Markov Chain Monte Carlo Methods

2023-11-01

MC and MC

Markov Chain: A stepwise process of state transition where each transition depends only on the current state

> Note: Markov chains are **memoryless** - they don't know where they came from - so each new step is independent of the last step.

Monte Carlo:

> Repeated random sampling to describe stochastic processes (or really really complicated deterministic processes)

Markov Chain Monte Carlo

> Using **repeated random sampling** to **model chains of independent events**

Example: Random Walk Models

The random walk - or, more evocatively and less sensitively, the drunkard's walk - is a classic example of a Markov Model.

Random walk models are used to model cases where things move from one state to another with randomness - it's *random* (or *drunk*) because prior states have no bearing on future steps.

> Example: Let's think of a case of a gambler who starts with $\$10$. She bets $\$1$ per game with a 49% chance of winning and a 51% chance of losing.

The expected value of this gamble is -2 cents per game. If she plays 100 games, she can expect to lose $\$2$.

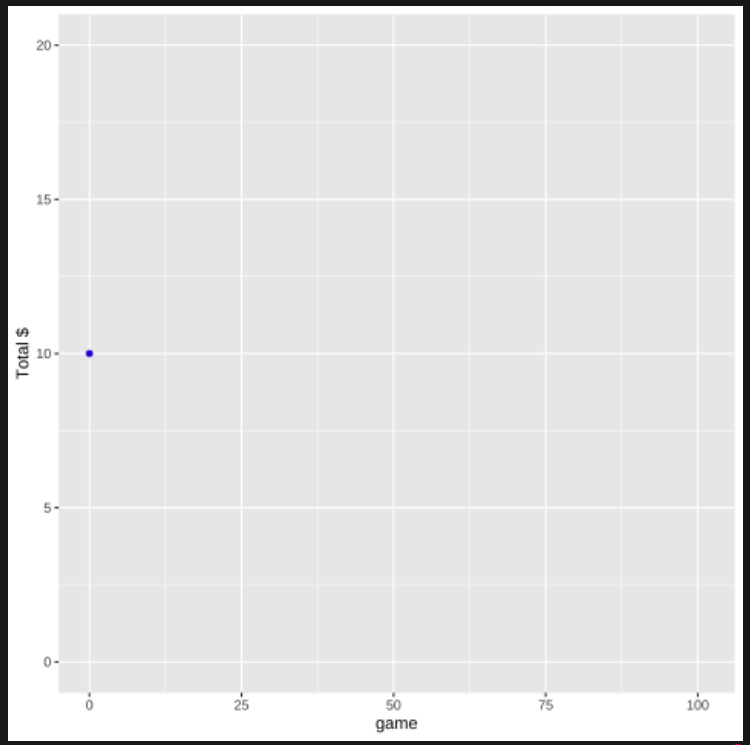
But that's not how gambling works! Nobody would play a game where you just repeatedly handed over 2 cents.

Example: Random Walk Models

The course of a game - and the reason our gambler friend might want to play - is better explained with a random walk model than with a gently downward-sloping line.

Her endowment starts at $\$10$. If she wins her first game $(p = 0.49)$, she will have $\$11$. Her next step is *independent* of what's come before!

If our friend will quit the game when she is either out of money or up $\$10$, the model looks like this:



Sampling Random Numbers

Random number generation is common across coding languages.

> **Example**: samling from a uniform distribution:

language	command
Basic	RND
C++	rand()
Python	random()
Excel	randbetween()
R	r + distribution + ()

Random Numbers in R

distribution	base command	shape parameters	default shape values
Uniform	`runif(n, ...)`	min, max	0, 1
Normal	`rnorm(n, ...)`	mean, sd	0, 1
Binomial	`rbinom(n, ...)`	size, π	-
χ^2	`rchisq(n, ...)`	df	-
beta	`rbeta(n, ...)`	alpha, beta	-
t	`rt(n, ...)`	df	-
Poisson	`rpois(n, ...)`	λ	-

n is the number of random samples you want (note that it's common to all the base commands)

The shape parameters are the sufficient statistics for the named distribution

Example: Simulating Coin Flips

To start, let's create an basic simulation program.

This program will simulate 100 flips of a fair coin.

(he's marginally less scary once you notice that he's terrible at flipping a coin, right?)

Example: Simulating Coin Flips

> Step 1: set a seed

The random numbers we're going to generate aren't *purely* random.

