A Markov chain Monte Carlo example

Summer School in Astrostatistics, Center for Astrostatistics, Penn State University
Adapted from notes by Murali Haran, Dept. of Statistics, Penn State University

We describe a model that is easy to specify but requires samples from a relatively complicated distribution for which classical Monte Carlo sampling methods are impractical. We describe how to implement a Markov chain Monte Carlo (MCMC) algorithm for this example. The purpose of this is twofold: First to illustrate how MCMC algorithms are easy to implement (at least in principle) in situations where classical Monte Carlo methods do not work and second to provide a glimpse of practical MCMC implementation issues.

Datasets and other files used in this tutorial:

- COUP551_rates.dat
- MCMCchpt.R
- batchmeans.R

pdf files referred to in this tutorial that give technical details:

- changePointExample.pdf
- batchmeans.pdf

Introduction

Monte Carlo methods are a collection of techniques that use pseudo-random (computer simulated) values to approximate solutions to mathematical problems. In this tutorial, we will focus on using Monte Carlo for Bayesian inference. In particular, we will use it for the evaluation of expectations with respect to a probability distribution. Monte Carlo methods can also be used for a variety of other purposes, including estimating maxima or minima of functions (as in likelihood-based inference) but we will not discuss these here.

Monte Carlo works as follows: Suppose we want to estimate an expectation of a function g(x) with respect to the probability distribution f. We denote this desired quantity $m = E_f g(x)$. Often, m is analytically intractable (the integration or summation required is too complicated). A Monte Carlo estimate of m is obtained by simulating N pseudo-random values from the distribution f, say $X_1, X_2, ..., X_N$ and simply taking the average of $g(X_1), g(X_2), ..., g(X_N)$ to estimate m. As N (number of samples) gets large, the estimate converges to the true expectation m.
A toy example to calculate the $P(-1 < X < 0)$ when X is a Normal(0,1) random variable:

```
xs = rnorm(10000) # simulate 10,000 draws from N(0,1)
xcount = sum((xs>-1) & (xs<0)) # count number of draws between -1 and 0
xcount/10000 # Monte Carlo estimate of probability
pnorm(0)-pnorm(-1) # Compare it to R answer (cdf at 0) - (cdf at -1)
```

Importance sampling: Another powerful technique for estimating expectations is importance sampling,

where we produce draws from a different distribution, say q, and compute a specific weighted average of these draws to obtain estimates of expectations with respect to f. In this case, A Monte Carlo estimate of m is obtained by simulating N pseudo-random values from the distribution q, say $Y_1,Y_2,..,Y_N$ and simply taking the average of $g(Y_1)w_1$, $g(Y_2)w_2$, ..., $g(Y_N)w_N$ to estimate m, where w_1, w_2, ..., w_N are weights obtained as follows: $w_i = f(Y_1)/q(Y_1)$. As N (the number of samples) gets large, the estimate converges to the true expectation m. Often, when normalizing constants for f or q are unknown, and for numerical stability, the weights are 'normalized' by dividing the above weights by the sum of all weights (sum over w_1, ..., w_N).
Importance sampling is powerful in a number of situations, including:

1. When expectations with respect to several different distributions (say $f_1,..,f_p$) are of interest. All these expectations can, in principle, be estimated by using just a single set of samples!
2. When rare event probabilities are of interest so ordinary Monte Carlo would take a huge number of samples for accurate estimates. In such cases, selecting q appropriately can produce much more accurate estimates with far fewer samples.

Discussions of importance sampling in astronomical Bayesian computation appear in papers by Lewis & Bridle and Trotta for cosmological parameter estimation and Ford for extrasolar planet modeling.

R has random number generators for most standard distributions and there are many more general algorithms (such as rejection sampling) for producing independent and identically distributed (i.i.d.) draws from f. Another very general approach for producing non-i.i.d. draws (approximately) from f is the Metropolis-Hastings algorithm.

