Data Science Probability

The textbook for the Data Science course series is freely available online.

This course corresponds to the Probability section of textbook, starting here.

This course assumes you are comfortable with basic math, algebra, and logical operations. HarvardX has partnered with DataCamp for all assignments in R that allow you to program directly in a browser-based interface. You will not need to download any special software.

Using a combination of a guided introduction through short video lectures and more independent in-depth exploration, you will get to practice your new R skills on real-life applications.

Probability theory is the mathematical foundation of statistical inference which is indispensable for analyzing data affected by chance, and thus essential for data scientists. In this course, you will learn important concepts in probability theory. The motivation for this course is the circumstances surrounding the financial crisis of 2007-2008. Part of what caused this financial crisis was that the risk of certain securities sold by financial institutions was underestimated. To begin to understand this very complicated event, we need to understand the basics of probability. We will introduce important concepts such as random variables, independence, Monte Carlo simulations, expected values, standard errors, and the Central Limit Theorem. These statistical concepts are fundamental to conducting statistical tests on data and understanding whether the data you are analyzing are likely occurring due to an experimental method or to chance. Statistical inference, covered in the next course in this series, builds upon probability theory.

Learning Objectives

- Important concepts in probability theory including random variables and independence
- How to perform a Monte Carlo simulation
- The meaning of expected values and standard errors and how to compute them in R
- The importance of the Central Limit Theorem

Course Overview

Section 1: Discrete Probability

You will learn about basic principles of probability related to categorical data using card games as examples.

Section 2: Continuous Probability

You will learn about basic principles of probability related to numeric and continuous data.

Section 3: Random Variables, Sampling Models, and The Central Limit Theorem

You will learn about random variables (numeric outcomes resulting from random processes), how to model data generation procedures as draws from an urn, and the Central Limit Theorem, which applies to large sample sizes.

Section 4: The Big Short

You will learn how interest rates are determined and how some bad assumptions led to the financial crisis of 2007-2008.

Section 1 Overview

Section 1 introduces you to Discrete Probability. Section 1 is divided into three parts:

- Introduction to Discrete Probability
- Combinations and Permutations
- Addition Rule and Monty Hall

After completing Section 1, you will be able to:

- Apply basic probability theory to categorical data.
- Perform a Monte Carlo simulation to approximate the results of repeating an experiment over and over, including simulating the outcomes in the Monty Hall problem.
- Distinguish between: sampling with and without replacement, events that are and are not independent, and combinations and permutations.
- Apply the multiplication and addition rules, as appropriate, to calculate the probably of multiple events occurring.
- Use **sapply** instead of a for loop to perform element-wise operations on a function.

Discrete Probability

The textbook for this section is available here

Key points

- The *probability of an event* is the proportion of times the event occurs when we repeat the experiment independently under the same conditions.

$Pr(A)$ = probability of event A

- An *event* is defined as an outcome that can occur when when something happens by chance.
- We can determine probabilities related to discrete variables (picking a red bead, choosing 48 Democrats and 52 Republicans from 100 likely voters) and continuous variables (height over 6 feet).

Monte Carlo Simulations

The textbook for this section is available here

Key points

- Monte Carlo simulations model the probability of different outcomes by repeating a random process a large enough number of times that the results are similar to what would be observed if the process were repeated forever.
- The **sample** function draws random outcomes from a set of options.
- The **replicate** function repeats lines of code a set number of times. It is used with **sample** and similar functions to run Monte Carlo simulations.

Code: The rep function and the sample function

```
beads <- rep(c("red", "blue"), times = c(2,3))    # create an urn with 2 red, 3 blue
beads    # view beads object
```

```
## [1] "red"  "red"  "blue" "blue" "blue"
```

```
sample(beads, 1)    # sample 1 bead at random
```

```
## [1] "blue"
```

Code: Monte Carlo simulation

Note that your exact outcome values will differ because the sampling is random.

```
B <- 10000    # number of times to draw 1 bead
events <- replicate(B, sample(beads, 1))    # draw 1 bead, B times
tab <- table(events)    # make a table of outcome counts
tab    # view count table
```

```
## events
## blue  red
## 6069 3931
```

```
prop.table(tab)    # view table of outcome proportions
```

```
## events
##   blue    red
## 0.6069 0.3931
```

Setting the Random Seed

The **set.seed** function

Before we continue, we will briefly explain the following important line of code:

```
set.seed(1986)
```

Throughout this book, we use random number generators. This implies that many of the results presented can actually change by chance, which then suggests that a frozen version of the book may show a different result than what you obtain when you try to code as shown in the book. This is actually fine since the results are random and change from time to time. However, if you want to to ensure that results are exactly the same every time you run them, you can set R's random number generation seed to a specific number. Above we set it to 1986. We want to avoid using the same seed every time. A popular way to pick the seed is the year - month - day. For example, we picked 1986 on December 20, 2018: $2018 - 12 - 20 = 1986$.