They come from a **pseudorandom algorithm** -- setting a seed specifies the **starting point** of that algorithm

Setting a seed (which can be any number you want) ensures that you use the *same* algorithm each time for **reproducible results**.

Pictured: a so-called random number generator exposed for the pseurorandom number generator that it is

```
set.seed(77) # Set seed for reproducibility.
```

Example: Simulating Coin Flips

> Step 2: Create an empty vector to hold our results

Our coin-flipping machine is going to use a `for` loop.

> `for (this many times) {do this thing}`

R can get a little laggy if it doesn't have a place to put the things that are created in a `for` loop, so we're going to give those things a home.

In this case, we are going to make an empty vector:

> `{NA, NA, NA, NA, ...}`

to hold the results of each of our 100 coin flips.

```
headsortails <- rep(NA, 100) # make an empty vector to store results
```

Example: Simulating Coin Flips

> Step 3: Write our `for` loop

The parentheses `()` after `for` specify how many times we are going to do the thing, with `i` as an index for which time we're on.

The work that the loop goes goes in the curly brackets `{}`

```
for (i in 1:100){ # we're gonna do this 100 times
  flip<-runif(1)  # generate 1 random number # between 0 and 1*
    headsortails[i]<-ifelse(flip<=0.5, 0, 1)
    # if the random number is less than or equal to
    # 0.5 we have tails, otherwise we have heads
}
```

NOTE: we don't have to specify "between 0 and 1" because those are the `runif()` defaults

Example: Simulating Coin Flips

> Let's run the code!

```
set.seed(1)

headsortails<-rep(NA, 100)

for (i in 1:100){
  flip<-runif(1)
  headsortails[i]<-ifelse(flip<=0.5, 0, 1)
}

table(headsortails) # show output
```

```
## headsortails
##  0  1
## 52 48
```

We got 52 *tails* and 48 *heads*. That is reasonable.

The Metropolis-Hastings Algorithm

> Metropolis, Rosenbluth, Rosenbluth, Teller, & Teller (1953); Hastings (1970)

A flexible MCMC method that generates samples from a posterior distribution that approximates the target distribution

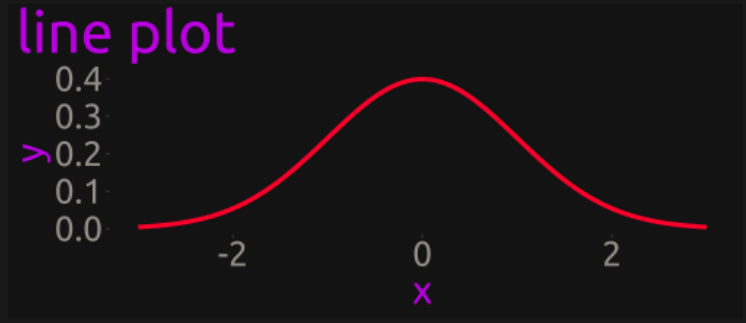
> *e.g.* for a binomial experiment, the target is a beta with $(\alpha = s' + 1, \beta = f' + 1)$

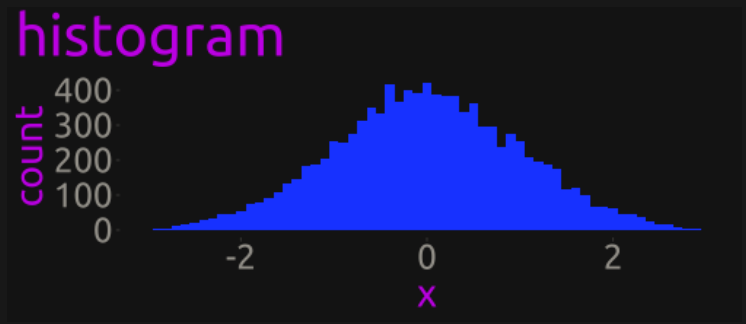
Other MCMC methods - most famously, the Gibbs Sampler - tend to be more efficient but require more assumptions to be met.

MH Algorithm Logic

A distribution can be represented by a theoretical *formula*:

$$f(x|\mu,\sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$





Or estimated by a **big vector** of **simulated data** from that distribution:

```
x<-rnorm(10000)
```

We're going to create a big vector, **one number at a time**.

MH Algorithm Logic: Overview

We are going to start with a literal guess of what the parameter is for the first number in the vector

> it helps if it's a good guess.

Then we're going to make another guess.