Markov chain Monte Carlo : For complicated distributions, producing pseudo-random i.i.d. draws from f is often infeasible. In such cases, the Metropolis-Hastings algorithm is used to produce a Markov chain say $X_1,X_2,..,X_N$ where the X_i's are *dependent* draws that are *approximately* from the desired distribution. As before, the average of $g(X_1)$, $g(X_2)$, ..., $g(X_N)$ is an estimate that converges to m as N gets large. The Metropolis-Hastings algorithm is very general and hence very useful. In the following example we will see how it can be used for inference for a model/problem where it would otherwise be impossible to compute desired expectations.

Problem and model description

First, a five minute review of Bayesian inference
We begin by specifying a probability model for our data Y by assuming it is generated from some distribution h(theta;Y), where theta is a set of parameters for that distribution. This is written Y~h(theta). We want to infer theta from the fixed, observed dataset Y. First, consider likelihood inference. We find a value of theta where the likelihood L(theta;Y) (which is obtained from the probability distribution h(theta;Y)) is maximized; this is the maximum likelihood estimate (MLE) for theta. Now consider Bayesian inference. We assume a prior distribution for theta, p(theta), based on our previous knowledge. This prior may be based on astrophysical insights (e.g. no source can have negative brightness), past astronomical observation (e.g. stars have masses between 0.08-150 solar masses), and/or statistical considerations (e.g. uniform or Jeffreys priors) when it is difficult to obtain good prior information. Inference is based on the posterior distribution Pi(theta|Y) which is proportional to the product of the likelihood and the prior. It is only *proportional* to this product because in reality Bayes theory requires that

we write down a denominator (the integral of the product of the likelihood and prior over the parameter space). Fortunately, Markov chain Monte Carlo algorithms avoid computation of this denominator while still producing samples from the posterior Pi(theta| Y). Note that the MCMC methods discussed here are often associated with Bayesian computation, but are really independent methods which can be used for a variety of challenging numerical problems. Essentially, any time samples from a complicated distribution are needed, MCMC may be useful.

Our example uses a dataset from the Chandra Orion Ultradeep Project (COUP). This is a time series of X-ray emission from a flaring young star in the Orion Nebula Cluster. More information on this dataset is available at: CASt Chandra Flares data set . The raw data, which arrives approximately according to a Poisson process, gives the individual photon arrival times (in seconds) and their energies (in keV). The processed data we consider here is obtained by grouping the events into evenly-spaced time bins (10,000 seconds width).

Our goal for this data analysis is to identify the change point and estimate the intensities of the Poisson process before and after the change point. We describe a Bayesian model for this change point problem (Carlin and Louis, 2000). Let Yt be the number of occurrences of some event at time t. The process is observed for times 1 through n and we assume that there is a change at time k, i.e., after time k, the event counts are significantly different (higher or lower than before). The mathematical description of the model is provided in changePointModel (pdf) . While this is a simple model, it is adequate for illustrating some basic principles for constructing an MCMC algorithm.

We first read in the data:

```
chptdat = read.table("http://www.stat.psu.edu/~mharan/MCMCtut/COUP551_rates.dat",skip=1)
```

Note: This data set is just a convenient subset of the actual data set (see reference below.)

We can begin with a simple time series plot as exploratory analysis.

```
Y=chptdat[,2] # store data in Y
ts.plot(Y,main="Time series plot of change point data")
```

The plot suggests that the change point may be around 10.

Setting up the MCMC algorithm

Our goal is to simulate multiple draws from the posterior distribution which is a multidimensional distribution known only upto a (normalizing) constant. From this multidimensional distribution, we can easily derive the conditional distribution of each of the individual parameters (one dimension at a time). This is described, along with a description of the Metropolis-Hastings algorithm in changePointModel (pdf) .

Programming an MCMC algorithm in R

We will need an editor for our program. For instance, we can use Wordpad (available under the Start button menu under Accessories). Ideally, a more `intelligent' editor such as emacs (with ESS or emacs speaks statistics installed) should be used to edit R programs.
Please save code from MCMC template in R into a file and open this file using the editor. Save this file as MCMCchpt.R .

