Simulation - part 2

Stat 133

Gaston Sanchez

Department of Statistics, UC–Berkeley

gastonsanchez.com
github.com/gastonstat

Course web: gastonsanchez.com/stat133

Random Numbers in R

Generation of random numbers is at the heart of many statistical methods

Use of Random Numbers

Some uses of random numbers

- Sampling procedures
- Simulation studies of stochastic processes
- Analytically intractable mathematical expressions
- Simulation of a population distribution by resampling from a given sample from that population
- More general: Simulation, Monte Carlo, Resampling

Random Samples

- Many statistical methods rely on random samples:
  - Sampling techniques
  - Design of experiments
  - Surveys
- Hence, we need a source of random numbers
- Before computers, statisticians used *tables of random numbers*
- Now we use computers to generate “random” numbers
- The random sampling required in most analyses is usually done by the computer

Generating Random Numbers

- We cannot generate truly random numbers on a computer
- Instead, we generate **pseudo-random** numbers
- i.e. numbers that have the appearance of random numbers
- they *seem* to be randomly drawn from some known distribution
- There are many methods that have been proposed to generate pseudo-random numbers

A very important advantage of using pseudo-random numbers is that, because they are deterministic, they can be reproduced (i.e. repeated)

Multiple Recursion

- Generate a sequence of numbers in a manner that appears to be random
- Use a deterministic generator that yields numbers recursively (in a fixed sequence)
- The previous k numbers determine the next one

$$x_i = f(x_{i-1}, \ldots, x_{i-k})$$

Simple Congruential Generator

- Congruential generators were the first reasonable class of pseudo-random number generators
- The congruential method uses modular arithmetic to generate “random” numbers

Ingredients

- An integer m
- An integer a such that $a < m$
- A starting integer x_0 (a.k.a. *seed*)

Simple Congruential Generator

The first number is obtained as:
$x_1 = (a \times x_0) \bmod m$

The rest of the pseudorandom numbers are generated as:
$x_{n+1} = (a \times x_n) \bmod m$

Simple Congruential Generator

For example if we take $m = 64$, and $a = 3$, then for $x_0 = 17$ we have:

$$
\begin{aligned}
x_1 &= (3 \times 17) \text{ mod } 64 = 51 \\
x_2 &= (3 \times 51) \text{ mod } 64 = 25 \\
x_3 &= (3 \times 25) \text{ mod } 64 = 11 \\
x_4 &= (3 \times 11) \text{ mod } 64 = 33 \\
x_5 &= (3 \times 33) \text{ mod } 64 = 35 \\
x_6 &= (3 \times 35) \text{ mod } 64 = 41 \\
&\vdots
\end{aligned}
$$

Congruential algorithm

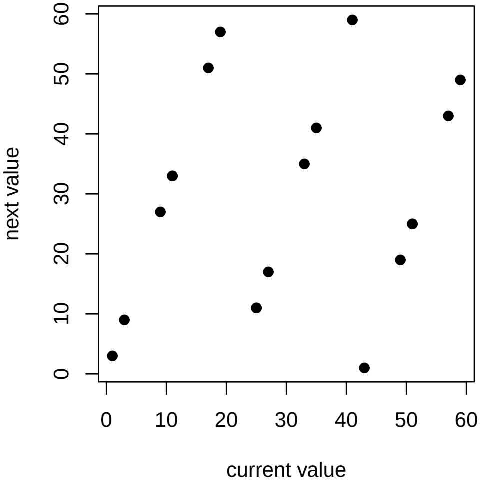
```r
a <- 3;  m <- 64;  seed <- 17

x <- numeric(20)

x[1] <- (a * seed) %% m

for (i in 2:20) {
  x[i] <- (a * x[i-1]) %% m
}

x[1:16]; x[17:20]

##  [1] 51 25 11 33 35 41 59 49 19 57 43  1  3  9 27 17
## [1] 51 25 11 33
```