If our second guess is as good or better*, we'll take it. If our second guess is worse, we might take it anyway.

That's the first number in the vector. Then we move on to the next one.

> and the next one, and the next one.

*we'll get to how we decide what is *better* soon

MH Algorithm Logic

> How do we evaluate our guesses?
>
> > We use **Bayes's Theorem**.

Each of our guesses represents a **hypothesis**.

If our first guess is H_1 (**Hypothesis 1**) and our second guess is H_2 (**Hypothesis 2**), then we are going to compare the posterior probabilities by taking the **ratio** (r):

$$r = \frac{p(H_2|D)}{p(H_1|D)}$$

MH Algorithm Logic

- How do we evaluate our guesses?

The base rate $p(D)$ is going to be the same for both hypotheses, so that cancels out, leaving us with:

$$r = \frac{p(H_2)p(D|H_2)}{p(H_1)p(D|H_1)}$$

It's even simpler than that if - as we usually do - we sample both guesses at random from the *same* prior distribution. If that's the case, then $p(H_2) = p(H_1)$, so:

$$r = \frac{p(D|H_2)}{p(D|H_1)}$$

which is the ratio of the likelihoods of H_2 and H_1.

MH Algorithm Logic

> Obviously we want good guesses. *Why do we want bad ones, too?*

If we only add values to the distribution that increase the likelihood, then we will either:

> get stuck at the mode of the distribution
>
> get stuck at a local peak of the distribution

Curves have low points, too, and those are important!

And - this is the most important part - choosing *lower-likelihood* values according to a ratio based on Bayes's Theorem is the key to producing a posterior distribution.

The Metropolis-Hastings Algorithm

1. Choose a starting parameter value. Call this the current value.
2. Generate a proposed parameter value from the prior distribution
3. Calculate the ratio r between the likelihood of the observed data given the proposed value to the and the likelihood of the observed data given the current value.
4. If $r \geq 1$, accept the proposed parameter.
    - If $r > u(0, 1)$ (where u represents the *uniform distribution*), accept the proposed parameter.
    - else, accept the current parameter.
5. Store the accepted parameter. That becomes the current parameter for the next iteration.
6. Repeat steps 1 - 5 as many times as you like (but a lot of times).

The Metropolis-Hastings Algorithm

r is the ratio of two posterior probabilities.

> the base rate cancels out and the priors usually do too, simplifying to a likelihood ratio

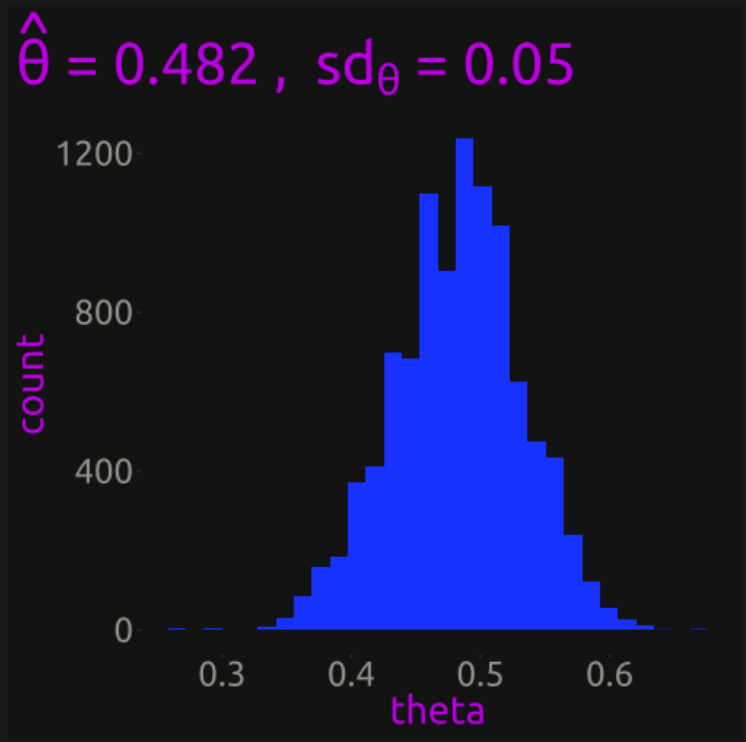
$U(0, 1)$ is just a random number from the uniform distribution between 0 and 1.

> If $r < 1$, the proposed parameter will be accepted with a probability exactly equal to r.

