the density *g* like *rnorm* the following general algorithm produces a single generation *y* from *f*.

```
u <- runif(1) * M
y <- randg(1)
while(u > f(y) / g(y)){u = runif(1) * M + y = randg(1)}
```

- **Example:** Simulate Beta[2.7, 6.3] random variables using the Accept-Reject method using as the candidate distribution the uniform $U_{[0,1]}$ distribution. The upper bound M is then the maximum of the beta density obtained by the optimize function.
- Since the candidate density g is equal to one, the proposed value Y is accepted if $M \times U < f[Y]$, that is, if $M \times U$ is under the beta density f at that realization. Note that generating $U \sim U_{[0,1]}$ and multiplying by M is equivalent to generating $U \sim U_{[0,M]}$.

Metropolis-Hastings Algorithm

- The full set of conditional distributions for the Gibbs sampler can frequently be easy to specify. However, when the complete conditionals for the $\boldsymbol{\theta}$ parameters do not result in an easily specified closed form conditional distribution we need a more general method for posterior simulation.
- The idea behind the Metropolis-Hastings algorithm is to produce a multidimensional candidate value of the posterior all at once, rather than serially throughout the $j = 1, \ldots, J$.
- The basic idea is that we use some "local" accept-reject mechanism whereby at each step we propose a candidate sample θ_{proposed} from a distribution over the same support that depends on the current state of the chain θ_{current}. This distribution is called the *jump* or *proposal* distribution $q\left[\theta_{\text{proposed}} \mid \theta_{\text{current}}\right]$ with the condition that the reverse function $q\left[\theta_{\text{current}} \mid \theta_{\text{proposed}}\right]$ is also determined.
- We define the accept-reject mechanism as the *acceptance ratio:*

$$\alpha\left[\theta_{\text{proposed}},\, \theta_{\text{current}}\right] = \frac{\left(p\left[\theta_{\text{proposed}}\right] q\left[\theta_{\text{current}} \mid \theta_{\text{proposed}}\right]\right)}{\left(p\left[\theta_{\text{current}}\right] q\left[\theta_{\text{proposed}} \mid \theta_{\text{current}}\right]\right)}$$

- At time t = current, the decision that either produces the $t+1$ = next point in the chain or keeps the chain in the same place is *probabilistically* determined according to:

$$\theta_{\text{next}} = \begin{cases} \theta_{\text{proposed}} & \text{with probability} \quad \min\left\{\alpha\left[\theta_{\text{proposed}},\, \theta_{\text{current}}\right],\, 1\right\} \\ \theta_{\text{current}} & \text{with probability} \quad 1 - \min\left\{\alpha\left[\theta_{\text{proposed}},\, \theta_{\text{current}}\right],\, 1\right\} \end{cases}$$

- That is, if the target distribution probability is greater at the proposed position than at our current position, then we definitely accept the proposed move ($\theta_{\text{next}} = \theta_{\text{proposed}}$ with probability 1) and if the target position is less at the proposed position than at our current position, we accept the move probabilistically ($\theta_{\text{next}} = \theta_{\text{current}}$ with probability $1 - \alpha\left[\theta_{\text{proposed}},\, \theta_{\text{current}}\right]$).
- Hence, unlike the Gibbs sampler the Metropolis-Hastings algorithm does not necessitate movement on every iteration.
- The idea is we are considering adding a sample value $\boldsymbol{\theta}'$ to our simulated posterior distribution. If our current sample $\boldsymbol{\theta}^{(t)}$ was close to our posterior distribution we would like our next candidate $\boldsymbol{\theta}'$ to be close to $\boldsymbol{\theta}^{(t)}$. Should we include $\boldsymbol{\theta}'$? Easy, if $p[\boldsymbol{\theta}'] > p\left[\boldsymbol{\theta}^{(t)}\right]$, then we want more sample $\boldsymbol{\theta}'$ *s* in the set than $\boldsymbol{\theta}^{(t)}$ *s* so we accept with probability 1. If $p[\boldsymbol{\theta}'] < p\left[\boldsymbol{\theta}^{(t)}\right]$, we shouldn't necessarily include $\boldsymbol{\theta}'$.
- The Metropolis-Hastings algorithm can be stated as:
- Starting with $\boldsymbol{\theta}^{(0)} = \left\{\theta_1^{(0)},\, \theta_2^{(0)},\, \ldots,\, \theta_n^{(0)}\right\}$ iterate for $t = 1, 2, \ldots$
  - 1. Generate $\boldsymbol{\theta}' \sim q_t\left[\boldsymbol{\theta}' \mid \boldsymbol{\theta}^{(t)}\right]$
  - 2. Compute $\alpha\left[\boldsymbol{\theta}' \mid \boldsymbol{\theta}^{(t)}\right] = \min\left\{1,\, \frac{p[\boldsymbol{\theta}']\, q_t\left[\boldsymbol{\theta}^{(t)} \mid \boldsymbol{\theta}'\right]}{p\left[\boldsymbol{\theta}^{(t)}\right] q_t\left[\boldsymbol{\theta}' \mid \boldsymbol{\theta}^{(t)}\right]}\right\}$
  - 3. Take $\boldsymbol{\theta}^{(t+1)} = \begin{cases} \boldsymbol{\theta}' & \text{with probability} \quad \alpha\left[\boldsymbol{\theta}' \mid \boldsymbol{\theta}^{(t)}\right] \\ \boldsymbol{\theta}^{(t)} & \text{with probability} \quad 1 - \alpha\left[\boldsymbol{\theta}' \mid \boldsymbol{\theta}^{(t)}\right] \end{cases}$