Congruential algorithm

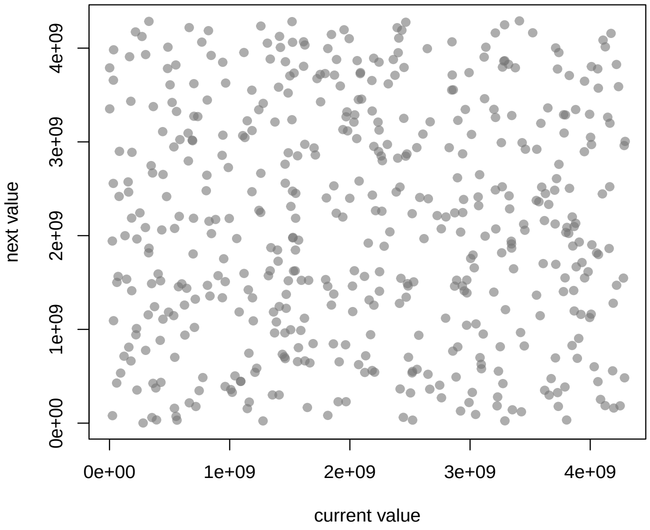
```
cong <- function(n, a = 69069, m = 2^32, seed = 17) {
  x <- numeric(n)
  x[1] <- (a * seed) %% m
  for (i in 2:n) {
    x[i] <- (a * x[i-1]) %% m
  }
  x
}

y <- cong(20, a = 3, m = 64, seed = 17)
```

Congruential algorithm

```
plot(y[1:(n-1)], y[2:n], pch = 19,
     xlab = 'current value', ylab = 'next value')
```



`cong(n, a = 69069, m = 2^32, seed = 17)`



Ingredients

- m is the modulus $(0 < m)$
- a is the multiplier $(0 < a < m)$
- x_0 is the seed $(0 \leq x_0 \leq m)$

The period length of a Random Generator Number is at most m, and for some choices of a much less than that.

About the seed

- You can reproduce your simulation results by controlling the seed
- `set.seed()` allows to do this
- Every time you perform simulation studies indicate the random number generator and the seed that was used

About the seed

```
# set the seed
set.seed(69069)

runif(3)  # call the uniform RNG

## [1] 0.1648855 0.9564664 0.3345479

runif(3)  # call runif() again

## [1] 0.01109596 0.18654873 0.94657805

# set the seed back to 69069
set.seed(69069)
runif(3)

## [1] 0.1648855 0.9564664 0.3345479
```

Simple Congruential Generator

We can use a congruential algorithm to generate uniform numbers

We'll describe one of the simplest methods for simulating independent uniform random variables on the interaval [0, 1]

Generating Uniform Numbers

The generator proceeds as follows

$$x_1 = (ax_0) \text{ mod } m$$
$$u_1 = x_1/m$$

u_1 is the first pseudo-random number, taking some value between 0 and 1.

Ingredients

The second pseudorandom number is obtained as:

$$x_2 = (ax_1) \text{ mod } m$$
$$u_2 = x_2/m$$

u_2 is another pseudorandom number.

Generating Uniform Numbers

- If m and a are chosen properly, it is difficult to predict the value u_2 given u_1
- For most practical purposes u_2 is approximately independent of u_1

Simple Congruential Generator

For example if we take $m = 7$, and $a = 3$, then for $x_0 = 2$ we have:

$$
\begin{aligned}
x_1 &= (3 \times 2) \text{ mod } 7 = 6, \quad & u_1 &= 0.857 \\
x_2 &= (3 \times 6) \text{ mod } 7 = 4, \quad & u_2 &= 0.571 \\
x_3 &= (3 \times 4) \text{ mod } 7 = 5, \quad & u_3 &= 0.714 \\
x_4 &= (3 \times 5) \text{ mod } 7 = 1, \quad & u_4 &= 0.143 \\
x_5 &= (3 \times 1) \text{ mod } 7 = 3, \quad & u_5 &= 0.429 \\
x_6 &= (3 \times 3) \text{ mod } 7 = 2, \quad & u_6 &= 0.286 \\
&\vdots
\end{aligned}
$$

Random Numbers in R

R uses a pseudo random number generator

- It starts with a **seed** and an **algorithm** (i.e. function)
- The seed is plugged into the algorithm and a number is returned
- That number is then plugged into the algorithm and the next number is created
- The algorithm is such that the produced numbers behave like random numbers

RNG functions in R

Function	Description
`runif()`	Uniform
`rbinom()`	Binomial
`rmultinom()`	Multinomial
`rnbinom()`	Negative binomial
`rpois()`	Poisson
`rnorm()`	Normal
`rbeta()`	Beta
`rgamma()`	Gamma
`rchisq()`	Chi-squared
`rcauchy()`	Cauchy

See more info: `?Distributions`

Random Number Functions

`runif(n, min = 0, max = 1)` sample from the uniform distribution on the interval (0,1)

The chance the value drawn is:

- between 0 and $1/3$ has chance $1/3$
- between $1/3$ and $1/2$ has chance $1/6$
- between $9/10$ and 1 has chance $1/10$

Random Number Functions

`rnorm(n, mean = 0, sd = 1)` sample from the normal distribution with center = `mean` and spread = `sd`

Random Number Functions

`rnorm(n, mean = 0, sd = 1)` sample from the normal distribution with center = `mean` and spread = `sd`

`rbinom(n, size, prob)` sample from the binomial distribution with number of trials = `size` and chance of success = `prob`

More Simulations

Simulation Probability examples

For instance

- Simulate flipping a coin
- Simulate rolling a die
- Simulate drawing a card from a deck
- Simulate a probability experiment with balls in an urn
- Simulate the "Monty Hall Problem"

Flipping a Coin

Simulating a Coin

How to simulate tossing a coin?