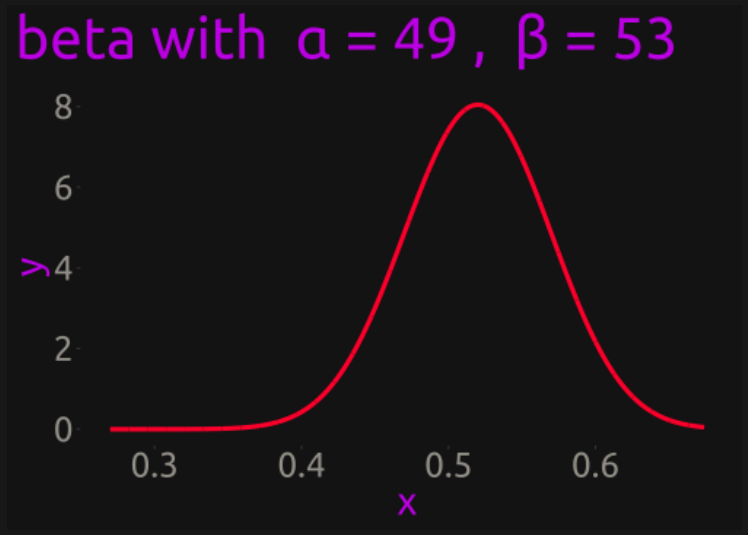
Burn-in and autocorrelation are relatively minor problems that can be handled by selectively dropping or including observations

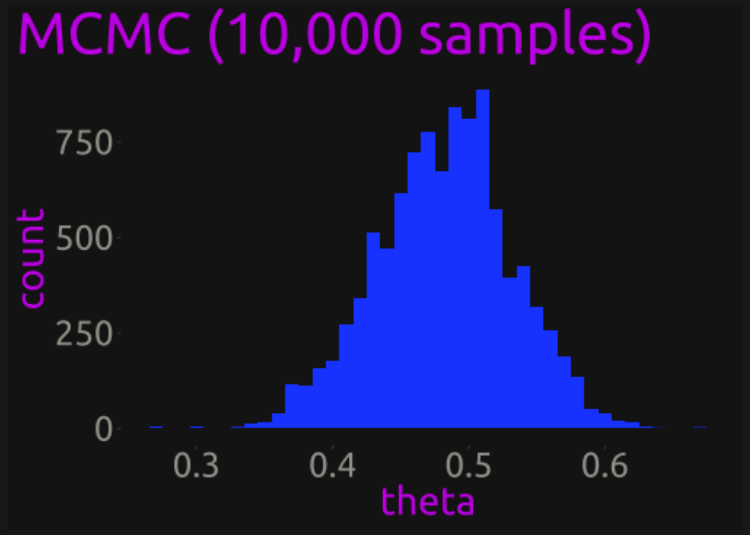
MH Algorithm: Estimating θ_{heads}

```
thetastart<-0.5
theta<-c(thetastart, rep(NA,
9999))

n1<-sum(headsortails==1)
n2<-sum(headsortails==0)

for (i in 2:10000){
  theta2<-runif(1)
  u<-runif(1)
  r<-dbinom(n1, n1+n2, theta2)/
    dbinom(n1, n1+n2, theta[i-1])
  theta[i]<-ifelse(r>=1, theta2,
                   ifelse(r>=u,
theta2, theta[i-1]))
}
```



MH Algorithm: Estimating θ_{heads}





$mean(\text{Beta}) = 0.48$

$sd(\text{Beta}) = 0.05$

`mean(theta)` $= 0.48$

`sd(theta)` $= 0.05$

MH Example: Multinomial Models in Psychology

Multinomial Models use probability to describe the relationship between:

- *latent processes*
  - *e.g.*, memory, perception, bias, preferences

and

- *observable phenomena*
  - e.g., behaviors, choices, physiological responses.

MH Example: Multinomial Models in Psychology

Sometimes multiple processes must co-occur to produce an outcome

- *e.g.*, recall = storage + retrieval

Sometimes different process combinations can produce the same outcome

- *e.g.*, *recognizing* and *guessing correctly* can look a lot alike.