Example

- We flip a coin $N = 100$ times and observe $x = 39$ heads. We use a Bernoulli likelihood function, $p[k,\, n \mid \theta] = \theta^k (1-\theta)^{(n-k)}$. We start with a prior $p[\theta] = \text{Beta}[\theta \mid \alpha,\, \beta]$. For the proposed jump in the Metropolis algorithm, we will use a normal distribution centered at zero with standard deviation σ. We denote the proposed jump as $\Delta\theta \sim \text{Normal}[0, \sigma]$**.** The proposed jump is usually close to the current position, because the mean jump is zero, but the proposed jump can be positive or negative, with larger magnitudes less likely than smaller magnitudes. Denote the current parameter value as θ_{cur} and the proposed parameter value as $\theta_{\text{pro}} = \theta_{\text{cur}} + \Delta\theta$. The Metropolis algorithm then proceeds as follows:
- Start at an arbitrary initial value of θ (in the valid range). This is the current value, denoted θ_{cur}. Then:
  - 1) Randomly generate a proposed jump, $\Delta\theta \sim \text{Normal}[0, \sigma]$ and denote the proposed value of the parameter as $\theta_{\text{pro}} = \theta_{\text{cur}} + \Delta\theta$.
  - 2) Compute the probability of moving to the proposed value as:

$$p_{\text{move}} = \min\left\{\frac{p[D \mid \theta_{\text{pro}}]\, p[\theta_{\text{pro}}]}{p[D \mid \theta_{\text{cur}}]\, p[\theta_{\text{cur}}]},\, 1\right\}$$

- 3) Accept the proposed parameter value if a random value sampled from a [0,1] uniform distribution is less than p_{move}, otherwise reject the proposed parameter value and tally the current value again.

In R

JAGS

- JAGS (just another Gibbs sampler) is a system that automatically builds Markov chain Monte Carto samplers for complex models. JAGS succeeded the pioneering system named BUGS (Bayesian inference using Gibbs sampling).
- Download JAGS 4. (http://mcmc-jags.sourceforge.net) and "rjags." Also you will need a C++ compiler such as Xcode.
- JAGS takes a user's description of a hierarchical model for data, and returns an MCMC sample of the posterior distribution.
- To run Jags in R you will need to install the package "rjags". The Jags script is written in a text editor and will need to be saved in your working library for rjags to call it.

JAGS Example

- Following the simplest example lets make posterior inferences about binomial data.

```
x = rbinom(100, 1, .3)
N = length(x)
data = list( x = x,  N = N)
```

- The data list is sent to JAGS and tells JAGS the names and values of variables that define the data. Importantly, the names of the components must match variable names in the JAGS model specification.
- Using a text editor the JAGS model is specified as

```
model{
 for( i in 1 : N ){
  y[i] ~ dbern( theta )
 }
theta ~dbeta(1, 1) }
```

- Because that relation is true for all instances of y_i, the statement is put inside a for-loop. The for-loop is merely a shortcut for telling JAGS to copy the statement for every value of *i* from 1 to N.
- JAGS will set initial values for the chains automatically. However, the efficiency of the MCMC process can sometimes be improved if we intelligently provide reasonable starting values to JAGS.
- In general, a useful choice for initial values of the parameters is their maximum likelihood estimate. In this example the MLE of θ is $\frac{k}{n}$.

```
thetaInit = sum(y) / length(y)
initsList = list( theta = thetaInit )
```

