Simulation Methods
Statistics 650/750
Week 12 Tuesday

Christopher Genovese and Alex Reinhart

14 Nov 2017

Announcements

- Feedback and challenge 1 submission
- Updated Wordcloud bounding box script

  word1 word2 @ 20px word3 @ 30px word4
- RCpp bug and its implications
- Plan for next time: Neural Nets and Deep Learning

Motivation and Overview

From its early use on the *Manhattan project* by Stanislaw Ulam and John Von Neumann (and others), computer simulations have proved an invaluable for the study of complex systems. The Manhattan Project code name for this approach was **Monte Carlo**, after the casino, and the name stuck.

Simulations let us setup a system as we like and in a way where we know the true, often latent, configuration. This let's us understand the evolution of the system and look at results of analyses that are difficult to study mathematically.

During a typical simulation, we evolve the system in discrete steps. At each step, one or more random inputs is generated and propagated into the system dynamics, leading to an update the system's state.

Cases Common in Statistics

Simulating Systems: Straight Monte Carlo

Generate "data" from a specified system either to understand the system's evolution or to study our analysis of the system.

Common *use cases* include:

1. We have a statistical procedure that works under specified assumptions. We would like to show that our procedure – as implemented in practice – achieves whatever theoretical guarantees we have established.

   We generate data sets under a variety of configurations and evaluate our procedure, compiling aggregate measures of results.

2. Our data is generated from a complex system that is the target of our study. We would like to understand both the evolution of this system and the performance of statistical tools. We need to compare the simulated "results" to the results we get with real data.

   We create a version of the system in the computer and allow the system to run and evolve, using pseudo-random numbers to generate the random inputs to the system.

Importance Sampling

We want to draw from a probability density function $\pi()$ that we only know up to an *unknown normalizing constant* c. (That is, we can compute $p(x) = c\pi(x)$

Suppose we can find a probability density g on $\mathcal{H}$ that approximates the density π in some sense. Let $\Theta_1, \Theta_2, \ldots$ be a *random sample drawn from* g.

Then for any suitable function h, we can estimate $\mathbb{E}_\pi h(X)$ using **Importance Sampling**.

This involves re-weighting our sample from g with cleverly chosen weights:

$$\begin{aligned}\frac{1}{n}\sum_{k=1}^{n} h(\Theta_i)\frac{p(\Theta_k)}{g(\Theta_k)} \to \mathbb{E}_g h(\Theta)\frac{p(\Theta)}{g(\Theta)} \\ &= \int h(x)\, p(x)\, \mathrm{d}x \\ &= c\int h(x)\, \pi(x)\, \mathrm{d}x \\ &= c\,\mathbb{E}_\pi h(\Theta).\end{aligned}$$

For instance, taking $h \equiv 1$ gives us a way to estimate c.

Much research has gone into Importance Sampling, but the short story is that in general, it is difficult to get right. One reason is that the tails of g must be at least as thick as the tails of π if the estimate is to be stable. It can be useful, however, in some cases.

The Bootstrap

Given a data set $Y_1, \ldots, Y_n$ drawn from some distribution F, we have an estimator for some functional of that distribution $\theta(F)$.

The **Bootstrap** repeatedly resamples from the *empirical distribution* $\widehat{F}$ and uses the variance of the estimators' values to assess uncertainty in the estimator, e.g.,

$$\frac{1}{B}\sum_{b=1}^{B}\left(\theta_b^* - \widehat{\theta}(Y)\right)^2.$$

The Bootstrap enables us to study the performance of complicated estimators for which analytical results are difficult or infeasible to derive.

The basic **bootstrap principle**: the variation of F^* around $\widehat{F}$ mimics the variation of $\widehat{F}$ around F.

The bootstrap is a powerful technique with strong theoretical support, and it can enable analyses that are hard to do other ways. For instance, it is helpful with non-standard estimators and for providing a check on (e.g., parametric) methods that require stronger assumptions. But, *be warned*, outside of basic cases, it does require some mathematical work to show that the method is valid for any particular problem.

