A Short Introduction to Applied Statistical Programming in R

Robert Schnitman

March 29, 2020 (Last updated: May 13, 2020)

Contents

Introduction

The purpose of this book is to teach students of social science statistics courses how to program in R for data analysis. Primarily focusing on Base R, this book will teach R "from the ground up," teaching the fundamentals without using external packages unless necessary or for quick demonstrations on the programming language's extensions. Overall, I hope that students will learn enough from this book to conduct data analysis in R independently.

Because I know people live busy lives, please feel free to skip chapters or simply only review the *Summary* subsections at the end of them–they are for your benefit!

Also, please feel free to email me at robertschnitman@gmail.com if you have any suggestions on improving this book!

Prerequisites

This book assumes that you have installed at least R version 3.6 at minimum (https://cran.r-project.org/). Installing the R Studio IDE afterward is strongly recommended (https://rstudio.com/products/rstudio/). Additionally, the focus of this book is more on programming in R rather than going in depth with the statistics–please consult your statistics textbook for the latter purpose instead.

Chapter 1

The Paradigms of R

There are three main programming paradigms–or styles–that R uses: Array Programming (AP), Functional Programming, and Object Oriented (OOP). Knowing how these paradigms work is important, as they will help one understand the syntax structure of R.

1.1 Array

The Array Programming (AP) paradigm allows us to access elements in a dataset via a matrix-like syntax.

```
# Select the 2nd row and 5th column
#   from mtcars, which is a pre-loaded dataset.
mtcars[2, 5]
```

```
## [1] 3.9
```

```
# Select the first 5 rows and all columns
mtcars[1:5, ]
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
```

1.2 Functional

Much like Excel, R has functions: execution statements with an input and an output–this syntax style is called Functional Programming (FP).

```r
# Mean of MPG from the mtcars dataset
mean(mtcars$mpg) # $ accesses MPG from mtcars.
```

```
## [1] 20.09062
```

```r
# Input  = mtcars$mpg
# OUtput = numeric value
```

Just like in math and Excel, we can compose multiple functions together.

```r
# Rounding the mean MPG by 2 digits.
round(mean(mtcars$mpg), 2)
```

```
## [1] 20.09
```

1.3 Object Oriented

In R, we can create and access objects, which are a storage of information with attributes: this paradigm is called Object Oriented Programming (OOP). This paradigm is concerned about classes and types–the "foreground" and "background" characteristics of a dataset, so to speak. Classes affect how data look to the user, whereas types are the specific attributes of some data.

Classes and types are discussed in the *Basics* chapter.

```r
# Access the MPG variable from mtcars
#   and save it to an object named "x"
x <- mtcars$mpg

# We can now refer to mtcars$mpg anytime with "x"
x
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
## [16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
## [31] 15.0 21.4
```

We can check the structure of our data objects to know their attributes.

```r
# Check the structure of mtcars.
# A data frame composed of numeric vectors.
str(mtcars)
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
```

```
## $ qsec: num 16.5 17 18.6 19.4 17 ...
## $ vs  : num 0 0 1 1 0 1 0 1 1 1 ...
## $ am  : num 1 1 1 0 0 0 0 0 0 0 ...
## $ gear: num 4 4 4 3 3 3 3 4 4 4 ...
## $ carb: num 4 4 1 1 2 1 4 2 2 4 ...
```

1.4 Summary

Table 1.1: Summary of Paradigms

Paradigm	Description	Example
Array	Bracket syntax structure to access data like a matrix	mtcars[2, 5]
Functional	Mathematical function syntax structure to compute over data.	mean(mtcars$mpg)
Object Oriented	Syntax structure in which data has stored attributes that affect how they look to the user.	str(mtcars)

Chapter 2

Basics

In this chapter, we will learn how to use R as a calculator; learn the different data types and classes in R; learn how to make assignments; and learn how to get help when you are stuck on a particular issue.

2.1 R as a calculator

You can use R like a calculator: the arithmetic operators are `+`, `-`, `*`, `/`, `^` (exponentiation), and `%%` (modular arithmetic)–there are more, but these operators are the basic ones (see more by typing `?'+'` into your console).