- To run the JAGS model we use the jags.model function where the first argument (*file*) is the name of the file in which the model specification is stored, the *initis* argument specifies the list of initial values that was created (omit and JAGS will supply its own initial values), and the last two

arguments specify the number of chains and the number of steps to take for burning in the samplers. These arguments default to 1 chain and 1000 adaptation steps if they are not specified by the user.

```
jagsModel =
 jags.model ( file = "Binomial.jags" , data = data , inits = initsList ,
           n.chains = 3 , n.adapt = 500 )
```

- After burn-in, we tell JAGS to generate MCMC samples that we will actually use to represent the posterior distribution. The chains of the parameter values are retried using the *coda* package and the function that generates the MCMC samples is called *coda.samples* in the *rjags* package.

```
codaSamples =
 coda.samples (jagsModel, variable.names = c ("theta"), n.iter = 1000)
summary (codaSamples)
plot (codaSamples)
```

- The first argument is the JAGS object, the *n.iter* argument is the number of iterations, or steps, taken by each chain and the *variable.names* argument specifies which parameters will have their values recorded during the MCMC walk. JAGS will only record the trajectories of parameters that you explicitly tell it to.
- We can explore the output with many different packages. One particuarly nice one is *ggmcmc.*

```
library (ggmcmc)
mcmc.out < - ggs (codaSamples)
ggs_histogram (mcmc.out)
ggs_density (mcmc.out)
ggs_traceplot (mcmc.out)
```

Regression in JAGS

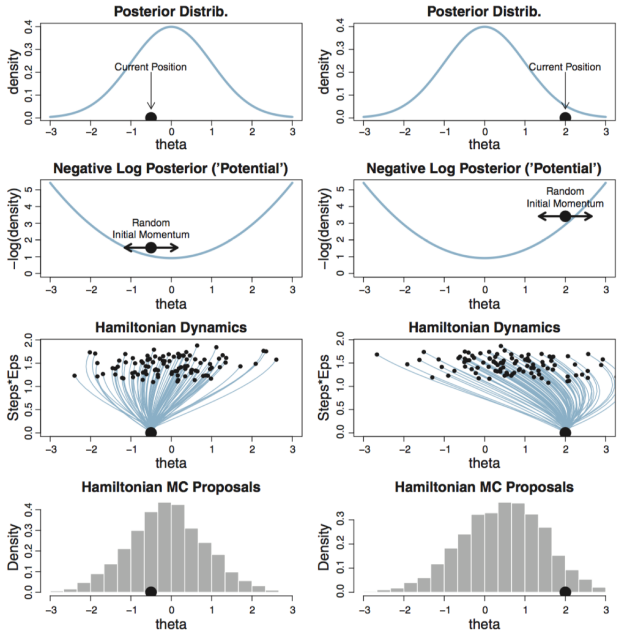
- To run a simple linear regression in JAGS we can specify the error model as

```
model {
  for (i in 1 : N) {
     y[i] ~ dnorm (y.hat[i], tau)
       y.hat[i] < - a + b * x[i]
    }
    a ~ dnorm (0, .0001)
    b ~ dnorm (0, .0001)
   tau < - pow (sigma, - 2)
   sigma ~ dunif (0, 100)
 }
```

- JAGS uses the precision $\tau = \frac{1}{\sigma^2}$ for normal distributions.
- We can compare these results to the built in LM regression function

Stan and RStan

- Stan was developed to support full Bayesian inference and is an imperative probabilistic programming language, like C. For continuous parameters, Stan uses Hamiltonian Monte Carlo (HMC) sampling which is a form of Markov chain Monte Carlo (MCMC) sampling that uses the gradient of the log probability function and treats the vector of parameters θ as the position of a fictional particle.
- Each iteration generates a random momentum and simulates the path of the particle with potential energy determined the (negative) log probability function. Hamiltonian decomposition shows that the gradient of this potential determines change in momentum and the momentum determines the change in position.
- These continuous changes over time are approximated using the leapfrog algorithm, which breaks the time into discrete steps which are easily simulated. A Metropolis reject step is then applied to correct for any simulation error and ensure detailed balance of the resulting Markov chain transitions.



- For details see Kruschke (2015) Ch. 14.
- **Example:** Take a model that estimates the bias of a coin. The data are flips of a coin, $x_i \in \{0, 1\}$, described by a Bernoulli distribution, $x_i \sim \text{Bern}[\theta]$, with a beta prior, $\theta \sim \text{Beta}[\theta \mid \alpha, \beta]$**.** The model for Stan with RStan is specified as a string in R. The model specification begins with explicit declarations of which variables are data and which variables are parameters, in separately marked blocks before the model statement. Each variable declaration also states what numeric type the variable is, and any restrictions on its domain. For example:

```
int < lower = 0 > N ;
```