Note that in this version of the code, all parameters are sampled except for k (which is fixed at our guessed change point).
To load the program from the file MCMCchpt.R we use the "source" command. (Reminder: It may be helpful to type:

```
setwd("V:/")
```

to set the default directory to the place where you can save your files)

```
source("MCMCchpt.R") # with appropriate filepathname
```

We can now run the MCMC algorithm:

```
mchain <- mhsampler(NUMIT=1000,dat=Y) # call the function with appropriate arguments
```

MCMC output analysis

Now that we have output from our sampler, we can treat these samples as data from which we can estimate quantities of interest. For instance, to estimate the expectation of a marginal distribution for a particular parameter, we would simply average all draws for that parameter so to obtain an estimate of E(theta):

```
mean(mchain[1,]) # obtain mean of first row (thetas)
```

To get estimates for means for all parameters:

```
apply(mchain,1,mean) # compute means by row (for all parameters at once)
apply(mchain,1,median) # compute medians by row (for all parameters at once)
```

To obtain an estimate of the entire posterior distribution:

```
plot(density(mchain[1,]),main="smoothed density plot for theta posterior")
plot(density(mchain[2,]),main="smoothed density plot for lambda posterior")
hist(mchain[3,],main="histogram for k posterior")
```

To find the (posterior) probability that lambda is greater than 10

```
sum(mchain[2,]>10)/length(mchain[2,])
```

Now comment the line that fixes k at our guess (add the # mark) :

```
# currk <- KGUESS
```

Rerun the sampler with k also sampled.

```
mchain <- mhsampler(NUMIT=1000,dat=Y)
```

With the new output, you can repeat the calculations above (finding means, plotting density estimates etc.)

You can also study how your estimate for the expectation of the posterior distribution for k changes with each iteration.

```
estvssamp(mchain[3,])
```

We would like to assess whether our Markov chain is moving around quickly enough to produce good estimates (this property is often called 'good mixing'). While this is in general difficult to do rigorously,

estimates of the autocorrelation in the samples is an informal but useful check. To obtain sample autocorrelations we use the acf plot function:

```
acf(mchain[1,],main="acf plot for theta")
acf(mchain[2,],main="acf plot for lambda")
acf(mchain[3,],main="acf plot for k")
acf(mchain[4,],main="acf plot for b1")
acf(mchain[5,],main="acf plot for b2")
```

If the samples are heavily autocorrelated we should rethink our sampling scheme or, at the very least, run the chain for much longer. Note that the autocorrelations are negligible for all parameters except k which is heavily autocorrelated. This is easily resolved for this example since the sampler is fast (we can run the chain much longer very easily). In problems where producing additional samples is more time consuming, such as complicated high dimensional problems, improving the sampler `mixing' can be much more critical.

Why are there such strong autocorrelations for k? The acceptance rate for k proposals (printed out with each MCMC run) are well below 10% which suggests that k values are stagnant more than 90% of the time. A better proposal for the Metropolis-Hastings update of a parameter can help improve acceptance rates which often, in turn, reduces autocorrelations. Try another proposal for k and see how it affects autocorrelations. In complicated problems, carefully constructed proposals can have a major impact on the efficiency of the MCMC algorithm.

How do we choose starting values? In general, any value we believe would be reasonable under the posterior distribution will suffice. You can experiment with different starting values. For instance: modify the starting value for k in the function (for instance, try setting k=10), "source" the function in R and run the sampler again as follows:

```
mchain2 <- mhsampler(NUMIT=1000,dat=Y)
```

You can study how your estimate for the expectation of the posterior distribution for k changes with each iteration.

```
estvssamp(mchain2[3,])
```

Assessing accuracy and determining chain length

There are two important issues to consider when we have draws from an MCMC algorithm: (1) how do we assess the accuracy of our estimates based on the sample (how do we compute Monte Carlo standard errors?) (2) how long do we run the chain before we feel confident that our results are reasonably accurate ?