Simulating a coin

One way to simulate a coin

```
coin <- c("heads", "tails")
```

Simulating a coin

One way to simulate a coin

```
coin <- c("heads", "tails")
```

One way to simulate flipping a coin

```
sample(coin, size = 1)

## [1] "heads"
```

Probability

Probability allows us to quantify statements about the chance of an event taking place

For example, flip a fair coin

- What's the chance it lands heads?
- Flip it 4 times, what proportion of heads do you expect?
- Will you get exactly that proportion?
- What happens when you flip the coin 1000 times?

Simulating a Coin

When you flip a coin

- it may land heads
- it may land tails
- with what probability it lands heads?
- If it is a fair coin: $p = 0.5$

Simulating a Coin

Tossing a coin can be modeled with a random variable following a Bernoulli distribution:

- heads ($X = 1$) with probability p
- tails ($X = 0$) with probability $q = 1 - p$

The Bernoulli distribution is a special case of the Binomial distribution: $B(1, p)$

Simulating tossing a coin

Tossing a coin simulated with a **binomial** distribution:

```
# binomial distribution generator
rbinom(n = 1, size = 1, prob = 0.5)

## [1] 0
```

Flipping a Coin function

Function that simulates flipping a coin

```
# flipping coin function
coin <- function(prob = 0.5) {
  rbinom(n = 1, size = 1, prob = prob)
}

coin()

## [1] 1
```

Flipping a Coin function

It's better if we assign labels

```
# flipping coin function
coin <- function(prob = 0.5) {
  out <- rbinom(n = 1, size = 1, prob = prob)
  ifelse(out, "heads", "tails")
}

coin()

## [1] "tails"
```

Flipping a Coin function

```r
# 10 flips
for (i in 1:10) {
  print(coin())
}

## [1] "heads"
## [1] "tails"
## [1] "tails"
## [1] "heads"
## [1] "tails"
## [1] "tails"
## [1] "tails"
## [1] "tails"
## [1] "tails"
## [1] "heads"
```

4 Flips

In 4 flips

- Possible outputs:
  - HHHH, THHH, HTHH, HHTH, HHHT, ...
- we can get 0, 1, 2, 3, 4 heads
- so the proportion of heads can be: 0, 0.25, 0.5, 0.75, 1
- we expect the proportion to be 0.5
- but a proportion of 0.25 is also possible

4 Flips

- we can think of the proportion of Heads in 4 flips as a **statistic** because it summarizes data
- this proportion is a random quantity: it takes on 5 possible values, each with some probability
  - $0 \to 1/16$
  - $0.25 \to 4/16$
  - $0.50 \to 8/16$
  - $0.75 \to 4/16$
  - $1.0 \to 1/16$

Simulating flipping n coins

Function that simulates flipping a coin n times (i.e. flipping n coins)

```
# generic function
flip_coins <- function(n = 1, prob = 0.5) {
  out <- rbinom(n = n, size = 1, prob = prob)
  ifelse(out, "heads", "tails")
}

flip_coins(5)

## [1] "tails" "heads" "heads" "tails" "heads"
```

Proportion of Heads

```r
# number of heads
num_heads <- function(x) {
  sum(x == "heads")
}


# proportion of heads
prop_heads <- function(x) {
  num_heads(x) / length(x)
}
```

1000 Flips

- when we flip the coin 1000 times, we can get many different possible proportions of Heads
- 0, 0.001, 0.002, 0.003, ..., 0.999, 1.000
- It's highly unlikely that we would get 0 for the proportion—how unlikely?
- what does the distribution of the porpotion of heads in 1000 flips look like?