2.1.1 Operators

```
2+2  # Addition
```

```
## [1] 4
```

```
2-2  # Subtraction
```

```
## [1] 0
```

```
2*2  # Multiplication
```

```
## [1] 4
```

```
2/2  # Division
```

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

```
2^2  # Exponentiation
```

```
## [1] 4
```

```
2%%2 # Modular arithmetic
```

```
## [1] 0
```

2.2 Data Types and Classes

Classes are the "foreground" and types are the "background" characteristics of data. Classes affect how data look to the user, whereas types are the specific attributes of some data. We can check the class of an object with the `class()` function and type with the `typeof()` function.

Additionally, we can test to see if some data are a particular class or type with the `is.*()` and convert them with `as.*()`, where `*` can represent the classes and types that follow.

2.2.1 Classes

There are many classes–some pre-defined in R, while others have been created externally. The three main classes (besides the `numeric` and `character` vector class, which are also types) are the `matrix`, `list`, and `data frame`.

Table 2.1: Summary of Classes

Type	Decsription	Example
matrix	A 2-dimensional array of elements, where each column is of the same type.	matrix(1:9, 3, 3)
list	A collection of elements, where each one can be a different type or class.	list(1, "a", matrix(1:9, 3, 3)
data frame	A 2-dimensional set of elements, where each column can be a different type.	mtcars

2.2.2 Types

2.3 Assignments

Making assignments in R allows us to save information into an object, which further allows us to refer to a specific value without having to recalculate it each time.

Table 2.2: Summary of Types

Type	Decsription	Example
numeric	A vector of numbers.	2
character	A vector of strings (i.e. characters encased in quotes.	"String"
logical	Value of TRUE or FALSE.	TRUE
factor	A categorical vector with specified levels.	factor(mtcars$am)

```
x <- 2

x
```

```
## [1] 2
```

For a more complex example, we will run a regression model, save it to an object, and pass it to the `summary()` function to get more information from our model besides just its coefficients–we'll learn more about regressions in the *Linear Modeling* chapter.

```
my_model <- lm(mpg ~ wt + hp + am, mtcars)

summary(my_model)
```

```
##
## Call:
## lm(formula = mpg ~ wt + hp + am, data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.4221 -1.7924 -0.3788  1.2249  5.5317
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 34.002875   2.642659  12.867 2.82e-13 ***
## wt          -2.878575   0.904971  -3.181 0.003574 **
## hp          -0.037479   0.009605  -3.902 0.000546 ***
## am           2.083710   1.376420   1.514 0.141268
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.538 on 28 degrees of freedom
## Multiple R-squared:  0.8399, Adjusted R-squared:  0.8227
```

```
## F-statistic: 48.96 on 3 and 28 DF,  p-value: 2.908e-11
```

2.3.1 Adding/Removing Variables

There are two main ways we can add variables to our dataset: (1) the `$` ("accessor"/dollar-sign) method and (2) the `transform()` method.

```
mydata <- mtcars # copy data

# Let's create a variable called "my_new_var"
mydata$my_new_var <- with(mtcars, mpg/wt)

# Alternatively, the right-hand side
#   could be written as mtcars$mpg/mtcars$wt.

# Show a few rows from our dataset
head(mydata)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb my_new_var
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4   8.015267
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4   7.304348
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1   9.827586
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1   6.656299
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2   5.436047
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1   5.231214
```

```
# Let's do it again but with transform().
mydata2 <- transform(mydata, my_new_var = mpg/wt)

head(mydata2)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb my_new_var
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4   8.015267
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4   7.304348
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1   9.827586
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1   6.656299
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2   5.436047
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1   5.231214
```

To remove variables, we assign them to be `NULL`.

```
mydata$my_new_var <- NULL

head(mydata)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
```

```
## Datsun 710         22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive     21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant            18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

2.4 Viewing Data

We can view data like an Excel spreadsheet in a separate window in R (or separate tab in RStudio) via the `View()` function. Try it out in your console!

```r
View(mtcars)
```

2.5 Getting Help

There are two main ways of getting help in R: (1) using the `?` operator to access a function's documentation and (2) Googling your questions OR searching for them on StackOverflow–when you're beginning R, chances are that the problems you encounter have been solved.