Variants:

- Parametric bootstrap – draw from a parametric model $F_{\widehat{\theta}}$
- Subsampling – varying the size of the re-sample from the data
- Balanced resampling – computational techniques to get more efficiency

- Corrected bootstrap confidence intervals –
- Double bootstrap –

Key (early) sources:

- Efron (1979) "Bootstrap methods: another look at the jacknife," *Annals of Statistics*
- Efron (1979) "Computers and the theory of statistics: thinking the unthinkable" *Siam Review*
- Diaconis and Efron (1983) "Computer intensive methods in statistic," *Scientific American.*
- Efron and Tibshirani (1993) *Introduction to the Bootstrap*
- Davison and Hinkley (1997) *Bootstrap Methods*

Situations to watch out for:

- Dependent data with resampling that does not properly reflect the dependence structure (e.g., resampling as if data independent).
- Resampling in a way that does not match the original sampling process.
- Statistics (e.g., maximum) that are very sensitive to local features of the data.

Packages: boot (see guide) in R, scikits.bootstrap in python, Weka for the JVM

Markov Chain Monte Carlo

The Problem: Consider a statistical model where the data $\mathbf{Y}$ are drawn from one of the distributions in the collection $\{f_\theta : \theta \in \mathcal{H}\}$. Each f_θ is a probability distribution indexed by a *parameter* θ from some *parameter space* $\mathcal{H}$.

In Bayesian inference, we use probability as a calculus of uncertainty. We put a *prior distribution* on the unknown parameter, treating it as a random variable Θ. The *posterior distribution*, the conditional distribution of Θ given $\mathbf{Y}$, updates our uncertainties about the parameter and contains all the information we need for inference.

Suppose $f_\theta(y)$ is the conditional density of $Y \mid \Theta \operatorname{near} \theta$, and the prior p is the marginal distribution of Θ, which I will assume is a density. (Here, we write $X \operatorname{near} x$ to mean $X \in [x, x + \mathrm{d}x)$.)

We want to **update our uncertainty** for Θ in light of the observed data Y.

Our uncertainty can be expressed by the *posterior distribution* π. For a set of possible values A, we have

$$\begin{aligned}\pi(A) &= \mathbb{P}\{\Theta \in A \mid Y \operatorname{near} y\} \\ &= \frac{\int_{\theta \in A} f_\theta(y) p(\theta)\, \mathrm{d}\theta}{\int_{\theta \in \mathcal{H}} f_\theta(y) p(\theta)\, \mathrm{d}\theta}.\end{aligned}$$

As a probability density function, this yields:

$$\pi(\theta) = \frac{f_\theta(y) p(\theta)}{\int_\psi f_\psi(y) p(\psi)\, \mathrm{d}\psi}. \tag{1}$$

This is straightforward to express but in general is difficult to calculate because the normalizing constant in the numerator is hard to compute, especially for high-dimensional parameters.

But simulation gives us a possible solution...

The Idea: Create a Markov chain with limiting distribution π. Run the chain and then (after a while to achieve equilibrium), read off the values as a sample from π. From that we can estimate the distribution or any functional thereof.

Issues:

- If we can't compute π, how do we make a chain that converges to it?
- What conditions on the chain do we need to make this work?
- When is the equilibrium reached to sufficient approximation?
- How accurate are the approximations derived from the chain, given especially that the samples are dependent?
- And wait, this will not usually be on a countable state space, does what we know still work?

A Pseudo-Primer on Pseudo-Random Numbers

Random inputs to a simulation are desirable but hard to acquire. Instead, we generate *pseudo-random numbers*: a deterministic sequence of numbers that appears similar to a random sequence.

A Pseudo-Random Number Generator (PRNG) generates its sequence through repeated transformations of its internal state. One specifies a *seed* that entirely determines the sequence, which makes the sequence reproducible for checking results.

PRNGs typically produce Uniform<0,1> variates, which can then be transformed to produce a sample from a target distribution (see below).

A good PRNG will pass a variety of statistical tests that measure apparent "randomness" in the sequence. These can include range tests, run tests on the generated bits, chi-squared tests of probability in different bints, predictability of some bits from others or of numbers from other numbers or of state from the sequence of numbers, and so forth.

Beware: it is very easy to write a bad PRNG, and bad ones are still in use even in popular software systems. Don't try to create your own algorithm unless you really know what you are doing. There are open-source test suites for testing an implementation of an algorithm.

Generating Non-Uniform Random Numbers (Exact Sampling)

- Inverse CDF Sampling

  $Y = F^{-1}(U)$. Why does this work?

- Rejection Method

  Want to sample from density f. Find density g such that $cg \geq f$ for some $c \geq 1$. (This g is said to be an *envelope* for f.)

  Repeat:

  1. Draw X from g.
  2. Draw Y from Uniform<0, z> where $z = c\frac{g(X)}{f(X)}$.

  Until $Y \leq 1$.

  Return the generated X.