1000 Flips

- With some probability theory and math tools we can figure this out
- But we can also get a good idea using simulation
- In our simulation we'll assume that the chance of Heads is 0.5 (i.e. fair coin)
- we can find out what the possible values for the proportion of heads in 1000 flips look like

Flipping coins

```
set.seed(99900)
flips <- flip_coins(1000)

num_heads(flips)

## [1] 494

prop_heads(flips)

## [1] 0.494
```

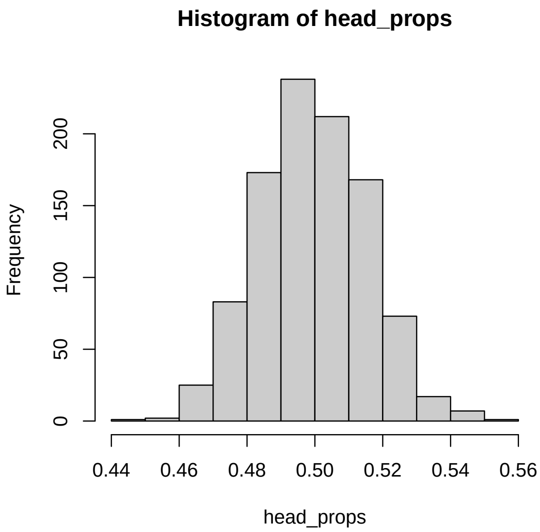
Flipping coins

```
set.seed(76547)
a_flips <- flip_coins(1000)
b_flips <- flip_coins(1000)

num_heads(a_flips)

## [1] 493

num_heads(b_flips)

## [1] 507
```

1000 Flips 1000 times

```
times <- 1000

head_props <- numeric(times)

for (s in 1:times) {
  flips <- flip_coins(1000)
  head_props[s] <- prop_heads(flips)
}
```

Empirical distribution of 1000 flips



Flipping coins

Experiment: flipping a coin 100 times and counting number of heads

You flip a coin 100 times and you get 65 heads. Is it a fair coin?

Flipping coins

Experiment: flipping a coin 100 times and counting number of heads

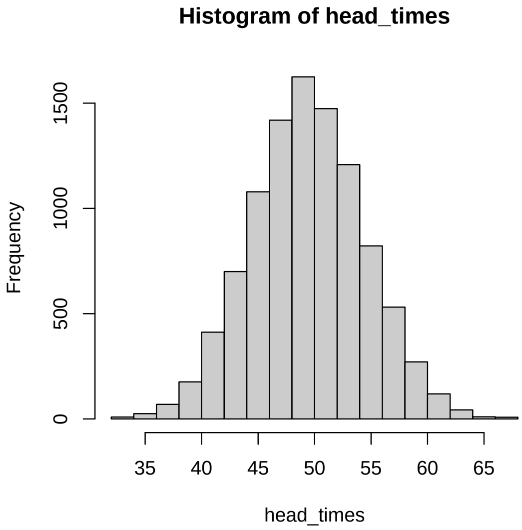
You flip a coin 100 times and you get 65 heads. Is it a fair coin?

We could perform a hypothesis test, or we could perform resampling

Flipping coins

```
# repeat experiment 100 times
times <- 10000
head_times <- numeric(times)
for (s in 1:times) {
  flips <- flip_coins(100)
  head_times[s] <- num_heads(flips)
}

sum(head_times >= 65)

## [1] 18

sum(head_times >= 65) / times

## [1] 0.0018
```

Flipping coins



Bootstrapping

Let's Generalize

- A **statistic** is just a function of a random sample
- Statistics are used as estimators of quantities of interest about the distribution, called **parameters**
- Statistics are random variables
- Parameters are NOT random variables

Let's Generalize

- In simple cases, we can study the *sampling distribution* of the statistic analytically
- e.g. we can prove under mild conditions that the distribution of the sample proportion is close to normal for large sample sizes
- In more complicated cases we can turn to simulation

Sampling Distributions

- In our example $X_1, X_2, \ldots, X_n$ are independent observations from the same distribution
- The distribution has center (mean value) μ and spread (standard deviation) σ
- e.g. interest in the distribution of $median(X_1, X_2, \ldots, X_n)$
- We take many samples of size n, and study the behavior of the sample medians

Some Limitations

- Consider *t-test* procedures for inference about means
- Most classical methods rest on the use of Normal Distributions
- However, most real data are not Normal
- We cannot use t confidence intervals for strongly skewed data (unless samples are large)
- What about inference for a *ratio* of means? (no simple traditional inference)

Fundamental Reasoning

- Apply computer power to relax some of the conditions needed in traditional tests
- Have tools to do inference in new settings
- What would happen if we applied this method many times?