MH Example: Multinomial Models in Psychology

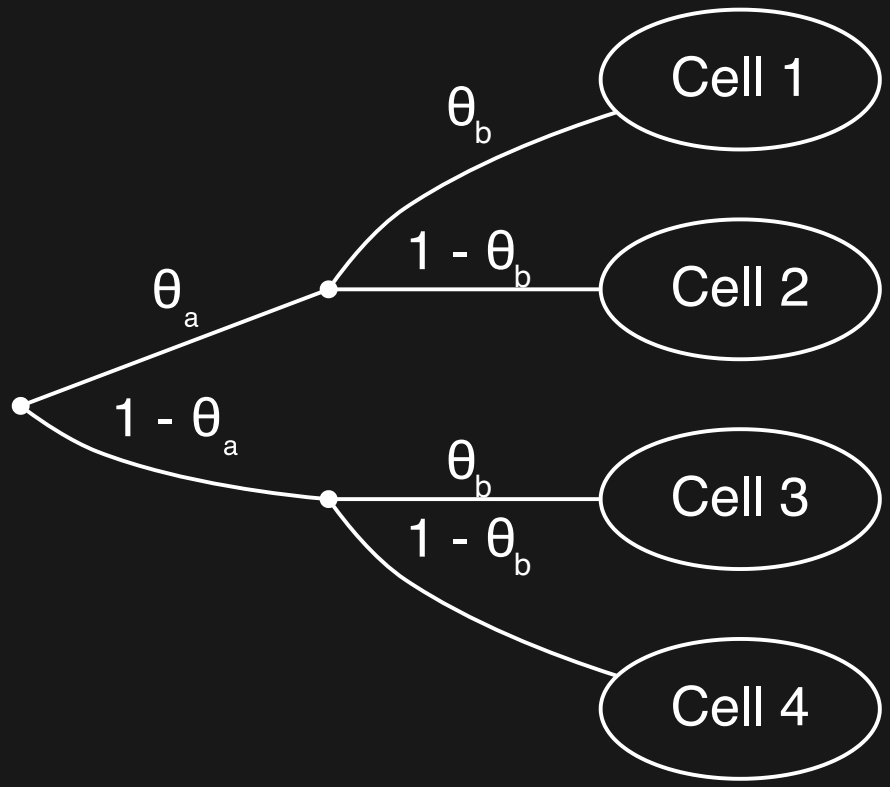
Multinomial models (sometimes called multinomial process tree or MPT models) are built on theories that can be tested with experimental design with categorical outcomes:

- what combination(s) of *processes* lead to *outcomes*?

MCMC methods are one technique to assess the probabilities of different processes given observable phenomena (*i.e.*, $p(H|D)$)

- same basic idea as estimating θ_{heads} from observed outcomes

A Relatively* Basic Tree

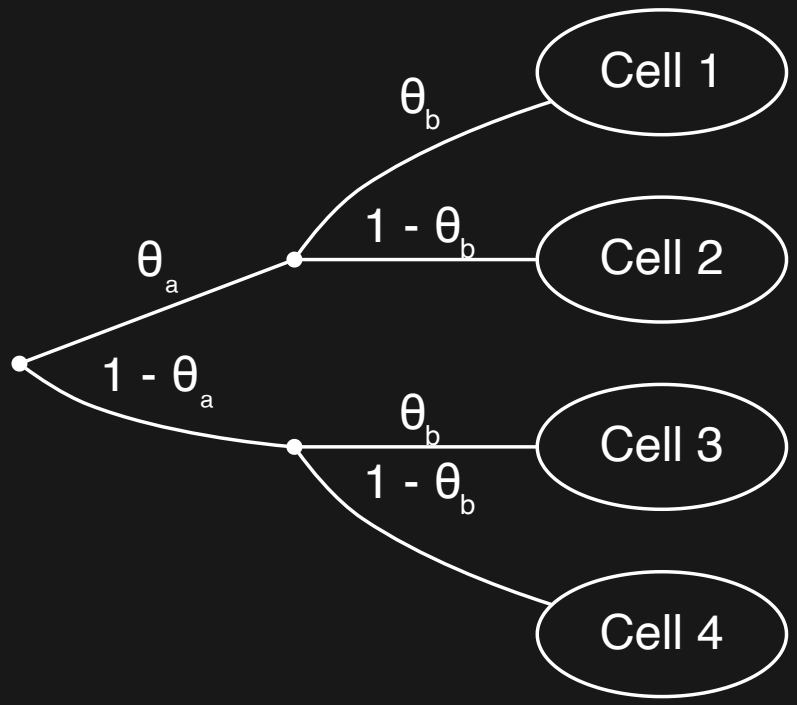


*they can get *much* more complicated than this.