You can learn more about setting the seed by looking at the documentation:

~~~~
```?set.seed
~~~~

In the exercises, we may ask you to set the seed to assure that the results you obtain are exactly what we expect them to be.

Important note on seeds in R 3.5 and R 3.6

R was updated to version 3.6 in early 2019. In this update, the default method for setting the seed changed. This means that exercises, videos, textbook excerpts and other code you encounter online may yield a different result based on your version of R.

If you are running R 3.6, you can revert to the original seed setting behavior by adding the argument sample.kind="Rounding". For example:

```
set.seed(1)
set.seed(1, sample.kind="Rounding")    # will make R 3.6 generate a seed as in R 3.5
```

Using the `sample.kind="Rounding"` argument will generate a message:

`non-uniform 'Rounding' sampler used`

This is not a warning or a cause for alarm - it is a confirmation that R is using the alternate seed generation method, and you should expect to receive this message in your console.

If you use R 3.6, you should always use the second form of set.seed in this course series (outside of DataCamp assignments). Failure to do so may result in an otherwise correct answer being rejected by the grader. In most cases where a seed is required, you will be reminded of this fact.

Using the mean Function for Probability

An important application of the `mean` function

In R, applying the `mean` function to a logical vector returns the proportion of elements that are TRUE. It is very common to use the `mean` function in this way to calculate probabilities and we will do so throughout the course.

Suppose you have the vector beads:

```
beads <- rep(c("red", "blue"), times = c(2,3))
beads
```

```
## [1] "red"  "red"  "blue" "blue" "blue"
```

```
# To find the probability of drawing a blue bead at random, you can run:
mean(beads == "blue")
```

```
## [1] 0.6
```

This code is broken down into steps inside R. First, R evaluates the logical statement `beads == "blue"`, which generates the vector:

`FALSE FALSE TRUE TRUE TRUE`

When the `mean` function is applied, R coerces the logical values to numeric values, changing TRUE to 1 and FALSE to 0:

`0 0 1 1 1`

The mean of the zeros and ones thus gives the proportion of TRUE values. As we have learned and will continue to see, probabilities are directly related to the proportion of events that satisfy a requirement.

Probability Distributions

The textbook for this section is available here

Key points

- The probability distribution for a variable describes the probability of observing each possible outcome.
- For discrete categorical variables, the probability distribution is defined by the proportions for each group.

Independence

The textbook section on independence, conditional probability and the multiplication rule is available here

Key points

- *Conditional probabilities* compute the probability that an event occurs given information about dependent events. For example, the probability of drawing a second king given that the first draw is a king is:

$Pr(Card\,2\,is\,a\,king \mid Card\,1\,is\,a\,king) = 3/51$

- If two events A and B are independent, $Pr(A \mid B) = Pr(A)$.
- To determine the probability of multiple events occurring, we use the *multiplication rule*.

Equations

The multiplication rule for independent events is:

$Pr(A\,and\,B\,and\,C) = Pr(A)\ X\ Pr(B)\ X\ Pr(C)$

The multiplication rule for dependent events considers the conditional probability of both events occurring:

$Pr(A\,and\,B) = Pr(A)\ X\ Pr(B \mid A)$

We can expand the multiplication rule for dependent events to more than 2 events:

$Pr(A\,and\,B\,and\,C) = Pr(A)\ X\ Pr(B \mid A)\ X\ Pr(C \mid A\,and\,B)$

Assessment - Introduction to Discrete Probability

1. Probability of cyan

One ball will be drawn at random from a box containing: 3 cyan balls, 5 magenta balls, and 7 yellow balls.

What is the probability that the ball will be cyan?

```
cyan <- 3
magenta <- 5
yellow <- 7
balls <- cyan + magenta + yellow
p_cyan <- cyan/balls
p_cyan
```

```
## [1] 0.2
```

2. Probability of not cyan

One ball will be drawn at random from a box containing: 3 cyan balls, 5 magenta balls, and 7 yellow balls.

What is the probability that the ball will not be cyan?

```r
cyan <- 3
magenta <- 5
yellow <- 7
balls <- cyan + magenta + yellow
p_cyan <- cyan/balls
p_not_cyan = 1 - p_cyan
p_not_cyan
```

```
## [1] 0.8
```

or:

```r
p_not_cyan = (magenta + yellow)/balls
p_not_cyan
```

```
## [1] 0.8
```

3. Sampling without replacement

Instead of taking just one draw, consider taking two draws. You take the second draw without returning the first draw to the box. We call this sampling without replacement.

What is the probability that the first draw is cyan and that the second draw is not cyan?