Bootstrap Idea

- Statistical inference is based on the sampling distributions of sample statistics
- A sampling distribution is based on many random samples from the population
- The bootstrap is a way of finding the sampling distribution (approximately)

Bootstrap Samples

```
x <- c(3.15, 0, 1.58, 19.65, 0.23, 2.21)
mean(x)

## [1] 4.47
```

Bootstrap Samples

```r
x <- c(3.15, 0, 1.58, 19.65, 0.23, 2.21)
mean(x)

## [1] 4.47
```

```r
(x1 <- sample(x, size = 6, replace = TRUE))

## [1] 19.65  3.15 19.65  0.00 19.65  2.21

mean(x1)

## [1] 10.71833
```

Bootstrap Samples

```
(x2 <- sample(x, size = 6, replace = TRUE))

## [1] 3.15 0.23 3.15 2.21 2.21 1.58

mean(x2)

## [1] 2.088333

(x3 <- sample(x, size = 6, replace = TRUE))

## [1] 19.65  2.21 19.65  2.21  3.15  1.58

mean(x3)

## [1] 8.075
```

Procedure for Bootstrapping

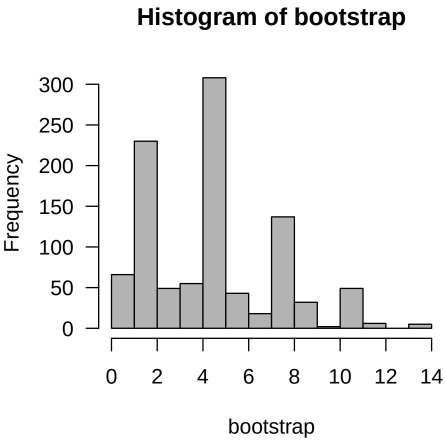
- Repeatedly sampling **with replacement** from a random sample
- Each bootstrap sample is the same size as the original sample
- Calculate the statisc of interest (e.g. mean, median, sd)
- Draw hundreds or thousands of samples
- Obtain a bootstrap distribution

Bootstrap Samples

```r
bootstrap <- numeric(1000)

for (b in 1:1000) {
  boot_sample <- sample(x, size = 6, replace = TRUE)
  bootstrap[b] <- mean(boot_sample)
}
```

Bootstrap Distribution

```r
hist(bootstrap, col = 'gray70', las = 1)
```



How does Bootstrapping work?

- We are not using the resamples as if they were real data
- Bootstrap samples is not a substitute to gather more data to improve accuracy
- The bootstrap idea is to use the resample statistics to estimate how the sample statistic varies from the studied random sample
- The bootstrap distribution approximates the sampling distribution of the statistic

Bootstrap samples

Computing the bootstrap distribution implies

- Calulate the statistic for each sample
- The distribution of these resample statistics is the bootstrap distribution
- A bootstrap sample is the same size as the original random sample

Another Example

Bootstrap resampling

```r
# Iris Virginica subset (the "population")
virginica <- subset(iris, Species == 'virginica')

# random sample of Petal.Length (size = 5)
set.seed(7359)
rand_sample <- sample(virginica$Petal.Length, size = 5)
rand_sample

## [1] 5.1 5.6 5.8 5.7 5.8
```

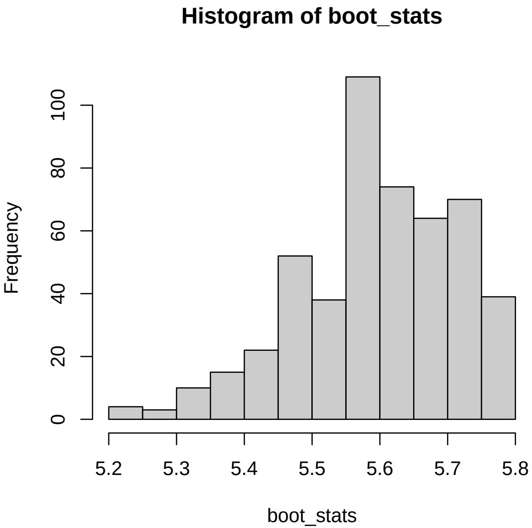
Bootstrap resampling

```r
# create 500 bootstrap samples of size 5 with replacement
resamples <- 500
n <- length(rand_sample)

boot_stats <- numeric(resamples)

for (i in 1:resamples) {
  boot_sample <- sample(rand_sample, size = n, replace = TRUE)
  boot_stats[i] <- mean(boot_sample)
}
```

Bootstrap resampling

```r
# "population" mean
mean(virginica$Petal.Length)

## [1] 5.552

# bootstrap mean
mean(boot_stats)

## [1] 5.60028
```

Bootstrap Distribution



Bootstrap standard error

The bootstrap standard error is just the standard deviation of the bootstrap samples

```
# descriptive statistics
summary(boot_stats)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    5.20    5.52    5.60    5.60    5.70    5.80

# Bootstrap Standard Error
sd(boot_stats)

## [1] 0.1167183
```