

Some Examples of (Markov Chain) Monte Carlo Methods

Ryan R. Rosario

What is a Monte Carlo method?

Monte Carlo methods rely on repeated sampling to get some computational result. Monte Carlo methods originated in Physics, but no Physics knowledge is required to learn Monte Carlo methods! The name “Monte Carlo” was the codename applied to some computational methods developed at the Los Alamos Lab while working on nuclear weapons. Yes, the motivation of the codename was the city in Monaco, but does not come *directly* from gambling.

Monte Carlo methods all follow a similar pattern:

1. Define some domain of inputs. This just means we have some set of variables and what values they can take on, or we have some observations that are part of a dataset.
2. Generate inputs (the values of the variables, or sets of observations) randomly, governed by some probability distribution.
3. Perform some computation on these inputs.
4. Repeat 2 and 3 over and over either an infinite number of times (a very large number of times usually ≥ 10000), or until convergence.
5. Aggregate the results from the previous step into some final computation.

The result is an approximation to some true but unknown quantity, which is no big deal since that is all we ever do in Statistics!

A Motivating Example with Code: The Bootstrap

Suppose we have a small sample and we want to predict y using x . If we fit a standard linear regression model, we may not be able to trust the parameter estimates, and particularly that standard error of these estimates ($SE(\hat{\beta}_1)$). To estimate the standard error, we can use the bootstrap.

Step 1 from the little algorithm above is already done for us. The domain of inputs is just the observations in the sample.

2. Draw a random sample (with replacement) of size n from the data. This is called a *bootstrap sample*. (generate inputs)
3. Fit a linear regression model and get the value of $\hat{\beta}_1$ from it. **Store this value, as we will need it.** (do something)
4. Repeat steps 1 and 2, say 10,000 times (like with any good shampoo, repeat).
5. Once we have finished the 10,000th iteration, we have a nice collection of $\hat{\beta}_1$ s, a *sampling distribution*. Recall from Stats 10 that the standard deviation of a sampling distribution is called the *standard error*. Thus, if we take the standard deviation of our collection of $\hat{\beta}_1$ s, we can *approximate* the standard error of $\hat{\beta}_1$. (aggregate)

Again, this is just an example. If you are not familiar with the bootstrap, it's not a problem. It's a pretty cool method though, so you may want to take a look at...

In this class we will be programming in R. There are a couple of things from your PIC 10A class we will use. Of course, in R land, not C++ land.

These Monte Carlo methods rely on *repeatedly* drawing samples. So we need a loop. There are a few different kinds of loops. Let's look at two that you will need for this class.

The for Loop

The `for` loop defines some counter variable. In basic programming texts, this is usually `i` or `j` etc. Here I will call it `it` which stands for iteration, but it really doesn't matter what you call it. I define a variable, `its` which stands for *iterations*, the total number of times I want to run this simulation – or more succinctly, the number of samples I want to draw.

The `for` loop runs once for each value of `it` in the range given. Here, the range will be `1:its` that is, `it = 1, ..., its`. The `for` loop will do *something* `its = 10,000` times. We also need to initialize some values that will be used within the loop. First, we need to specify the sample size for each bootstrap sample, $n = 15$. We also need to initialize a vector to store all of the intermediate results from the sampling process.

Also, sampling in R is usually better achieved by using indices of observations rather than the observations/rows themselves. The `seq(a, b)` function just returns a vector of integers from a to b inclusive, which are the indices. The indices will just be the values 1 through the number of observations in X and Y which is given by the `length` function.

So far we have

```
1  its <- 10000           #number of iterations
2  n <- 15                #size of each bootstrap sample
3  b <- rep(0, its)      #repeats the number 0, its times
4                          #yielding a vector of length its.
5  indices <- seq(1, length(Y)) #generate a sequence from 1 to whatever
6                          #the length of Y is.
7  for (it in 1:its)
8  {
9      ..... stuff to repeat ..... (steps 2 and 3)
10 }
```

Now we need to implement the little algorithm above. First, we need to draw a random sample from the dataset. So first let's write the code to draw a random sample of size n from the dataset. The `sample` function has the format `sample(from, size, with.replacement?)` and draws a random sample from `from` of size `size`. Without the third argument, `sample` will sample *without* replacement, so we need to pass `replace=TRUE`. `bootstrap.sample` below contains the sampled indices.

```
#draw a random sample of size n WITH replacement.
bootstrap.sample <- sample(indices, n, replace=TRUE)
```