```r
# p = p1_cyan * p2_not_cyan, without replacement
p_1 = cyan/balls
p_2 = 1 - (cyan-1)/(balls-1)
p_1 * p_2
```

```
## [1] 0.1714286
```

4. Sampling with replacement

Now repeat the experiment, but this time, after taking the first draw and recording the color, return it back to the box and shake the box. We call this sampling with replacement. What is the probability that the first draw is cyan and that the second draw is not cyan?

```r
# p = p1_cyan * p2_not_cyan, with replacement
p_1 = cyan/balls
p_2 = 1 - cyan/balls
p_1 * p_2
```

```
## [1] 0.16
```

1.2 Combinations and Permutations

Let's start by constructing a deck of cards using R. For this, we will use the function expand.grid() and the function paste()

Here is how we generate a deck of cards:

```
suits <- c("Diamonds", "Clubs", "Hearts", "Spades")
numbers <- c("Ace", "Deuce", "Three", "Four", "Five",
             "Six", "Seven", "Eight", "Nine", "Ten",
             "Jack", "Queen", "King")
deck <- expand.grid(number=numbers, suit=suits)
deck <- paste(deck$number, deck$suit)
```

With the deck constructed, we can now double check that the probability of a King in the first card is 1/13. We simply compute the proportion of possible outcomes that satisfy our condition:

```
kings <- paste("King", suits)
mean(deck %in% kings)
```

```
## [1] 0.07692308
```

which is 1/13.

Now, how about the conditional probability of the second card being a King given that the first was a King? Earlier, we deduced that if one King is already out of the deck and there are 51 left, then this probability is 3/51. Let's confirm by listing out all possible outcomes.

To do this, we can use the permutations() function from the gtools package. For any list of size n, this function computes all the different combinations we can get when we select r items. Here are all the ways we can choose two numbers from a list consisting of 1,2,3:

```
#here all the ways we can choose 2 numbers from the list 1, 2, 3, 4, 5.
#Notice that the order matters. So 3, 1 is different than 1, 3, So it appears in our permutations.
#Also notice that 1, 1; 2, 2; and 3, 3 don't appear, because once we pick a number, it can't appear aga
library(gtools)
library(permutations)
library(dplyr)
```

```
permutations(5, 2)
```

```
##       [,1] [,2]
##  [1,]    1    2
##  [2,]    1    3
##  [3,]    1    4
##  [4,]    1    5
##  [5,]    2    1
##  [6,]    2    3
##  [7,]    2    4
##  [8,]    2    5
##  [9,]    3    1
## [10,]    3    2
## [11,]    3    4
## [12,]    3    5
## [13,]    4    1
## [14,]    4    2
## [15,]    4    3
## [16,]    4    5
## [17,]    5    1
## [18,]    5    2
## [19,]    5    3
## [20,]    5    4
```

Optionally for this function permutations, we can add a vector. So for example, if you want to see 5 random 7-digit phone numbers out of all possible phone numbers, you could type code like this.

```
all_phone_numbers <- permutations(10, 7, v = 0:9)
n <- nrow(all_phone_numbers)
index <- sample(n, 5)
all_phone_numbers[index,]
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    5    3    1    2    8    6    9
## [2,]    7    3    5    8    1    9    6
## [3,]    3    4    0    5    7    8    9
## [4,]    0    9    8    4    5    7    1
## [5,]    2    6    1    5    3    9    8
```

```
#Here we're defining a vector of digits that goes from 0 to 9 rather than 1 through 10.
#So these four lines of code generate all phone numbers, picks 5 at random.
```

To compute all possible ways that we can choose 2 cards when the order matters, we simply type the following piece of code.

Here we use permutations.

```
hands <- permutations(52, 2, v = deck)
```

There's 52 cards, we're going to choose 2, and we're going to select them out of the vector that includes our card names, which we called deck earlier. This is going to be a matrix with 2 dimensions, 2 columns, and in this case, it's going to have 2,652 rows. Those are all the permutations.

Define the first card and the second card by grabbing the first and second columns using this simple piece of code.

```
first_card <- hands[,1]
second_card <- hands[,2]

#Now the cases for which the first hand was a King can be computed like this:
kings <- paste("King", suits)
sum(first_card %in% kings)
```

```
## [1] 204
```

```
#To get the conditional probability, we compute what fraction of these have a King in the second card:
sum(first_card %in% kings & second_card %in% kings) /
  sum(first_card %in% kings)
```

```
## [1] 0.05882353
```

```
#which is exactly 3/51, as we had already deduced. Notice that the code above is equivalent to:
mean(first_card %in% kings & second_card %in% kings) /
  mean(first_card %in% kings)
```

```
## [1] 0.05882353
```

the difference between the permutations functions, which lists all permutations, and the combination function, where order does not matter.