  The expected number of generated pairs is c, which we thus want as close to 1 as possible.

- Acceptance-Complement Method

  Suppose $f = f_1 + f_2$, where we know how to sample from f_2 (properly normalized) and $f \leq g$ for a density g from which we can also sample.

  1. Draw X from g.

2. Draw U from Uniform<0,1>
3. If $U > f_1(X)/g(X)$, set X to be a draw from $f_2/\int f_2$.
4. Return X.

Example: $f_1 = \min(f, g)$ and $f_2 = (f - g)_+$. If we can sample from g and f_2, the method applies.

- MCMC-Enhanced Exact Sampling

  Suppose we can find an envelope for the rejection method but c is large. This would lead to an unhappily large number of rejections on average. But we can still get exact sampling by applying the rejection method once to get an X_0 and then running an MCMC from that starting point, which would by definition be in its equilibrium distribution!

See Devroye's book *Non-Uniform Random Variate Generation* for all the gritty details.

Good general purpose PRNGs for Monte Carlo simulations:

- Mersenne Twister (and refinement SIMD-oriented Fast Mersenne Twister)

  Long period, generally fast, potential zero states early
- WELL (Well equidistributed long-period linear)
- Xorshift (better with nonlinear step, e.g., xorshift+ and xorshift*)

Markov Chain Monte Carlo (MCMC)

A Brief Review/Overview of Markov Chains

Definitions

Heuristic Definition 1. A stochastic process has the **Markov property** if the future of the process is conditionally independent of the past given the present state.

Heuristic Definition 2. A **Markov chain** is a stochastic process $X = (X_n)_{n\geq 0}$ with the Markov property. We assume that the X_n have values in some common space $\mathcal{S}$, which is called the **state space** of the chain.

Today, we will put a little meat on the bones of these heuristic definitions, but we will still skirt all the technical details.

To describe a Markov chain, we need to specify

- The state space $\mathcal{S}$.
- The **initial distribution,** which is just the probability distribution of the initial state X_0.
- The probabilities with which a chain makes transitions from state to state. This is most generally given by the **transition kernel**, which in some cases can be represented by a matrix

For a state $x \in \mathcal{S}$ and a suitable set of states $A \subset \mathcal{S}$, we define the **transition kernel** $P(x, A)$ to be the conditional probability of the process moving into A given that it starts at s. When A is a singleton set, like $A = \{y\}$, we write $P(x, y)$ for $P(x, \{y\})$.

A Markov chain is **time-homogeneous** if the transition kernel does not change over time as the process evolves.

A time-homogeneous, countable-state Markov Chain with initial distribution μ and transition Suppose $X = (X_n)_{n\geq 0}$ is a time-homogeneous Markov chain with a countable state space. If X has initial distribution μ and transition kernel P, then

$$\mathbb{P}\{X_0 = s_0, \ldots, X_n = s_n\} = \mu(s_0)P(s_0, s_1)\cdots P(s_{n-1}, s_n).$$

This can generalized directly to uncountable state spaces.

The definition above embodies the Markov property, which can also be given in conditional form as

$$\mathbb{P}_\mu\{X_n = s_n \mid X_{n-1} = s_{n-1}, \ldots, X_0 = s_0\} = P(s_{n-1}, s_n)$$

The Punchline

We are primarily interested in Markov chains that satisfy some specific properties for which it is said to be **ergodic**.

Under slightly stronger conditions, a Markov chain is **reversible**: it looks probabilistically the same in forward and reverse time. Such a chain satisfies *detailed balance*:

$$\pi(s)P(s, s') = \pi(s')P(s', s).$$

That is, the rate of flow between any two states is the same in both directions.

An ergodic chain has a stationary and limiting distribution π. If we start the chain with that distribution it stays in that distribution. For any other initial distribution, the probability of finding the chain in that state converges to that given by π.

So, if we run the chain long enough, the random variables X_n start to look more and more like *independent samples from* π.

Markov Chain Monte Carlo (MCMC) is about constructing reversible, ergodic Markov chains whose limiting distributions we want to simulate. The key to the methods is that we can do this while only being able to compute the limiting distribution *up to an unknown constant*.

Important Properties That We Will Skip Here

- Communicating Classes

  The state space can be partitioned into sets called *communicating classes* within which the chain can move between any pair of states (in both directions) with positive probabilities.

  If there is only one communicating class, a chain is said to be *irreducible*. This is a key condition to have an ergotic chain.

