Markov Chain Monte Carlo

Felipe José Bravo Márquez

June 30, 2021

Markov Chain Monte Carlo

- This class introduces estimation of posterior probability distributions using a stochastic process known as **Markov chain Monte Carlo** (MCMC).
- Here we'll produce samples from the joint posterior without maximizing anything.
- We will be able to sample directly from the posterior without assuming a Gaussian, or any other, shape.
- The cost of this power is that it may take much longer for our estimation to complete.
- But the benefit is escaping multivariate normality assumption of the Laplace approximation.
- More advanced models such as the generalized linear and multilevel models tend produce non-Gaussian posterior distributions.
- In most cases they cannot be estimated at all with the techniques of earlier classes.
- This class is based on Chapter 9 of [McElreath, 2020] and Chapter 7 of [Kruschke, 2014].

Markov Chain Monte Carlo

- The essence of MCMC is to produce samples from the posterior $f(\theta|d)$ by only accessing a function that is proportial to it.
- This proportial function is the product of the likelihood and the prior $f(d|\theta) * f(\theta)$, which is always available in a Bayesian model.
- So, merely by evaluating $f(d|\theta) * f(\theta)$, without normalizing it by $f(d)$, MCMC allows us to generate random representative values from the posterior distribution.
- This property is wonderful because the method obviates direct computation of the evidence $f(d)$, which, as you'll recall, is one of the most difficult aspects of Bayesian inference.
- It has only been with the development of MCMC algorithms an software that Bayesian inference is applicable to complex data analysis.
- And it has only been with the production of fast and cheap computer hardware that Bayesian inference is accessible to a wide audience.
- The question then becomes this: How does MCMC work? For an answer, let's ask a politician.

A politician stumbles upon the Metropolis algorithm

- Suppose an elected politician lives on a long chain of islands.
- He is constantly traveling from island to island, wanting to stay in the public eye.
- At the end of a day he has to decide whether to:
  1. stay on the current island
  2. move to the adjacent island to the west
  3. move to the adjacent island to east
- His goal is to visit all the islands **proportionally** to their **relative population**.
- But, he doesn't know the total population of all the islands.
- He only knows the population of the current island where he is located.
- He can also ask about the population of an adjacent island to which he plans to move.

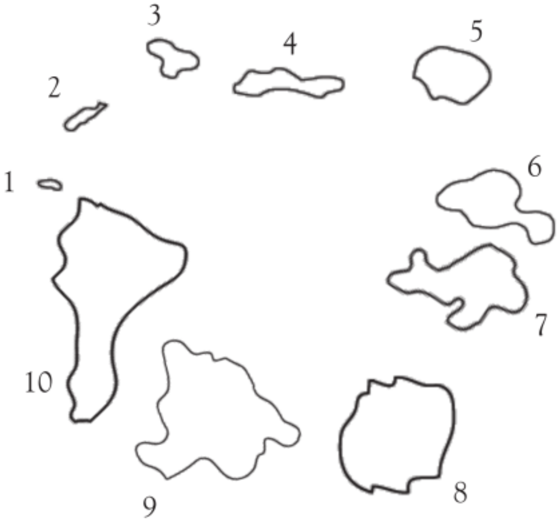
The Metropolis algorithm

- The politician has a simple heuristic for travelling accross the islands called the **Metropolis** algorithm.
- First, he flips a (fair) coin to decide whether to propose the adjacent island to the left or the adjacent island to the right.
- If the proposed island has a larger population than the current island ($P_{proposed} > P_{current}$), then he goes to the proposed island.
- If the proposed island has a smaller population than the current island ($P_{proposed} < P_{current}$), then he goes to the proposed island with probability $p_{move} = P_{proposed}/P_{current}$.
- In the long run, the probability that the politician is on any one of the islands exactly matches the relative population of the island!

The Metropolis algorithm

- Let's analyze the Metropolis algorithm in more detail.
- Suppose there are 10 islands in total.
- Each island is neighbored by two others, and the entire archipelago forms a ring.
- The islands are of different sizes, and so had different sized populations living on them.
- The second island is twice as populous as the first, the third three times as populous as the first.
- And so on, up to the largest island, which is 10 times as populous as the smallest.

The Metropolis algorithm



The Metropolis algorithm

- We are going to show an implementation of this algorithm in R.
- But before that, we will combine combine the two possibilities: the proposed island having a 1) higher or 2) lower population than the current island, into a single expression:

$$p_{move} = \min(1, P_{proposed}/P_{current}). \tag{1}$$

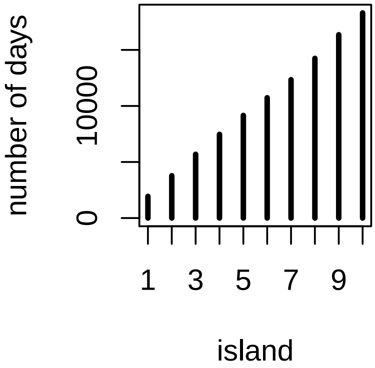
- So, if $P_{proposed} > P_{current}$, $P_{proposed}/P_{current} > 1$ and $p_{move} = 1$.
- For example, *current* $= 4$ and *proposed* $= 5$, $5/4 > 1$ so we move to the proposed island.
- On the other hand, if $P_{proposed} < P_{current}$, $P_{proposed}/P_{current} < 1$, and $p_{move} = P_{proposed}/P_{current}$.
- For example, *current* $= 4$ and *proposed* $= 3$, $3/4 < 1$ so we move to the proposed island with probability $3/4$.

The Metropolis algorithm

```
num_days <- 1e5
positions <- rep(0,num_days)
current <- 10
for ( i in 1:num_days ) {
  # record current position
  positions[i] <- current
  # flip coin to generate proposal
  proposal <- current + sample( c(-1,1) , size=1 )
  # now make sure he loops around the archipelago
  if ( proposal < 1 ) proposal <- 10
  if ( proposal > 10 ) proposal <- 1
  # move?
  prob_move <- min(proposal/current,1)
  decision <- rbinom(1,1,prob_move)
  current <- ifelse( decision == 1 , proposal , current )
}

library(rethinking)
simplehist(positions,xlab="island",ylab="number of days")
```

The Metropolis algorithm



The time spent on each island is proportional to its population size.

The Metropolis algorithm

- The first three lines of the method just define the number of days to simulate, an empty history vector, and a starting island position (the biggest island, number 10).
- Then the for loop steps through the days.
- Each day, it records the politician's current position.
- Then it simulates a coin flip to nominate a proposal island.
- The only trick here lies in making sure that a proposal of "11" loops around to island 1 and a proposal of "0" loops around to island 10.
- Finally, a random binary number is generated with a binomial distribution with probability of success (or moving)$= \mathrm{min}(1, P_{proposed}/P_{current})$
- If this random number is 1 we move, otherwise we stay.

The Metropolis algorithm

- In real applications, the goal is of course not to help a politian, but instead to draw samples from an unknown and usually complex posterior probability distribution.
- The "islands" in our objective are parameter values θ, and they need not be discrete, but can instead take on a continuous range of values as usual.
- The "population sizes" in our objective are the posterior probabilities (or densities) at each parameter value: $f(\theta|d)$
- The "days" in our objective are samples taken from the posterior distribution.
- The Metropolis algorithm will eventually give us a collection of samples from the posterior.
- We can then use these samples just like all the samples we have already used in this course.

Why it works

- Now, let's try to understand the mathematics behind why the algorithm works.
- Consider two adjacent positions and the probabilities of moving from one to the other.
- We'll see that the relative transition probabilities, between adjacent positions, exactly match the relative values of the target distribution.
- Extrapolate that result across all the positions, and you can see that, in the long run, each position will be visited proportionally to its target value.
- Suppose we are at position θ.
- The probability of moving to $\theta + 1$, denoted $P(\theta \rightarrow \theta + 1)$, is the probability of proposing that move times the probability of accepting it if proposed, which is:

$$P(\theta \rightarrow \theta + 1) = 0.5 \times \min(P(\theta + 1)/P(\theta), 1)$$

Conclusions

- Blabla

References I

Kruschke, J. (2014).
Doing bayesian data analysis: A tutorial with r, jags, and stan.

McElreath, R. (2020).
Statistical rethinking: A Bayesian course with examples in R and Stan.
CRC press.