A Relatively Basic Tree

The *cells* represent different, categorical *behaviors*.

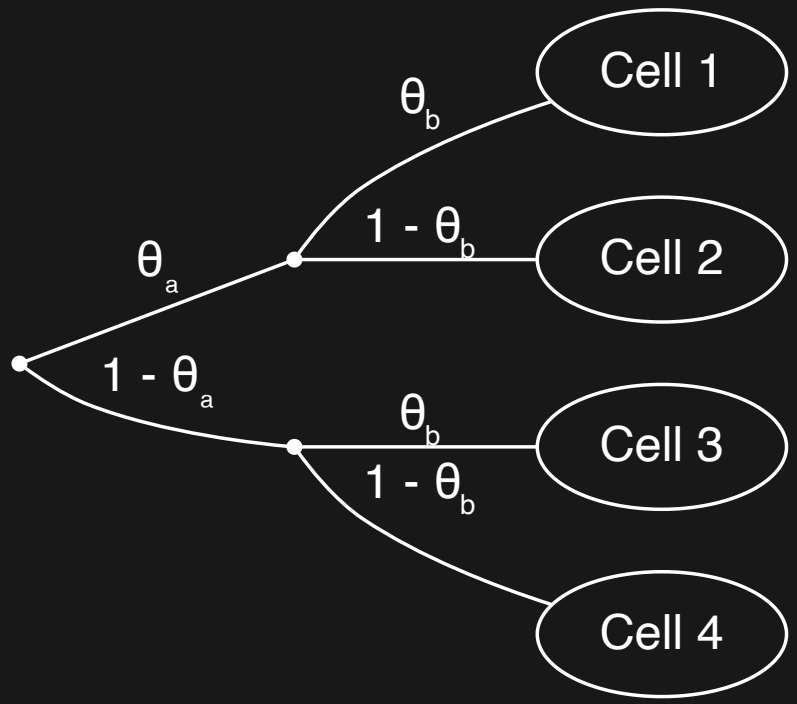
> *Behavior 1* happens with probability $\theta_a\theta_b$
>
> *Behavior 2:* $\theta_a(1-\theta_b)$
>
> *Behavior 3:* $(1-\theta_a)(\theta_b)$
>
> *Behavior 4:* $(1-\theta_a)(1-\theta_b)$



A Relatively Basic Tree

Just as we did when we constructed the binomial likelihood function, we are going to construct a likelihood function for this tree.

And, as with the binomial likelihood function, our new likelihood function will have a combinatorial term and a kernel probability term.