- Positive Recurrence

  A state is *positive recurrent* if the expected time of return to that state is finite. This ensures that each state is visited infinitely often over time. It is a key condition to ensure an ergotic chain.

- Periodicity

  The periodicity of a state is the gcd of the times in which it is possible to return to a state. A state with period 1 is *aperiodic*. If the chain has period above 1, it cannot reach true equilibrium, although the chain taken every d steps (for period d chain) can. Hence, aperiodicity is a necessary condition for a chain to be ergodic.

- A subset of states is *absorbing* if the chain never leaves once it lands within that subset.

- State Decomposition

  The state space can be decomposed into a disjoint union of absorbing, communicating classes plus a union of non-absorbing communicating classes.

  This is the key to full analysis of a Markov chain, which is beyond our purview here.

Common Markov Chain Monte Carlo Samplers

Gibbs Samplers

The Gibbs' sampler is an efficient MCMC strategy where at each step, we sample from the **conditional posterior distribution of one parameter given all the other parameters**. These conditional distributions are called the *complete conditionals*, and though they can be hard to derive, when we can compute these, we get a series of univariate draws. By selecting parameters at each step, we ensure that all the parameters are included.

Example: The Bivariate Normal-Normal Model

Consider the following statistical model. Data (X, Y) is drawn from a bivariate Normal distribution with mean vector (Θ_1, Θ_2), where $\mathsf{Var}(X) = \mathsf{Var}(Y) = 1$ and $(X, Y) = \rho$. We put independent Normal$(0, 1)$ priors on Θ_1, Θ_2.

Then, the *complete conditionals* are

1. the conditional distribution of Θ_1 given Θ_2, X, Y; and
2. the conditional distribution of Θ_2 given Θ_1, X, Y.

Because X, Y, Θ_1, Θ_2 are jointly Normal, we can calculate these distributions directly. They will be Normal distributions.

The **Gibbs sampler** works by successively sampling from these complete conditionals, either alternating between them or, better yet, choosing one at random at each step.

Notice that when we sample from the complete conditional of Θ_1, say, we think of that as a step in the chain from (θ_1, θ_2) to (θ_1', θ_2). That is, only the sampled coordinate changes.

The resulting Markov chain has as equilibrium distribution the posterior distribution of Θ_1, Θ_2. (That is, the conditional distribution of Θ_1, Θ_2 given X, Y.)

We run the sampler for a long time and then pull off samples from the chain, treating them like a draw from this posterior distribution. We can use this sample to compute posterior expectations and probabilities.

For instance, if after running the chain for 10,000 steps, I look at every fifth value (say) of Θ_1 for 10,000 more cases, then I have a sample of size 10,000 from the (marginal) posterior distribution of Θ_1. (Call this Θ_{1i} for $i = 1, \ldots, 10000$.)

I can approximate:

- $\mathbb{E}(h(\Theta_1) \mid X, Y)$ by $\frac{1}{10000} \sum_{i=1}^{10000} h(\Theta_{1i})$
- $\mathbb{P}\{\Theta_1 > c \mid X, Y\}$ by the proportion of Θ_{1i}'s that are $> c$.
- and so forth.

The Gibbs' sampler is convenient and efficient, *when we can compute the complete conditionals.*

- Why does this work?

  To see why, the Gibbs sampler works, we look at a slightly more general situation first.

  Suppose X has distribution π and h is a function on the possible values of X. Define $Y = h(X)$. Then

  $$P(x, A) = \mathbb{P}\{X \in A \mid Y \text{ near } h(x)\}$$

  is the transition kernel for a Markov chain $(X_n)_{n \geq 0}$ where we sample X_{n+1} from the conditional distribution of $X \mid Y = h(X_n)$.

The key fact is that π is an invariant distribution for the chain. To see this, write πP to denote the distribution of a random variable obtained by starting in distribution π and taking one step under the transition kernel P. Formally, we have