```r
# Accessing the documentation for the mean function.
?mean
```

2.6 Summary

Table 2.3: Summary of Basics

Functionality	Description	Example
+,-,*,/,^/%%	Arithmetic operators	2+2
Types/Classes	Attributes of an object.	str(mtcars); class(mtcars); typeof(mtcars$mpg)
Assignments	Storing a value into an object.	x <- 2
Viewing data	How to view your dataset like an Excel spreadsheet.	View(mtcars)
Getting help	How to look for help.	?mean; Google/StackOverflow

Chapter 3

Data Management

In this chapter, we will learn how to replace values, switch values, import data, combine data, subset data, and split data.

3.1 Replacing Values

We can replace values with the `replace()` function.

```
x <- 1:10 # 1 through 10.

# If x equals 2, 5, or 7, replace with 0.
## replace(vector, condition, replacement value).
replace(x, x %in% c(2, 5, 7), 0)
```

```
##  [1]  1  0  3  4  0  6  0  8  9 10
```

3.2 Switching Values

We can switch–or recode–values with the `switch()` function. By default, `switch()` is a "scalar" function in that it only produces a single value. To produce a vector of values, we combine it with `sapply()`–see the *Functionals* chapter for more details on `sapply()`.

3.2.1 Scalar Case

In the case of a single value, all we need to pass into `switch()` are (1) the data object and (2) an expression stating what the old value should become (i.e, provide the old value and the replacement value).

```
# SYNTAX OF switch():
## switch(x, old_value = new_value)
```

```
x <- "a"

xs <- switch(x, a = 1)

xs
```

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

3.2.2 Vector Case

For the case of applying `switch()` to vectors, we make use of `sapply()`.

To take a case in point, let's first generate some random data of racial groups.

```
set.seed(1) # Remember our random sampling

# Generate vector of unique values.
my_vector <- c('Asian', 'African American', 'White', 'Other')

# Conduct repeat sampling of my_vector
## See the Probability Functions chapter for more details on sample().
my_vector2 <- sample(my_vector, 20, replace = TRUE)

# Print the new vector.
my_vector2
```

```
##  [1] "Asian"            "Other"            "White"            "Asian"
##  [5] "African American" "Asian"            "White"            "White"
##  [9] "African American" "African American" "White"            "White"
## [13] "Asian"            "Asian"            "Asian"            "African American"
## [17] "African American" "African American" "African American" "White"
```

Let's say that we want to recode these values: 0 for *White*, 1 for *African American*, 2 for *Asian*, and 3 for *Other*. To do so, we first define a function and pass it through `sapply()`–see the *Function Writing* and *Functionals* chapters respectively for more information.

```
# First, define a function that recodes the races into integers.
my_switch <- function(v) {

  switch(v, White = 0, `African American` = 1, Asian = 2, Other = 3)
  # We use back quotes for "African American" because of the space.

}

# Now we can pass my_switch to sapply() to execute the recoding.
sapply(my_vector2, my_switch)
```

```
##            Asian            Other            White            Asian 
##                2                3                0                2 
## African American            Asian            White            White 
##                1                2                0                0 
## African American African American            White            White 
##                1                1                0                0 
##            Asian            Asian            Asian African American 
##                2                2                2                1 
## African American African American African American            White 
##                1                1                1                0
```

3.3 Importing Data

We can import datasets with `read.table()`–this method is the most general.

```
# Set path to dataset
# For this example, our data is in the data folder
#  and our data are separated by commas.
my_data <- read.table('data/mtcars.csv', sep = ',', stringsAsFactors = FALSE)

# Setting stringsAsFactors = FALSE maintains strings as strings.
## See the Basics chapter for more detail on classes and types.
```

In the case of files with comma-separated values, we can use `read.csv()` to import them more easily.

```
# Set path to dataset
my_data <- read.csv('data/mtcars.csv', stringsAsFactors = FALSE)
```

3.4 Combining Data

There are three main ways to combine data: (1) `cbind()`, (2) `rbind()`, and (3) `merge()`.

3.4.1 cbind()/rbind()

The function `cbind()` combines vectors or datasets column-wise, while `rbind()` does so row-wise.