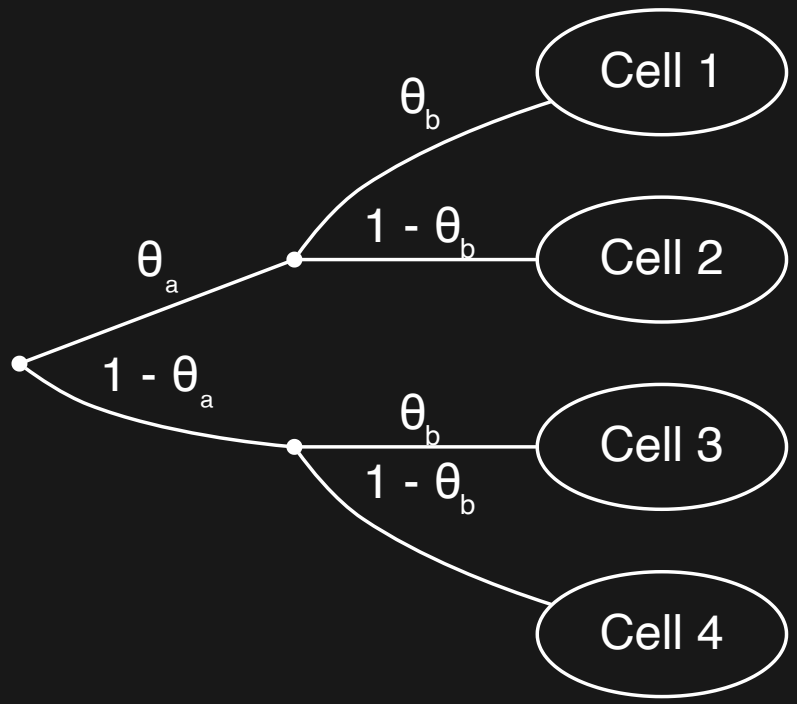


A Relatively Basic Tree

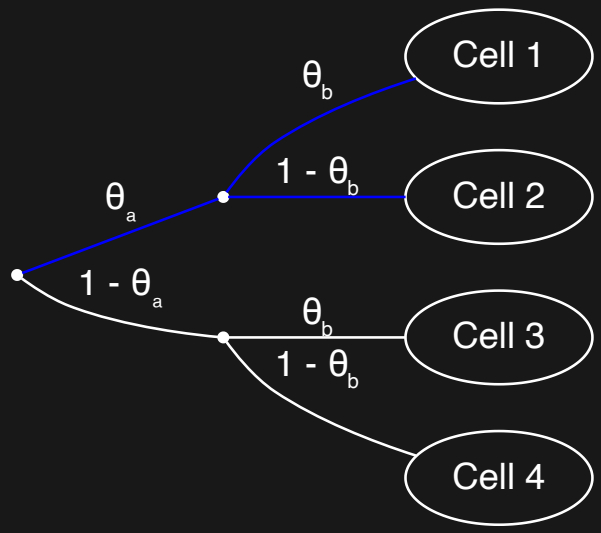
Combinatorial Term

If we have N total trials, and $n_1, \ldots, n_4$ are the number of outcomes for cell 1, ... cell 4, then we are *combining* N things n_1, n_2 , and n_3 things at a time, with n_4 left over:

$$\frac{N!}{n_1!n_2!n_3!n_4!}$$



A Relatively Basic Tree



Kernel Probability Term

The **kernel probability** is the product of each probability $\theta_a, \theta_b, (1-\theta_a)$, and $(1-\theta_b)$, each raised to the power of the cells they lead to.

For example, θ_a is on the path to cells 1 and 2 (highlighted in blue), so the kernel probability includes the term $\theta_a^{n_1+n_2}$

The full **kernel probability** term is:

$$\theta_a^{n_1+n_2}\theta_b^{n_1+n_3}(1-\theta_a)^{n_3+n_4}(1-\theta_b)^{n_2+n_4}$$

MCMC Coding

Thus, our full likelihood function is:

$$p(D|N,\theta_1,\theta_2) = \frac{N!}{n_1!n_2!n_3!n_4!}\theta_a^{n_1+n_2}\theta_b^{n_1+n_3}(1-\theta_a)^{n_3+n_4}(1-\theta_b)^{n_2+n_4}$$

We're going to code that as a function so we can use it repeatedly:

```
likely<-function(n1, n2, n3, n4, theta.a, theta.b){
  L<-
(factorial(n1+n2+n3+n4)/(factorial(n1)*factorial(n2)*factorial(n3)*factorial(
    (theta.a^(n1+n2))*
    (theta.b^(n1+n3))*
    ((1-theta.a)^(n3+n4))*
    ((1-theta.b)^(n2+n4))
  return(L)
}
```

MCMC Coding

Let's run our function and see what we get.

First, we will set up the number of MCMC iterations we want (`iterations`), our starting guesses for θ_a and θ_b (both 0.5), and enter the observed data (n_1, n_2, n_3, n_4)....

```
iterations<-1000000

theta.a<-c(0.5, rep(NA, iterations-1))
theta.b<-c(0.5, rep(NA, iterations-1))

n1=28
n2=12
n3=42
n4=18
```