$$\begin{aligned}
\pi P(A) &= \int P(x, A)\pi(dx) \\
&= \int \int_{t \in A} P(x, dt)\pi(dx) \\
&= \int \int_{\substack{t \in A \\ h(t)=h(x)}} P(x, dt)\pi(dx) \\
&= \int \int_{\substack{t \in A \\ h(t)=h(x)}} \mathbb{P}\{X \text{ near } t \mid Y \text{ near } h(x)\}\, \mathbb{P}\{X \text{ near } x\} \\
&= \int \int_{\substack{t \in A \\ h(t)=h(x)}} \mathbb{P}\{Y \text{ near } h(t) \mid X \text{ near } t\}\, \frac{\mathbb{P}\{X \text{ near } x\}}{\mathbb{P}\{Y \text{ near } h(x)\}} \mathbb{P}\{X \text{ near } t\} \\
&= \int \int_{\substack{t \in A \\ h(t)=h(x)}} \frac{\mathbb{P}\{X \text{ near } x, Y \text{ near } h(x)\}}{\mathbb{P}\{Y \text{ near } h(x)\}} \mathbb{P}\{X \text{ near } t\} \\
&= \int \int_{\substack{t \in A \\ h(t)=h(x)}} \frac{\mathbb{P}\{X \text{ near } x, Y \text{ near } h(t)\}}{\mathbb{P}\{Y \text{ near } h(t)\}} \mathbb{P}\{X \text{ near } t\} \\
&= \int \int_{\substack{t \in A \\ h(t)=h(x)}} P(t, dx)\pi(dt) \\
&= \int_{t \in A} \int_{\substack{x \\ h(x)=h(t)}} P(t, dx)\pi(dt) \\
&= \int_{t \in A} \pi(dt) \\
&= \pi(A).
\end{aligned}$$

In general, $P(x, A)$ will not necessarily satisfy the conditions we need to make the chain ergodic.

But we can find a set of functions $h_1, \ldots, h_m$ with corresponding kernels $P_1, \ldots, P_m$ and then construct a new kernel by choosing one at random at each stage or cycling between them.

Now, back to the **Gibbs sampler**. This uses the above strategy with functions h_i defined by

$$h_i(x) \equiv h_i(x_1, \ldots, x_m) = (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_m).$$

The distribution of X given $h_i(X)$ is a complete conditional.

In practice, we either cycle through the variables or select them randomly. (There are significant advantages to the latter – specifically detailed balance – although the former is more common.)

Metropolis-Hastings Sampler

In the **Metropolis-Hastings (MH) sampler**, the Markov chain moves to a new state in a two-stage process:

1. We propose a **candidate state**, drawn from one or more pre-specified distributions, called **candidate distributions**.
2. We test the candidate state. If we *accept* it, then the chain moves to that state. If we *reject* it, then the chain stays where it is.

Acceptance or rejection of a candidate depends on the relative values of our target equilibrium distribution. For Bayesian modeling, the target distribution is the posterior distribution, but we only have the *unnormalized posterior distribution*. An advantage of the MH sampler is that it only uses the unnormalized values of the posterior because it chooses to accept or reject based on relative values at the current and candidate states.

Loosely speaking, the MH sampler will always move if the move increases the posterior; otherwise, it will move with a probability based on how much smaller the posterior will be after the move (smaller values are less likely to be accepted).

By choosing candidate distributions, we can craft the kinds of moves our chain will make, which is balanced by the shape of the posterior. The result is an ergodic chain wiht the posterior as limiting distribution.

- **Example**: The Normal-Normal Model.

  Consider data X drawn from a Normal(Θ, σ^2) distribution, where σ^2 is known. We put a Normal$(0, 1)$ prior on Θ. We want to compute the posterior distribution of Θ given X.

  To craft a MH sampler for this problem, we need to specify a candidate distribution (or more than one). For now, we will use a simple one: a symmetric, Normal random walk.

  That is, if the chain is currently in state θ, we will choose the candidate θ' to be $\theta' = \theta + \text{Normal}(0, \tau^2)$, for a value $\tau > 0$ that we select.

- Details

  The above gives the basic ideas. Let's look at the details more formally.

  Let Q be a probability transition kernel defined by

  $$Q(x, dy) = q(x, y)dy.$$

  This q is density of the *candidate distribution*. It specifies how to generate a candidate next state y for the chain given that the chain is currently at state x. (We could write it $q(y|x)$, which would probably be more clear.)

  The Metropolis-Hastings (MH) chain then chooses randomly whether to *accept* the candidate and move to y or *reject* the candidate and stay at x. The choice is made so that an increase in the posterior is always accepted and a decrease is sometimes accepted.