Next, we fit a linear model using only the observations in the bootstrap sample. We subset on `bootstrap.sample`:

```
the.model <- lm(Y[bootstrap.sample]~X[bootstrap.sample])
```

`the.model` is an object, and contains several different things: the coefficients, residuals, etc. We want to extract the coefficients from the model, particularly $\hat{\beta}_1$ which will be the second element of the coefficient vector – $\hat{\beta}_0$ is the first element.

```

#First, fit the model.
the.model <- lm(Y[bootstrap.sample]~X[bootstrap.sample])
#Next, extract the second coefficient from this model.
bootstrap.beta <- the.model$coefficients[2]
#Finally, store this value for later use!
b[it] <- bootstrap.beta

```

Note that we have constructed a sampling distribution of $\hat{\beta}_1$ now. The standard deviation of a sampling distribution is the standard error, so taking the standard deviation of **b** yields the value we want, $SE(\beta_1)$.

The final code is,

```

1  its <- 10000           #number of iterations
2  n <- 15               #size of each bootstrap sample
3  b <- rep(0, its)      #repeats the number 0, its
4                          #times yielding a vector of length its.
5  #generate a sequence from 1 to whatever the length of Y is.
6  indices <- seq(1, length(Y))
7  for (it in 1:its)
8  {
9      #Steps 2 and 3
10     #draw a random sample of size n WITH replacement.
11     bootstrap.sample <- sample(indices, n, replace=TRUE)
12     #First, fit the model.
13     the.model <- lm(Y[bootstrap.sample]~X[bootstrap.sample])
14     #Next, extract the second coefficient from this model.
15     bootstrap.beta <- the.model$coefficients[2]
16     #Finally, store this value for later use!
17     b[it] <- bootstrap.beta
18 }
19 #The iterations of the above for loop is step 4.
20 #Final aggregation, step 5.
21 sd(b)

```

The for loop is appropriate when we know how many iterations we need to run, otherwise, the repeat loop is better.

The repeat Loop

The repeat loop is similar to the while loop you learned about in PIC 10A. We use the repeat loop when we do not necessarily know the number of iterations to perform. This is particularly useful when we need to *monitor for convergence*, something we will do over and over.

We start by initializing a counter variable, called *it* again (but you can call it whatever you want). We also must initialize a vector where we will store the $\hat{\beta}_1$ s, but since we do not assume to know the number of iterations in advance, we must start with an empty vector, and insert a value at each iteration.

We then tell R to repeat a series of computations. At the end of each iteration, a condition is checked (this is where we would check for convergence). If the condition is true, we break out of the loop, otherwise we increment the counter and continue.

The `ifelse(logical condition to check, what to do if TRUE, what to do if FALSE)` statement is a major difference in the body of the loop. Here, we will just check whether or not we have reached *its*, the total number of iterations.

```

1 #Same as before:
2 its <- 10000           #number of iterations
3 n <- 15               #size of each bootstrap sample
4 #generate a sequence from 1 to whatever the length of Y is.
5 indices <- seq(1, length(Y))
6
7 #Differences for repeat:
8 it <- 1 #start at 1st iteration.
9 b <- c()             #c combines two items into a vector. c() = empty vector
10 repeat {
11     #Steps 2 and 3
12     #draw a random sample of size n WITH replacement.
13     bootstrap.sample <- sample(indices, n, replace=TRUE)
14     #First, fit the model
15     the.model <- lm(Y[bootstrap.sample]~X[bootstrap.sample])
16     #Next, extract the second coefficient from this model.
17     bootstrap.beta <- the.model$coefficients[2]
18     #Finally, tack on this value to the b vector.
19     b <- c(b, bootstrap.beta)
20     #Determine if we can break out of the loop.
21     ifelse(it >= its, break, it <- it + 1)
22     #We test if it >= its
23     #(usually this is where we put convergence test)
24     #If true, break out of the loop.
25     #If false, increment it and start the next iteration.
26 }
27 sd(b) #Step 5, final result

```

Example: Computing π

How can we compute π ? Mathematicians have been doing this for ages, *precisely*. Every few years, some mathematician will announce “I have found the 542nd digit of π !” How the mathematicians do this exactly, I am not sure, but we can do it much easier using Monte Carlo methods.