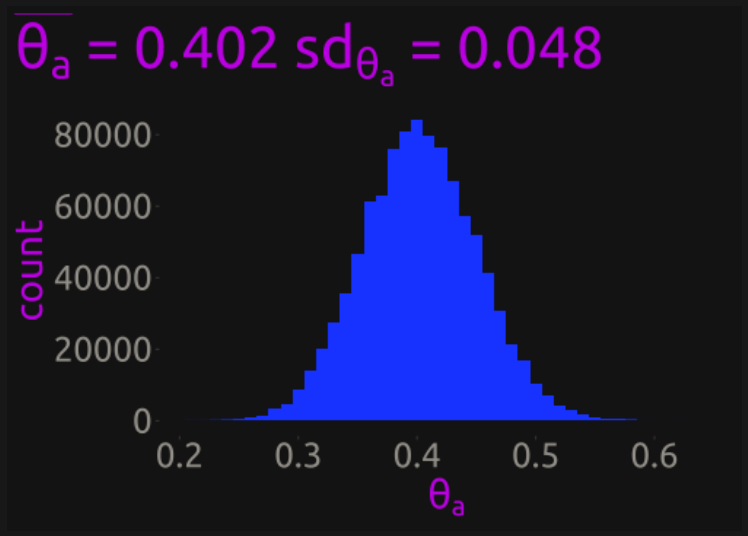
MCMC Coding

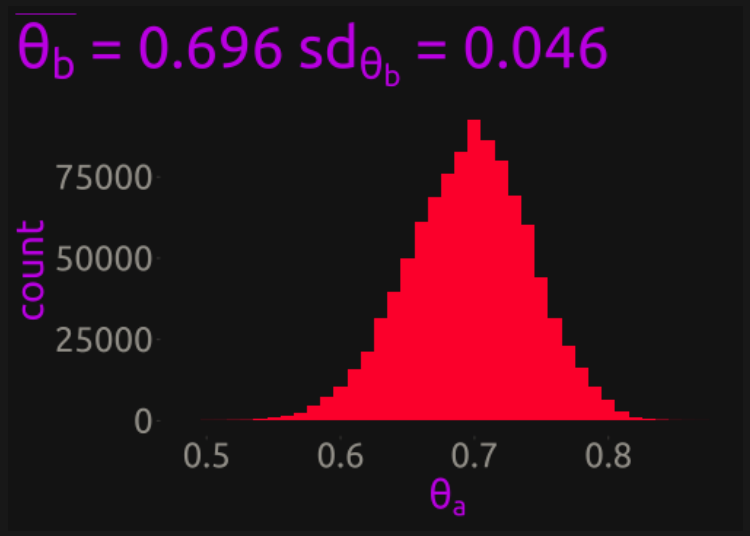
...and then we will run our code with the `likely()` function we wrote earlier to get posterior distributions for θ_a and θ_b.

```
for (i in 2:iterations){
  theta.a2<-runif(1)
  theta.b2<-runif(1)
  r<-likely(n1, n2, n3, n4, theta.a2, theta.b2)/likely(n1, n2, n3, n4,
theta.a[i-1], theta.b[i-1])
  u<-runif(1)
  if (r>=1){
  theta.a[i]<-theta.a2
  theta.b[i]<-theta.b2
  } else if (r>u){
  theta.a[i]<-theta.a2
  theta.b[i]<-theta.b2
  } else {
    theta.a[i]<-theta.a[i-1]
    theta.b[i]<-theta.b[i-1]
  }}
```

MCMC Results

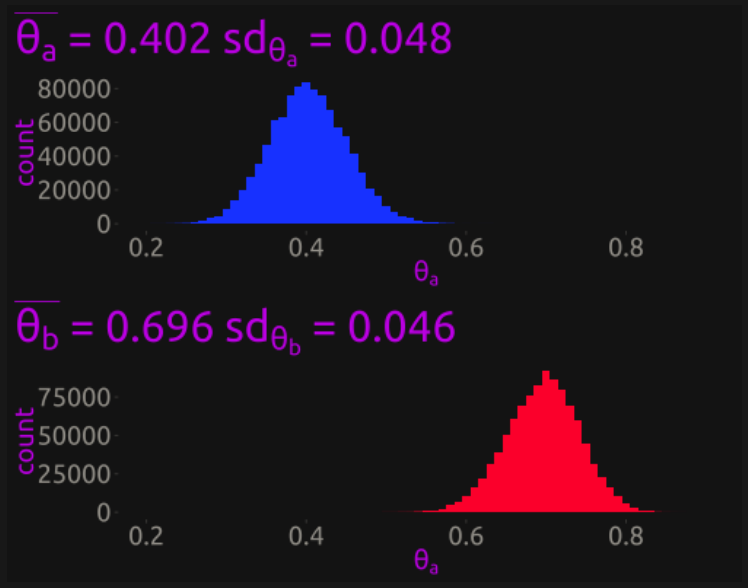
And here are the distributions!





With these distributions, we can then make inferences regarding θ_a and θ_b.

MCMC Results



95% HDIs

```
quantile(theta.a, c(0.025, 0.975))
```

```
##      2.5%     97.5%
## 0.3092269 0.4981928
```

```
quantile(theta.b, c(0.025, 0.975))
```

```
##      2.5%     97.5%
## 0.6033346 0.7826061
```

$p(\theta_a > \theta_b)$

```
sum(theta.a > theta.b)/length(theta.a)
```

```
## [1] 1e-06
```