  More specifically, define a Markov chain $(X_n)_{n\geq 0}$ with transition probabilities

  $$P(x, \mathrm{d}y) = p(x, y)\,\mathrm{d}y + r(x)\delta_x(\mathrm{d}y)$$

  where δ_x is a point-mass at the current state (stay where we are) and where

  $$p(x, y) = \begin{cases} q(x, y)\alpha(x, y) & \text{if } y \neq x \\ 0 & \text{if } y = x, \end{cases}$$

  $$r(x) = 1 - \int p(x, t)\,\mathrm{d}t,$$

and where the *acceptance probability* is given by

$$\alpha(x,y) = \begin{cases} \min\left\{\frac{\pi(y)q(y,x)}{\pi(x)q(x,y)}, 1\right\} & \text{if } \pi(x)q(x,y) > 0 \\ 0 & \text{if } \pi(x)q(x,y) = 0. \end{cases}$$

This α is the key to the algorithm. The ratio $\pi(y)/\pi(x)$ determines the relative posterior value at the candidate versus the current states. Notice that the transition probabilities are defined only in terms of ratios of π, so the unknown normalizing constant cancels. The ratio $q(y,x)/q(x,y)$ (or alternatively $q(x|y)/q(y|x)$ if that's clearer) accounts for asymmetry in our candidate selection mechanism. For a symmetric choice, like the symmetric random walk above, this is 1.

The behavior off the Metropolis-Hastings (MH) chain is determined by the choice of candidate distribution. There is an art to this selection: too concentrated and we accept every move but do not move far, too diffuse and we move far but very rarely.

Examples of commonly used candidate families are:

- Independence chains: $q(x,y) = f(y)$ for some density f
- Random Walk chains: $q(,y) = f(y-x)$ for some density f.
- Symmetrix Candidate distribution: $q(x,y) = q(y,x)$
- Gibb's Sampler: q is complete conditional (or mixture of them)

- Why does the MH chain work? Notice first that

$$\pi(x)p(x,y) = \pi(y)p(y,x). \tag{2}$$

(This implies the chain is reversible, giving detailed balance.)

Now, we can show that π is a stationary distribution for the chain. Again, for a subset of states A, we write πP to denote the distribution of a random variable obtained by drawing from π and then taking a step according to the transition kernel P.

The result is:

$$\begin{aligned}
\pi P(A) &= \int P(x,A)\pi(dx) \\
&= \int \left[\int_{y\in A} p(x,y)dy\right]\pi(x)dx + \int r(x)\delta_x(A)\pi(x)dx \\
&= \int \left[\int_{y\in A} \pi(x)p(x,y)dy\right]dx + \int_{x\in A} r(x)\pi(x)dx \\
&= \int \left[\int_{y\in A} \pi(y)p(y,x)dy\right]dx + \int_{x\in A} r(x)\pi(x)dx \\
&= \int_{y\in A}\left[\int p(y,x)dx\right]\pi(y)dy + \int_{x\in A} r(x)\pi(x)dx \\
&= \int_{y\in A}(1-r(y))\pi(y)dy + \int_{x\in A} r(x)\pi(x)dx \\
&= \int_{y\in A}\pi(y)dy \\
&= \pi(A).
\end{aligned}$$

Thus, we have a stationary (and therefore limiting) distribution for the chain.

Reversible Jump MCMC (Brief)

First described by Peter Green in 1995, Reversible Jump MCMC is a method that allows a Markov Chain to jump across spaces of different dimensions.

An example might be when considering regression models with different numbers of variables.

The key issue is maintaining detailed balance in jumps between dimensions. Consider moving between two and one dimensional parameter spaces, for instance.

In brief:

- The state space for the chain is the disjoint union of the various dimensional spaces.
- One selects a variety of different *move types*, some within components and some between. This makes the chain quite flexible.
- Moves types are selected randomly at each step.
- For moves between dimensions, extra care is needed to ensure detailed balance. Green (1995) describes one approach, but there is scope for creativity in crafting these moves

RJMCMC is not fast, but it can be an effective tool for complex models with disjoint pieces that are among the best showcases for Bayesian inference.

Dynamical MCMC

- Idea and Motivation
  - Random walk methods such as Metropolis-Hastings can move slowly, roughly a distance proportional to $\sqrt{m}$ in m steps.

    In high dimensions, this is far too slow to be practical.
  - Embedding the problem in a dynamical system can produce a candidate that moves more easily to distant points.

    Baaic idea:
    - Define a dynamical system
    - Simulate the dynamics for some (small) fixed time step.
    - This candidate is deterministic but under certain conditions (reversibility and unit Jacobian) gives a valid – though not ergodic – chain.
  - Dynamical (aka Hybrid or Hamiltonian) MCMC combines these two approaches ...

    Can produce a faster moving, ergodic chain that can reach distant points even with high-dimensional parameter spaces.
- Setup

  Suppose $\pi(q)$ is (up to a proportionality constant) the posterior we want to sample from, in parameter vector $q = (q_1, \ldots, q_d)$.