1. Create a square “game board” and inscribe a circle in it (domain of infinitely many binary inputs, either a piece of rice lands on a space, or it doesn’t, ignoring the boundary of the circle that is). What is the ratio of the area of the circle and the corresponding square?

$$\frac{\text{Area of Circle}}{\text{Area of Square}} = \frac{\pi r^2}{l^2} = \frac{\pi \left(\frac{l}{2}\right)^2}{l^2} = \frac{\pi l^2}{4l^2} = \frac{\pi}{4}$$

2. Drop small pieces of rice onto this board **uniformly** at random. (generate inputs randomly)
3. Do this over and over again, say 10,000 times. (repeat)
4. Count the number of grains of rice that are in the circle. Assume that no piece of rice is on the boundary of the circle, and assume that we kept track of how many grains of rice we dropped. Then, the *probability* that a piece of rice landed in the circle is

$$p = \frac{\# \text{ of Grains of Rice in Circle}}{\# \text{ of Grains of Rice Dropped, Total}}$$

We know that the probability should be approximately $\frac{\pi}{4}$. Then what is π ?

$$\frac{\pi}{4} = p \quad \implies \quad \pi = 4p$$

But, if it were this simple, some mathematicians would be out of a job. **The trick to getting a good approximation of π is to use a huge number of grains of rice.** If we were doing this experiment in real life, we would also probably want to make the square as large as possible.

A common question may be, “but if I were to drop a bunch of grains of rice onto the game board, wouldn’t they all stack up in the same place?” Yes, that is why we must sample the square **uniformly** – each time we drop a piece of rice, we drop it from a randomly chosen location! A robot would be ideal for this little experiment.

There is another way to formulate the problem. The famous **Buffon’s Needle**¹ problem is a more complicated, but beautiful, explanation.

¹see http://en.wikipedia.org/wiki/Buffon's_needle

Example: Simulated Annealing and Stochastic Search Applied to Scheduling and AI

When you were in high school, you were given a class schedule each year containing classes that you wanted to take, at the appropriate difficulty level. Somehow, your schedule always seemed to work out so that you *always* got the classes you *needed* and *usually* got the classes you *wanted*.

High school scheduling and similar assignment problems are *constraint satisfaction problems*. Schedules must adhere to certain rules. For example, a football player needed to be scheduled into “Football” during the last period of the day (at least at my school). This means that the AP Statistics class he needs to take must be scheduled during some other period of the day, otherwise, there is a *conflict*. With thousands of students at a high school, how can a counselor schedule all of the courses and teachers AND schedule all of the students with minimal conflicts? It is not done manually. In fact, many commercial high school scheduling/records systems use methods based on the process below to schedule teachers and their classes, as well as students of course. Some of these methods include simulated annealing and Tabu search. These systems are very expensive. **There can be big money in MCMC.**

Or, consider University class timetabling. We know that, for example, there must be enough sections of Statistics 100C so that students that are taking 101C and 102C can also take 100C. Faculty preferences and room availability also serve as constraints. At UCLA course scheduling is done by the Department and is usually handled manually, but at many universities it is done automatically using software based on these methods.

Or, what about roommate matching? When a student applies for housing, he/she fills out a questionnaire about various living habits. Some of these are hard constraints such as smoking. A non-smoker that does not want to live with a smoker will not be put in that situation. Others are soft constraints, such as a rating on how social the students are.

A mathematician may initially want to solve these problems combinatorially by enumerating every possible combination of results and searching for the best one. There are two problems with this approach 1) it favors those that come first in the process, 2) it will never finish. Instead, we perform a Monte Carlo optimization.

1. First, generate a random *partial* solution. Assign students to random high school classes or roommate configurations. While a random selection works, in practice we usually choose a partial solution that makes sense. This simply shortens the computational process. **Each partial solution is called a *state*.**
2. Next, we propose a new state by *randomly* swapping two objects: a class on a student’s schedule, a roommate etc. We also calculate what the cost would be for this new state, and compare it to the cost of the current state.
 - if the new state is worse than the current state, the cost will increase and we *may* move to this new state.
 - if the new state is better than the current state, the cost will decrease and we *will* move to this new state.