Regarding (1): Computing standard errors for a Monte Carlo estimate for an i.i.d. (classical Monte Carlo) sampler is easy, as shown for the toy example on estimating P(-1< X < 0) when X is a Normal(0,1) random variable. Simply obtain the sample standard deviation of the g(xi) values and divide by square root of n (the number of samples). Since Markov chains produce dependent draws, computing precise Monte Carlo standard errors for such samplers is a very difficult problem in general. For (2): Draws produced by classical Monte Carlo methods (such as rejection sampling) produced draws from the correct distribution. The MCMC algorithm produces draws that are asymptotically from the correct distribution. All the draws we see after a finite number of iterations are therefore only approximately from the correct distribution. Determining how long we have to run the chain before we feel sufficiently confident that the MCMC

algorithm has produced reasonably accurate draws from the distribution is therefore a very difficult problem. Most rigorous solutions are too specific or tailored towards relatively simple situations while more general approaches tend to be heuristic.

There are many ways to compute Monte Carlo standard errors. Two simple but reasonable ways of calculating it: the consistent batch means (bm) and the iterated monotone sequence estimator (imse) method in R. References: Flegal, J.M., Haran, M., and Jones, G.L. (2008) Markov chain Monte Carlo: Can we trust the third significant figure? *Statistical Science (in press),* and Geyer, C.J. (1992) Practical Markov chain Monte Carlo, *Statistical Science* .

To compute MC standard error via batch means, source the batchmeans.R file:

```
source("batchmeans.R")
```

We can now calculate standard error estimates for each of the five parameter estimates:

```
bm(mchain[1,])
bm(mchain[2,])
bm(mchain[3,])
bm(mchain[4,])
bm(mchain[5,])
```

Are these standard errors acceptable ?

There is a vast literature on different proposals for dealing with the latter issue (how long to run the chain) but they are all heuristics at best. The links at the bottom of this page (see section titled "Some resources") provide references to learn more about suggested solutions. One method that is fairly simple, theoretically justified in some cases and seems to work reasonably well in practice is as follows: run the MCMC algorithm and periodically compute Monte Carlo standard errors. Once the Monte Carlo standard errors are below some (user-defined) threshold, stop the simulation.

Often MCMC users do not run their simulations long enough. For complicated problems run lengths in the millions (or more) are typically suggested (although this may not always be feasible). For our example run the MCMC algorithm again, this time for 100000 iterations (set NUMIT=100000).

```
mchain2 <- mhsampler(NUMIT=100000,dat=Y)
```

You can now obtain estimates of the posterior distribution of the parameters as before and compute the new Monte Carlo standard error. Note whether the estimates and corresponding MC standard error have changed with respect to the previous sampler.

Making changes to the model

If we were to change the prior distributions on some of the individual parameters, only relatively minor changes may be needed in the program. For instance if the Inverse Gamma prior on b1 and b2 were replaced by Gamma(0.01,100) priors on them, we would only have to change the lines in the code corresponding to the updates of b1 and b2 (we would need to perform a Metropolis-Hastings update of each parameter). The rest of the code would remain unchanged. Modifying the program to make it sample from the posterior for the modified model is a useful exercise. For the modified full conditionals see modified full conditional distributions.

An obvious modification to this model would be to allow for more than one change point. A very sophisticated model that may be useful in many change point problems is one where the number of change points is also treated as unknown. In this case the *number* of Poisson parameters (only two of them in our example: theta and lambda) is also unknown. The posterior distribution is then a mixture over distributions of varying dimensions (the dimensions change with the number of change points in the model). This requires an advanced version of the Metropolis-Hastings algorithm known as **reversible-jump Metropolis Hastings** due to Peter Green (*Biometrika, 1995*). Some related information is available at the HSSS variable dimension MCMC workshop

Some resources

The "CODA" and BOA packages in R implement many well known output analysis techniques. Charlie Geyer's MCMC package in R is another free resource. There is also MCMC software from the popular Bugs/WINBugs and OpenBugs projects.

In addition to deciding how long to run the sampler and how to compute Monte Carlo standard error, there are many possibilities for choosing how to update the parameters and more sophisticated methods used to make the Markov chain move around the posterior distribution efficiently. The literature on such methods is vast. The following references are a useful starting point.

Acknowledgment: The model is borrowed from Chapter 5 of "Bayes and Empirical Bayes Methods for Data Analysis" by Carlin and Louis (2000). The data example was provided by Konstantin Getman (Penn State University). Return to home page