```
x <- 1:5
y <- 6:10

cbind(x, y)
```

```
##      x  y
## [1,] 1  6
## [2,] 2  7
```

```
## [3,] 3  8
## [4,] 4  9
## [5,] 5 10
```

```
rbind(x, y)
```

```
##   [,1] [,2] [,3] [,4] [,5]
## x    1    2    3    4    5
## y    6    7    8    9   10
```

If we have a list of values we want to combine, we can use `do.call()` and `cbind()`/`rbind()` together. The former iteratively calls a function on a list, which can be useful for combining multiple datasets together. `do.call()` is a special case of a function called a *functional*, which is a function that takes other functions as inputs–this concept is discussed more in the *Functionals* chapter.

```
my_list <- list(x = 1:5, y = 6:10, z = 11:15)

do.call(cbind, my_list)
```

```
##      x  y  z
## [1,] 1  6 11
## [2,] 2  7 12
## [3,] 3  8 13
## [4,] 4  9 14
## [5,] 5 10 15
```

```
do.call(rbind, my_list)
```

```
##   [,1] [,2] [,3] [,4] [,5]
## x    1    2    3    4    5
## y    6    7    8    9   10
## z   11   12   13   14   15
```

3.4.2 `merge()`

Merging data with `merge()` (AKA "joining data") is powerful, as we can combine disparate datasets that have a common linking variable between them.

```
set.seed(1) # remember our random numbers from rnorm().

data1 <- data.frame(survey_id = 1:5,
                    wage      = rnorm(5, mean = 15, sd = 5))

data2 <- data.frame(survey_id  = 5:1,
                    experience = rnorm(5, mean = 5, sd = 3))

# merge(first data, second data, by = 'a common variable').
data_merge <- merge(data1, data2, by = 'survey_id')
```

```
data_merge # An "inner-join" of datasets
```

```
##   survey_id     wage experience
## 1         1 11.86773   4.083835
## 2         2 15.91822   6.727344
## 3         3 10.82186   7.214974
## 4         4 22.97640   6.462287
## 5         5 16.64754   2.538595
```

What we accomplished here is an *inner join*: a join in which two datasets overlap. See the documentation file for `merge()` for more information on different types of joins (i.e., type `?merge` into the R console).

3.5 Subsetting Data

To subset data, we can pass data and relational/logic operators[1] into the `subset()` function, or we can use the bracket syntax and use the operators there.

The relational operators are the following:

- `<`, `>`, `<=` (less than or equal to), `>=` (greater than or equal to)
- `==` (equal to), `!=` (not equal to)

The main logic operators are the following:

- `&` (and)
- `|` (or)

3.5.1 Vector Case

Suppose we have the following vector:

```
x <- -10:10 # integers from -10 to 10.
x
```

```
##  [1] -10  -9  -8  -7  -6  -5  -4  -3  -2  -1   0   1   2   3   4   5   6   7   8
## [20]   9  10
```

Then we can subset like the following:

```
x[x < 0] # same as subset(x, x < 0)
```

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

[1]These operators are "binary operators," which compare values (R Documentation, Comparison). See `?Comparison` for more information. For logic operators, see `?Logic`.

```
x[x > 2 & x < 5]
```

```
## [1] 3 4
```

```
# We can use functions inside the brackets.
## For example, %in% is a matching function:
##   let's use it to subset for only 1 through 5.
x[x %in% 1:5]
```

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

3.5.2 Data Frame Case

Suppose the dataset `mtcars`. Then we can subset like the following:

```
subset(mtcars, mpg > 30) # Same as mtcars[mtcars$mpg > 30, ]
```

```
##                 mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Fiat 128       32.4   4 78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic    30.4   4 75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla 33.9   4 71.1  65 4.22 1.835 19.90  1  1    4    1
## Lotus Europa   30.4   4 95.1 113 3.77 1.513 16.90  1  1    5    2
```

```
subset(mtcars, mpg > 30 & wt > 1.7)
```

```
##                 mpg cyl disp hp drat    wt  qsec vs am gear carb
## Fiat 128       32.4   4 78.7 66 4.08 2.200 19.47  1  1    4    1
## Toyota Corolla 33.9   4 71.1 65 4.22 1.835 19.90  1  1    4    1
```

3.6 Splitting Data

To split data, we pass a data frame and a variable into the `split()` function.