Let δ be the change in the cost from the current state to the new state. If the new state is worse than the current state, we compute the quantity

$$p = e^{-\frac{\delta}{T(t)}}$$

T is a function with respect to time, or a constant, (called the *temperature*) and is chosen beforehand so that the quantity is a probability.

Then, we draw a random uniform number m . If $m < p$, we move to the new state – we swap the two objects and move on. Otherwise, we do not perform the swap, and the solution remains the same as it was before. If the new state is better than the current state, we move to it with probability 1.

3. Repeat the previous two steps either N times, where N is a big number, or until the difference $\delta < \epsilon$ where ϵ is a tiny number.
4. The output is the state that yielded the smallest cost after convergence.

What is a Cost Function?

A cost function is some function f such that bad states (incompatibilities) yield high cost, and good states yield low cost. For example, consider the roommate matching problem. Let f be defined as a weighted sum of *mismatching characteristics*.

$$f = \sum 10|\text{Smoke}_2 - \text{Smoke}_1| + 3|\text{Night}_2 - \text{Night}_1| + |\text{Social}_2 - \text{Social}_1|$$

where the sum is computed over all pairs of roommates in the current configuration/state.

- Smoke_i is a 0/1 variable whether or not roommate i smokes. The difference given above represents a mismatch. If they both smoke, or both do not smoke, the difference is 0, no mismatch.
- Night_i is 0/1 variable whether or not roommate i is a night person. Again, if the roommates differ we get a positive value representing a mismatch, otherwise 0 represents no mismatch.
- Social_i is a number between 1 (hermit) and 10 (celebrity).

The weights I chose are arbitrary and just represent that a smoking mismatch is much more serious than a sociality mismatch. Of course there are ways to choose which variables are important, and ways to optimize the weights.

The goal is to minimize the cost/dissimilarity function. This is the big picture. In Statistics we are always optimizing some function: loss, squared residuals, finding maximum likelihood estimators, finding maxima/minima of a function.

This process is called **simulated annealing**². Simulated annealing is an example of the **Metropolis-Hastings algorithm**. Essentially, this process models the process of annealing in metallurgy – the heating and controlled cooling of a material to increase the size of its crystals and decrease the number of defects³. These weird analogies are common in MCMC.

Biologists and computer scientists have come up with other optimization algorithms that come from the study of animals and evolution. Some species that have been studied and applied to optimization problems are ants (ant colony optimization, where ants search a space to find productive regions), bees, and fireflies. These algorithms are called **swarm optimization algorithms** and are similar to **genetic algorithms**. Other similar processes include particle swarm optimization, and harmony search (mimics musicians).

All of these methods collectively are called **stochastic search** or **stochastic optimization**.

²see <http://www.mit.edu/~dbertsim/papers/Optimization/Simulated%20annealing.pdf> for more theory.

³http://en.wikipedia.org/wiki/Simulated_annealing

Example: Markov Chains applied to PageRank

Google computes a value called PageRank for each entry in its search engine. PageRank essentially measures how important or valuable a particular website/webpage is. PageRank is part of what determines which web page is displayed in which rank when you hit the Search button. For some page p , PageRank uses the number of pages that link to p , as well as the number of pages that p links to compute the metric.

For some page p , the PageRank is defined as

$$\text{PR}(p) = \sum_{v \in B_p} \frac{\text{PR}(v)}{L(v)}$$

where B_p is the set of webpages that link to p and $L(v)$ is the number of pages that p links to. Note that PageRank is essentially a recurrence relation, but the important thing is that the PageRank of p assumes that the user *randomly* surfs from $v \in B_p$ to p . Since the next page visited by the random surfer is, well, random, **it depends only on the previous page visited and no other page. This hints at a Markov chain.** Let X_1, X_2, \dots, X_n be random variables. We say that X satisfies a Markov condition if

$$P(X_{n+1} = x | X_1 = x_1, X_2, \dots, X_n = x_n) = P(X_{n+1} = x | X_n = x_n)$$

Google is able to estimate the PageRank values using MCMC. After a certain number of iterations, the Markov chain converges to a *stationary distribution* of PR values that approximate the probability that a user would randomly surf to some page p . It should be no surprise that the sum of all PageRank values is 1. Why...?