  Augment the problem by introducing additional parameters $p = (p_1, \ldots, p_d)$, which we take to be standard Gaussian variables independent of each other and q. Later, we will throw away p.

  Writing $U(q) = -\ln \pi(q)$ and $T(p) = \frac{1}{2}\|p\|^2$, define the "Hamiltonian"

$$H(q,p) = U(q) + T(p) = -\ln \pi(q) + \frac{1}{2}\sum_{i=1}^{d} p_i^2.$$

  This gives us an augmented posterior $\pi(q,p) \propto e^{-H(q,p)}$.

The Hamiltonian H induces "dynamics" via

$$\frac{dq_i}{dt} = \frac{\partial H}{\partial p_i} = p_i$$
$$\frac{dp_i}{dt} = -\frac{\partial H}{\partial q_i} = \frac{\pi'(q)}{\pi(q)}.$$

The dynamics conserve H, preserve volume, and are reversible.

- Method

  The Hybrid Monte Carlo (HMC) algorithm (Duane et al. 1987, Neal 1996) samples from $\pi(q, p)$ by alternating the following two steps:

  1. Gibbs step on p.
     Draw new standard Normal's iid for each p_i.
  2. Metropolis step on q and p.
     Generate candidate state (q', p') by simulating the dynamics for a fixed time τ and negating p.
     (The negation makes the step reversible, ensuring that detailed balance holds.)

  Because $\pi(q, p)$ factors, we can drop p from the sample to recover a sample from the original posterior $\pi(q)$.

- Evaluation

  Strengths:

  - Can move more consistently in one direction
    After m steps, can move distance $\propto m$ rather than $\propto \sqrt{m}$ like Metropolis-Hastings.
  - Can change the probability density value more quickly
    Updating p changes the log density by order $\sqrt{d}$ rather than order 1 as for Metropolis-Hastings
  - Can move more easily to distant points
    Dynamics moves along $\pi(q, p)$ contours (up to discretization error)

  Weaknesses:

  - Performance can depend on discretization scheme
  - Can become trapped in isolated modes or near sharp gradients
  - $\pi(q)$ can change slowly with highly skewed distributions

Practical Issues

Design

Many MCMC simulations have a natural *functional* structure: we apply a transformation to a chain state to obtain a new chain state. This design is quite flexible, allowing good speed, modularity, and memory footprint.

Issues like burn-in, skip, and selection of particular parameters are easily handled as filters on the resulting stream.

Multiple move types are easily supported through selection of different transformations and thus also have a functional structure.

Constructing Estimators/Inferences

The basic idea is to consider the series of states from the chain as a draw from the posterior. Everything we can do with a sample we can do with this: estimate parameters, confidence intervals, compute posterior probabilities of arbitrary events, and so forth.

There are complications because, with some exceptions, we only have an approximate iid draw.

Burn-in

We start the chain somewhere, not necessarily representive of the unknown posterior. It is common to "burn-in" the chain by letting it move for a time to a more reasonable position.

True required burn-in times can be quite long, though people rarely give it that much effort.

There are various approaches to avoid burn-in at all, but they are usually have more narrow application.

Autocorrelation (Skip)

Even in its equilibrium distribution an MCMC sampler can exhibit nontrivial autorrelation. This makes the computation of statistics from the "sample" more complicated – for instance, the variance of a parameter estimate.

One common solution is to skip states in the output sample to reduce autocorrelation. We can also estimate autocorrelation as a diagnostic.

Other Issues

- Combining move types
- One Chain or Many?
- How many samples?
- Diagnostics
- Complex Move Types

Activity

In the simple Normal-Normal model described earlier, we have:

- data X drawn from a Normal(Θ, σ^2) distribution, where σ^2 is known, and
- a Normal$(0, 1)$ prior on Θ.

We want to compute the posterior distribution of Θ given X.

Implement a Metropolis-Hastings MCMC sampler using the symmetric random walk as a candidate distribution: $\theta' = \theta + \text{Normal}(0, \tau^2)$, for a fixed parameter $\tau > 0$.

Your solution should involve writing the six functions described below. The first four are **simple** functions that each return another function. Start by implementing `mh_step`, and then write the others.

- `unnormalized_posterior_density(data, sigma)`

  This returns a *function* of `theta` that computes the unnormalized posterior at `theta`.

- `candidate_density(tau)`

  This returns a *function* of `current` and `candidate` that computes the candidate density of at `candidate` given `current`. That is, the returned function computes $q(\theta_{\text{current}}, \theta_{\text{candidate}})$.