```
split(mtcars, mtcars$gear) # Splits into 3 subsets.
```

```
## $`3`
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
```

```
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
##
## $`4`
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4      21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280       19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C      17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
##
## $`5`
##                 mpg cyl  disp  hp drat    wt qsec vs am gear carb
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
## Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.5  0  1    5    4
## Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.5  0  1    5    6
## Maserati Bora  15.0   8 301.0 335 3.54 3.570 14.6  0  1    5    8
```

Splitting can be useful when you want to apply a function that's contingent on subsets of data. For example, we can split the data and perform a regression model on each of them.

```r
# 1. Split dataset by a splitting variable.
my_split <- split(mtcars, mtcars$gear)

# 2. Estimate a regression model based on each subset.
my_models <- lapply(my_split, function(data) lm(mpg ~ wt, data))

# 3. Print the coefficients in a matrix form.
sapply(my_models, coef)
```

```
##                     3         4         5
## (Intercept) 28.395036 42.492769 42.562784
## wt          -3.156854 -6.863478 -8.046336
```

For more information about `lapply()` and `sapply()`, see the *Functionals* chapter; for more information about `lm()`, see the *Linear Modeling* chapter.

3.7 Summary

Table 3.1: Summary of Data Management Functions

Function	Description	Example
replace(x, condition, replacement)	Replace a value in a vector based on a condition.	x <- 1:10; replace(x, x %in% c(2, 5, 7), 0)
switch(x, expression)	Switch (recode) values.	x <- ’a’; switch(x, a = 1)
read.table(’path/to/file.csv’, sep = ’,’)	Import a dataset.	my_data <- read.table(’data/mtcars.csv’, sep = ’,’)
cbind(x,y)/rbind(x,y)	Combine data column- or row-wise.	x <- 1:5; y <- 6:10; cbind(x, y); rbind(x,y)
do.call(function, list)	Iteratively call a function on a list	my_list <- list(x = 1:5, y = 6:10, z = 11:15); do.call(cbind, my_list)
merge(x, y, by = ’linking_var’)	Join data by a linking variable.	data1 <- data.frame(survey_id = 1:5, wage = rnorm(5, mean = 15, sd = 5)) data2 <- data.frame(survey_id = 5:1, experience = rnorm(5, mean = 5, sd = 3)) data_merge <- merge(data1, data2, by = ’survey_id’)
subset(data, condition); x[condition]	Subset data via relational and logic operators.	subset(mtcars, mpg > 30 & wt > 1.7)
split(data, grouping_variable)	Split data by a grouping variable	split(mtcars, mtcars$gear)

Chapter 4

String Functions

String functions allow us to combine, pattern-match, and substitute character vectors. These functions are useful for detecting and recoding specific values.

4.1 Concatenate Strings

There are two concatenation functions we can use: `paste()` and `paste0()`. The former assumes you want to separate the concatenated elements with a space, whereas the latter will assume no separation.

```
paste('a', 'b')
```

```
## [1] "a b"
```

```
paste('a', 'b', sep = '-')
```

```
## [1] "a-b"
```

```
paste0('a', 'b')
```

```
## [1] "ab"
```

4.2 Subset Strings

In Excel, we can subset strings with `LEFT()`, `MID()`, and `RIGHT()`. In R, we can subset strings with `substr()`/`substring()`, which both act similarly as `MID()` from Excel.

```
x <- 'Albatross'

substr(x, 1, 4)
```

```
## [1] "Alba"
```

```
substring(x, 5) # Goes to the end by default
```

```
## [1] "tross"
```

4.3 Split Strings

We can split strings with the `strsplit()` function. The output is a list, where each list element is a character vector.

```
x <- c('This is a sentence.',
       'This is another sentence.',
       'This is yet another sentence.')

x
```

```
## [1] "This is a sentence."           "This is another sentence."
## [3] "This is yet another sentence."
```

```
# Split vector elements by space
my_split <- strsplit(x, split = ' ')

# Output is a list
my_split
```

```
## [[1]]
## [1] "This"      "is"        "a"         "sentence."
##
## [[2]]
## [1] "This"      "is"        "another"   "sentence."
##
## [[3]]
## [1] "This"      "is"        "yet"       "another"   "sentence."
```

We can use `do.call()` and `c()` to combine these list elements into a single vector for a total of 13 elements. The function `do.call()` iteratively executes a function and `c()` ("combine") combines elements into a vector.