```
permutations(3,2)
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    1    3
## [3,]    2    1
## [4,]    2    3
## [5,]    3    1
## [6,]    3    2
```

```
combinations(3,2)
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    1    3
## [3,]    2    3
```

```
#In the second line, the outcome does not include (2,1) because (1,2) already was enumerated. The same
```

So to compute the probability of a Natural 21 in Blackjack, we can do this:

```
aces <- paste("Ace", suits)

facecard <- c("King", "Queen", "Jack", "Ten")
facecard <- expand.grid(number = facecard, suit = suits)
facecard <- paste(facecard$number, facecard$suit)

hands <- combinations(52, 2, v = deck)
mean(hands[,1] %in% aces & hands[,2] %in% facecard)
```

```
## [1] 0.04826546
```

In the last line, we assume the Ace comes first. This is only because we know the way combination enumerates possibilities and it will list this case first. But to be safe, we could have written this and produced the same answer:

```
mean((hands[,1] %in% aces & hands[,2] %in% facecard) |
       (hands[,2] %in% aces & hands[,1] %in% facecard))
```

```
## [1] 0.04826546
```

Monte Carlo example

Instead of using combinations to deduce the exact probability of a Natural 21, we can use a Monte Carlo to estimate this probability.

```
B <- 10000
results <- replicate(B, {
  hand <- sample(deck,2)
 ((hands[,1] %in% aces & hands[,2] %in% facecard) |
       (hands[,2] %in% aces & hands[,1] %in% facecard))
})
mean(results)
```

```
## [1] 0.04826546
```

In this case, we draw two cards over and over and keep track of how many 21s we get. We can use the function sample to draw two cards without replacements:

```
hand <- sample(deck, 2)
hand
```

```
## [1] "Seven Diamonds" "Deuce Diamonds"
```

And then check if one card is an Ace and the other a face card or a 10. Going forward, we include 10 when we say face card. Now we need to check both possibilities:

```
(hands[1] %in% aces & hands[2] %in% facecard) |
  (hands[2] %in% aces & hands[1] %in% facecard)
```

```
## [1] FALSE
```

If we repeat this 10,000 times, we get a very good approximation of the probability of a Natural 21.

Let's start by writing a function that draws a hand and returns TRUE if we get a 21. The function does not need any arguments because it uses objects defined in the global environment.

```
blackjack <- function(){
   hand <- sample(deck, 2)
  (hand[1] %in% aces & hand[2] %in% facecard) |
    (hand[2] %in% aces & hand[1] %in% facecard)
}
```

Here we do have to check both possibilities: Ace first or Ace second because we are not using the combinations function. The function returns TRUE if we get a 21 and FALSE otherwise:

```
blackjack()
```

```
## [1] FALSE
```

Now we can play this game, say, 10,000 times:

```
B <- 10000
results <- replicate(B, blackjack())
mean(results)
```

```
## [1] 0.0468
```

The Birthday Problem

If we assume this is a randomly selected group of 50 people, what is the chance that at least two people have the same birthday? Here we use a Monte Carlo simulation. For simplicity, we assume nobody was born on February 29. This actually doesn't change the answer much.

```
#birthdays can be represented as numbers between 1 and 365, so a sample of 50 birthdays can be obtained
n <- 50
bdays <- sample(1:365, n, replace = TRUE)
```

To check if in this particular set of 50 people we have at least two with the same birthday, we can use the function duplicated, which returns TRUE whenever an element of a vector is a duplicate. Here is an example:

```
duplicated(c(1,2,3,1,4,3,5))
```

```
## [1] FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE
```

```
#The second time 1 and 3 appear, we get a TRUE. So to check if two birthdays were the same, we simply us
any(duplicated(bdays))
```

```
## [1] TRUE
```

In this case, we see that it did happen. At least two people had the same birthday.

To estimate the probability of a shared birthday in the group, we repeat this experiment by sampling sets of 50 birthdays over and over:

```
same_birthday <- function(n){
  bdays <- sample(1:365, n, replace=TRUE)
  any(duplicated(bdays))
}

B <- 10000
results <- replicate(B, same_birthday(50))
mean(results)
```

```
## [1] 0.9722
```

Were you expecting the probability to be this high?

People tend to underestimate these probabilities. To get an intuition as to why it is so high, think about what happens when the group size is close to 365. At this stage, we run out of days and the probability is one.

Say we want to use this knowledge to bet with friends about two people having the same birthday in a group of people. When are the chances larger than 50%? Larger than 75%?

Let's create a look-up table. We can quickly create a function to compute this for any group size: