

Lab 3: Conditional Probability

Justin Pelak

Spring 2025

Packages needed: None

R Markdown: I will still leave this default text here for this lab! This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

On for-loops and if-else statements for probability estimates in simulation experiments

In the last lab, we used a few slick R functions to simulate experiments and compute empirical probabilities. But we often have to resort to, what I call, “brute-force” for-loops to repeat a simulation experiment and then Boolean expressions or if-else statements to identify successful events in experiments and estimate probabilities. In fact, we did this in Task 5 of the “Introduction to R” lab practical.

Code set-up

Consider the simple experiment of rolling a six-sided die and estimating the probability of rolling a 2. Of course we know the answer is $1/6$ and we can simulate this experiment in R without for-loops. But as an illustration consider the following code chunk.

```
success = 0 # storage vector for whether a roll is a 2 or not
nrolls = 1000 # number of rolls of the die in the simulation experiment
for(i in 1:nrolls){
  roll = sample(1:6, 1) # roll the six-sided die once
  # Two ways to determine if the roll is a 2
  # In each case, we want to store the success result in
  # the ith element of the vector `success'.
  # If roll is a 2, store value of '1' in the success vector;
  # otherwise store a value of zero
  # 1) If-then statement: an if-else syntax for the Boolean expression
  if(roll==2){success[i]=1}else{success[i]=0}
  # 2) Straight Boolean expression
  #success[i] = (roll == 2)
}
mean(success) # proportion of 1s (successes) in the simulation experiment
```

```
## [1] 0.172
```

Notice that we embed a single simulation experiment (roll a die) inside a for-loop to repeat the experiment many times (here 1000 die rolls). After each experiment, we store an indicator of success of the experiment in a vector. So here, the vector `success` is 1000-dimensional and has a value of 1 if a roll is a 2 and a value

of 0 if the roll is not a two. Try running this code for 10 die rolls, and output the vector `success`. Notice that it is a vector of 0s and 1s. If we take an average of the success vector, we are computing the proportion of simulation experiments where a success occurred. So in this example, an empirical probability that a 2 is rolled on a six-sided die.

In each of the following two tasks, you will repeat a simulation experiment by embedding code for a single simulation within a for-loop. Each time, you need to store a success. In the first task we will use an if-else statement. In the second task we will use a Boolean expression.

Task 1: Simulating the birthday problem

The birthday problem considers the probability that two people in a group of a given size have the same birth date. We will assume a 365 day year (no leap year birthdays).

Code set-up

Dobrow 2.40 provides useful R code for simulating the birthday problem. Imagine we want to obtain an empirical estimate of the probability that two people in a class of a given size will have the same birth date. The code

```
trial = sample(1:365, numstudents, replace=TRUE)
```

simulates birthdays from a group of `numstudents` students. So you can assign `numstudents` or just replace `numstudents` with the number of students in the class of interest.

If we store the list of birthdays in the variable `trial`, the code

```
2 %in% table(trial)
```

will create a frequency table of birthdays and then determine if there is a match (2 birthdays the same). We can use this code in an if-else statement to record whether a class has at least one pair of students with the same birth date. We then can embed the code within a for-loop to repeat the experiment, store successes in a vector, and then take the average number of successes (a birthday match) across the repeated tasks.

The problems

- Simulate the birthday problem to obtain an empirical estimate of the probability that two people in a class of 23 will have the same birth date. In particular, simulate birthdays for 1000 classes (`for(i in 1:1000){...}`) each of size 23 and compute the proportion of these classes in which at least one pair of students has the same birth date.

Recall that the true probability is $1 - \prod_{i=1}^{22} (365 - i) / 365^{23}$ which is approximately 50%.

- Using your simulation code, estimate the number of students needed in the class so that the probability of a match is 95%. (You may do this by trial and error.)
- Using your simulation code, find the approximate probability that three people have the same birthday in a class of 50 students.

```
simulation = function(n, students) {
  successes=numeric(1000)
  for(i in 1:1000) {
    trial = sample(1:365, students, replace=T)
    match = n %in% table(trial)
    successes[i] = if (match>=1) 1 else 0
  }
  return(successes)
}
```

```
test1 = simulation(2, 23)
print(sprintf("Probability of two people in 23 having same birth date: %.3f", mean(test1)))

## [1] "Probability of two people in 23 having same birth date: 0.496"

test2 = mean(replicate(10, simulation(2, 48)))
print(sprintf("0.95 probability test: %.3f", mean(test2)))

## [1] "0.95 probability test: 0.955"

test3 = mean(replicate(10, simulation(3, 50)))
print(sprintf("Probability of three people in 50 having same birth date: %.3f", mean(test3)))

## [1] "Probability of three people in 50 having same birth date: 0.122"
```

Place your answers to the three items below here:

- Probability of 2 students in 23 is around 50%
- 48 students needed to get near 95%
- Probability of 3 students in 50 is around 12.5%

Task 2: Random permutations

Code set-up

A random permutation is a random shuffling of a set of integers $1, \dots, n$. The following code chunk presents code for a random permutation algorithm from Dobrow Example 2.14. The code chunk outputs the original list and then the random permutation of that list of n integers (in the code $n = 12$). Try running it yourself and note the shuffling or permutation of the n integers.

The algorithm sequentially moves down the list. At each position, say i , it swaps the number in slot i with a randomly chosen element from positions $i:n$ (so later in the list or no swap at all).

```
# Simulating random permutations
# Code from Example 2.14 of Dobrow (2014)
n = 12 # permutation of size n
perm = 1:n # store list of n integers
# sequentially move down the list
for(card in 1:(n-1)){
  x = sample(card:n,1) # randomly chose a position for swapping
  old = perm[card] # store the original number in position i
  perm[card] = perm[x] # replace the number in position i with the number in the randomly chosen
  perm[x] = old # complete the swap by placing the original number in position i in position x
}
1:n # original list
```

```
## [1] 1 2 3 4 5 6 7 8 9 10 11 12
```

```
perm # randomly permuted list
```

```
## [1] 4 8 10 9 6 1 5 11 7 12 3 2
```

In this task, we are going to shuffle a deck of cards and determine the probability that the top (first element in permutation) and bottom (last element in permutation) card are the same. The code

```
(floor((perm[1]-1)/13) == floor((perm[52]-1)/13))
```

checks if the first and last cards from the random permutation are the same suit. After performing the random permutation, you can store this value in a success vector (Boolean expression), and then wrap the

code in a for-loop to repeat the simulation. The proportion of experiments reporting a 1 in the success vector is an estimate of the probability that the top and bottom cards of a shuffle are the same.

The problems

Based on Dobrow problem 2.46. Revise the code chunk above to shuffle a standard deck of cards (52 cards). Simulate the probability that in a randomly shuffled deck, the top and bottom cards are the same suit.

```
n = 52
perm = 1:n
shuffle = function(reps) {
  successes=numeric(reps)
  for (i in 1:reps) {
    for (card in 1:(n-1)) {
      x = sample(card:n, 1)
      old = perm[card]
      perm[card] = perm[x]
      perm[x] = old
    }
    successes[i]=if(floor((perm[1]-1)/13)==floor((perm[52]-1)/13)) 1 else 0
  }
  return(successes)
}
sim = shuffle(1000)
mean(sim)
```

```
## [1] 0.24
```

Questions:

- Present the empirical probability (based on your simulation experiment) that in a randomly shuffled deck the top and bottom cards are the same suit.

The probability is around 23.5%

- Briefly explain how the code `(floor((perm[1]-1)/13) == floor((perm[52]-1)/13))` checks if the first and last cards from a random shuffling (permutation) are the same suit.

It checks if the cards are the same by dropping them to 0, 1, 2, or 3, then checking for equality. If they are equal, the expression returns true

- Based on our discussion of counting and probability, present an intuitive explanation of why the probability follows the value given by your simulation. (I.e., I am not asking you to mathematically find the probability. Just briefly explain why the value you obtained is what you may expect it to be based on the probability material covered to this point.)

It makes sense because there are 4 suits, there's equal chance for the first one to be of any suit. After the whole deck is gone, down to the last card, because the first one is guaranteed to be the suit to look for, we would expect there to be a less than exactly 1/4 chance of it being that same suit.