- `candidate_sampler(tau)`

  This returns a *function* of `current`, the current state that produces a candidate state drawn from the candidate distribution.

- `acceptance_probability(p, q)`

  This takes functions `p` and `q` as returned, respectively, by `unnormalized_posterior_density` and `candidate_density` and itself returns a function of `candidate` and `current` calculating the probability of accepting a move from current to candidate.

- `mh_step(current, draw_candidate, alpha)`

  The is function moves the chain a single step. It takes a value of θ and functions `draw_candidate` and `alpha` as returned, respectively, by `candidate_sampler` and `acceptance_probability`.

  It returns the next step in the chain, either `theta_candidate` or `theta_current`, calculating the probability of accepting a move from current to candidate.

- `run_sampler(n.steps, data, draw_initial_state, ...)`

  This puts everything together, a script to run your chain.

Try running your sampler for 2000 steps, and do the following:

- Drop the first 1000 and look at a histogram (or better, a density estimate) based on that sample. You can play with the numbers and parameters here.
- Estimate the mean and variance of the posterior.
- Q: How does the value chosen for τ effect your results?
- Q: How does the length of the "burn-in" affect your results?
- Q: How does the length of the sample affect your results?
- Q: Does it help your mean and variance estimates to skip samples after the burn-in?

The true posterior mean and variance should be $X/(1 + \sigma^2)$ and $\sigma^2/(1 + \sigma^2)$.

Solution

- `unnormalized_posterior_density(data, sigma)`

  This returns a *function* of `theta` that computes the unnormalized posterior at `theta`.

```
unnormalized_posterior_density <- function(data, sigma) {
    return( function(theta) {
                return( dnorm(data, theta, sigma) * dnorm(theta, 0, 1) )
            } )
}
```

- `candidate_density(tau)`

  This returns a *function* of `current` and `candidate` that computes the candidate density of at `candidate` given `current`. That is, the returned function computes $q(\theta_{\text{current}}, \theta_{\text{candidate}})$.

```
candidate_density <- function(tau) {
    return( function(current, candidate) {
                return( dnorm(candidate, current, tau) )
            } )
}
```

- `candidate_sampler(tau)`

  This returns a *function* of `current`, the current state that produces a candidate state drawn from the candidate distribution.

```
candidate_sampler <- function(tau) {
    return( function(current) { rnorm(1, current, tau) } )
}
```

- `acceptance_probability(p, q)`

  This takes functions `p` and `q` as returned, respectively, by `unnormalized_posterior_density` and `candidate_density` and itself returns a function of `candidate` and `current` calculating the probability of accepting a move from current to candidate.

```
acceptance_probability <- function(p,q) {
    return( function(current, candidate) {
                rate_xtoy <- p(current) * q(current, candidate)

                if ( rate_xtoy > 0 ) {
                    return( min(1, p(candidate) * q(candidate, current)/rate_xtoy) )
                } else {
                    return( 0.0 )
                }
            } )
}
```

- `mh_step(current, draw_candidate, alpha)`

  This moves the chain a single step. It takes a value of θ and functions `draw_candidate` and `alpha` as returned, respectively, by `candidate_sampler` and `acceptance_probability`.

  It returns the next step in the chain, either `theta_candidate` or `theta_current`, calculating the probability of accepting a move from current to candidate.

```
mh_step <- function(current, draw_candidate, alpha) {
    candidate <- draw_candidate(current)
    acceptance_prob <- alpha(current, candidate)

    if ( runif(1, 0, 1) <= acceptance_prob ) {
        return( candidate )
    } else {
        return( current )
    }
}
```

- `run_sampler(n.steps, data, draw_initial_state, ...)`

  Putting it all together, a script to run your chain looks like:

```
run_sampler <- function(n.steps, data, sigma=1, tau=1,
                        draw_initial_state=function(){rnorm(1, 0, 3)}) {
    draw_candidate <- candidate_sampler(tau)
    alpha <- acceptance_probability(unnormalized_posterior_density(data, sigma),
                                    candidate_density(tau))

    samples <- rep(NA, n.steps)
    samples[1] <- draw_initial_state()

    for ( i in 1:n.steps ) {
        samples[i+1] <- mh_step(samples[i], draw_candidate, alpha)
    }

    return(list(samples=samples, data=data, sigma_sq=sigma, tau=tau,
                initial_distribution=draw_initial_state))
}
```