- means that N is an integer value that has a lower bound of zero. The complete model specification, enclosed as a string in R (comments follow //):

```
modelString = "
```

```
 data {
int<lower=0> N;
int x[N]; // x is a length-N vector of integers
}

parameters {
 real<lower=0, upper=1> theta;
}
   model {
 theta ~ beta(1, 1);
 x ~ bernoulli(theta);
}
"
```

- Notice each block ends with an explicit end-of-command marker ";" which is the syntax used in C++, the underlying language used by Stan.
- Also, if no prior is specified then a uniform density is automatically assigned to the parameters. If you wanted to add an explicit prior to the model do so in the *model* block.
- RStan is the R interface for Stan and will need to be installed. Instructions are here: (https://github.com/stan-dev/rstan/wiki/RStan-Getting-Started)
- With the model specified as discussed above, the next step is to translate the model into C++ code and compile the C++ code into an executable dynamic shared object (DSO) since it will need a compiler such as Xcode. The command in RStan for doing this is stan_model. Before it is called, however, the RStan library must be loaded into R:

```
library(rstan)
```

- Now lets create some data and run the model

```
x <- rbinom(50, 1, .1)
N <- length(x)
z <- sum(x)
dataList = list(x = x, N = N)
stanFit = stan(model_code = modelString, data = dataList,
         chains = 3, iter = 1000, warmup = 200, thin = 1)
```

- At this point Stan is figuring out the gradient functions for the Hamiltonian dynamics. This may take a second.
- The arguments of the sampling command start with telling Stan what DSO to use. *warmup* is the number of initial samples to discard as "burn-in", *iter* is the total number of steps per chain, including *warmup* steps in each chain. *Thinning* merely marks some steps as not to be used; thinning does not increase the number of steps taken. Thinning can be used if the MCMC diagnostics suggest "bad" mixing.

- Thus, the total number of steps that Stan takes is *chains* × *iter*. Of those steps, the ones actually used as representative have a total count of $\frac{chains \times (iter - warmup)}{thin}$.
- Stan will randomize initial conditions, but the user can specify these as an argument

```
init = c(ic_1, ..., ic_n)
```

- Where n=# of chains.
- Look at the summary of the Stan output:

```
print(fit, digits_summary = 3)
```

- 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.
- We can extract the data using any of the following commands:

```
samplesAll <- as.data.frame(extract(stanFit))

# return a list of arrays

la <-extract(stanFit, permuted = TRUE)
theta <-la$theta

#return an array of three dimensions : iterations, chains, parameters

a <-extract(stanFit, permuted = FALSE)

### # use S3 functions as.array (or as.matrix) on stanfit objects

a2 <-as.array(fit)
m <-as.matrix(fit)
```

- The stanFit object can be explored in many ways for posterior inference. The package *coda* and *ggmcmc* allow quick plots for MCMC analysis. If you use the *stan* function to run the model to use these packages we need to convert the stan object into a MCMC object.

```
library(coda)
library(ggmcmc)
S <- ggs(stanFit)
```

- To see full details go to (http://xavier-fim.net/packages/ggmcmc)
- Let's look at out posterior density

```
ggs_density(S)
ggs_histogram(S)
ggs_caterpillar(S)
```

- The `ci(S)` function that calculates the credible intervals in the caterpillar plots can also be used outside the graphical display.

MCMC Diagnostics

- From our theory of Markov chains, we expect our chains to eventually converge to the stationary distribution, which is also our target distribution. How do we know whether our chain has actually converged?
- One way to see if our chain has converged is to use visual inspection to see how well our chain is mixing, or moving around the parameter space.
- If our chain is taking a long time to move around the parameter space, then it will take longer to converge.

Traceplots

- A *traceplot* is a plot of the iteration number against the value of the draw of the parameter at each iteration.
- In general we look for a plot of the Markov chains that shows a tight random scatter around a mean value that does not wander. The running mean plot should be relatively stable.

```
ggs_traceplot(S)
ggs_running(S)
```

- We can also see how the posterior density looks throughout the sampling process by examining it at varous steps:

```
ggs_compare_partial(S)
```

Autocorrelation

- Another way to assess convergence is to assess the autocorrelations between the draws of our Markov chain.

```
ggs_autocorrelation(S)
```

Geweke statistic

- The Geweke diagnostic takes two nonoverlapping parts (usually the first 0.1 and last 0.5 proportions) of the Markov chain and compares the means of both parts, using a difference of means test to see if the two parts of the chain are from the same distribution (null hypothesis).

```
ggs_geweke(S)
```

- The test statistic is a standard Z-score with the standard errors adjusted for autocorrelation. By default, the area between -2 and 2 is shadowed for a quicker inspection of problematic chains. That is, we want to see the score (dot) within the shadowed area.