Example: Random Walks applied to Sampling the Web

Web research is a hot field right now. To do research, we usually want a random sample. But how on the web (see what I did there?) do we get a random sample of web pages? Here are some naive ways:

1. generate a random IP address and find that most IP addresses are not associated with web servers, and some IP addresses represent multiple web servers. FAIL.
2. generate random URLs. How would we even start to do that? FAIL.
3. use a Monte Carlo method.

Rusmevichientong et. al. (2001) proposed this method for sampling web pages, and Gjoka et. al. (2009) used this method for selecting a random sample of Facebook users for an analysis.

The problem is formulated as follows.

1. Perform the following slew of steps...

- (a) Pick a web page at random. We could Google the word **the** and pick the first result, or we could pick our favorite web site. Doesn't matter.
- (b) Pick a link on the web page at random and move to the new page.
- (c) Repeat the previous step N times.
- (d) Perform step (b) another K times. Let X_1, \dots, X_K be the collection of web pages visited during this part of the crawl and let D_K be the set of unique web pages visited during this part of the crawl.
- (e) For each unique page p in D_K :
 - i. Pick a random link and move to the new page associated with that link.
 - ii. Repeat the previous step M times yielding a new collection of pages $Z_1^p, Z_2^p, \dots, Z_M^p$.
 - iii. Compute the number of times we visited page p :

$$\widehat{\pi(p)} = \frac{\sum_{r=1}^M 1(Z_r^p = p)}{M}$$

where $1(Z_r^p = p) = 1$ if $Z_r^p = p$ and is 0 otherwise.

- iv. Accept page p into the final sample with probability $\frac{\beta}{\widehat{\pi(p)}}$ where $0 < \beta < \min_{p=1, \dots, K} \widehat{\pi}(X_p)$.

2. Repeat Step 1 a number of times.

3. The resulting sample of web pages is approximately random.

This is quite a complicated algorithm! Notice again that this process is a Markov chain because the page we visit next depends only on the page we are currently at and no other page. First, we visit a bunch (N) of pages (a so called "burn-in" phase). We throw those out. Then, the next K pages are candidates for inclusion into our random sample. But we must select the members of this set according to some probability. We estimate a stationary probability $\hat{\pi}$ by starting another crawl at each of the K candidates. The estimated stationary probability is just the number of times we visited page p when we start a random walk at page p . We get a uniform distribution if we choose the members of the random sample from the K candidates with probability $\frac{1}{\widehat{\pi(p)}}$ but this may not be a probability. The value β is a normalizing constant and the probability that we include page p in the random sample is thus $\frac{\beta}{\widehat{\pi(p)}}$. The idea is that this chain converges to a random sample of the web.

This is an example of **Metropolis-Hastings random walk**. While we will discuss this algorithm in class, we will not do anything remotely this complicated!

All of this convergence talk may remind you of Real Analysis. No need for Real Analysis, however, if you like Analysis and MCMC, you could get carried away with theoretical research in this field!

Where is MCMC Common?

There are some fields/applications/situations where MCMC is very common, and a knowledge of it is a must.

1. **Data Mining and Machine Learning** often yields a lot of weird distributions, whereas in some branches of Statistics like Social Statistics, distributions are friendly normals etc. Many problems in machine learning are impossible to solve deterministically. Computer Vision makes very heavy use of MCMC.
2. **Small and Large Data.** Insufficient samples can require resampling to achieve a result. Big data (several GB) cannot fit in RAM and poses a lot of problems for computation. Again, we could draw smaller samples that will fit in RAM and then combine the results at the end.
3. **Bayesian methods** can often yield messy distributions because they require multiplying together some prior distribution with some likelihood distribution, neither of which need to be friendly. Even if they both are, there is no guarantee (except for conjugate priors) that the resulting posterior distribution will be friendly as well.
4. **Biological and genetic research** for the reasons listed above. The human genome is huge (big data), but sometimes you may only be working with a small number of markers, proteins, whatever (small data). Additionally, many problems in genetics are solved using Bayesian methods. For that reason, biology and genetics are big users of MCMC.