```
do.call(c, my_split)
```

```
##  [1] "This"      "is"        "a"         "sentence." "This"      "is"
##  [7] "another"   "sentence." "This"      "is"        "yet"       "another"
## [13] "sentence."
```

4.4 Substitute Strings

We can make character substitutions with `gsub()`.

```r
x <- c('This is a sentence.',
       'This is another sentence.',
       'This is yet another sentence.')

gsub('sentence', 'drink', x)
```

```
## [1] "This is a drink."           "This is another drink."
## [3] "This is yet another drink."
```

4.5 Match String Patterns

We can pattern-match strings with `grep()` and `grepl()`. The former outputs the position (or value) of a pattern match, while the latter outputs a Boolean value (i.e. `TRUE`/`FALSE`).

```r
# Cars that start witih "M"
grep('^M', rownames(mtcars), value = TRUE)
```

```
##  [1] "Mazda RX4"     "Mazda RX4 Wag" "Merc 240D"     "Merc 230"
##  [5] "Merc 280"      "Merc 280C"     "Merc 450SE"    "Merc 450SL"
##  [9] "Merc 450SLC"   "Maserati Bora"
```

```r
# Which cars start with and do not start with "M"?
grepl('^M', rownames(mtcars))
```

```
##  [1]  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
## [13]  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [25] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
```

```r
# Selecting columns that start with "m".
# We set drop = FALSE to maintain a data frame.
head(mtcars[, grep('^m', names(mtcars)), drop = FALSE])
```

```
##                    mpg
## Mazda RX4         21.0
## Mazda RX4 Wag     21.0
## Datsun 710        22.8
## Hornet 4 Drive    21.4
## Hornet Sportabout 18.7
## Valiant           18.1
```

Check out more regular expressions with RStudio's cheat sheet on strings.

4.6 Summary

Table 4.1: Summary of String Functions

Function	Description	Example
paste(x, y)/paste0(x, y)	Concatenation of x and y.	paste(’a’, ’b’); paste0(’a’, ’b’)
substr(x, start, end)	Subset strings.	substr(’Albatross’, 1, 4)
strsplit(x, split = ’ ’)	Split a string by a splitting character.	x <- c(’This is a sentence.’, ’This is another sentence.’, ’This is yet another sentence.’) strsplit(x, split = ’ ’)
gsub(pattern, replacement, x)	Substitute a portion of a string vector based on a given pattern.	gsub(’sentence’, ’drink’, ’This is a sentence.’)
grep/grepl(pattern, vector)	Pattern match a string and output its position OR Boolean (i.e. TRUE/FALSE).	grep(’^M’, rownames(mtcars), value = TRUE)

Chapter 5

Control Flow

Control flow statements allow us to control the flow of our script or data. This functionality is useful for when we want different results depending on specific conditions.

5.1 if and ifelse()

The `if` statement controls the flow of *your R script*, branching out to different possibilities if a condition is not met.

```
x <- 2

if (x == 2) {

  'x is 2!'

} else if (x == 3) {

  'x is 3!'

} else {

  'x is not 2 nor 3!'

}
```

```
## [1] "x is 2!"
```

The `ifelse()` function, on the other hand, controls the flow of *your vector*.

```
x <- c(1:10)
```

```
# If x is divisible by 2, then "even"; else, "odd."
ifelse(x %% 2 == 0, paste0(x, ': even'), paste0(x, ': odd'))
```

```
##  [1] "1: odd"   "2: even"  "3: odd"   "4: even"  "5: odd"   "6: even"
##  [7] "7: odd"   "8: even"  "9: odd"   "10: even"
```

5.2 Loops

Loops allow the user to operate on data iteratively, which is useful for reducing repetitive code.

5.2.1 `for` loop

In a `for` loop, we iterate over data *for each* data element in a sequence.

```
# Structure of a for loop
x <- c() # empty vector or list.

# For each data element in some_data...
for (i in seq_along(some_data)) {

  do_something(some_data[, i])
  # The "i" represents the column position in this case.

}
```

Let's take this example: getting the means for each column in the dataset `mtcars`, which is pre-loaded into R.

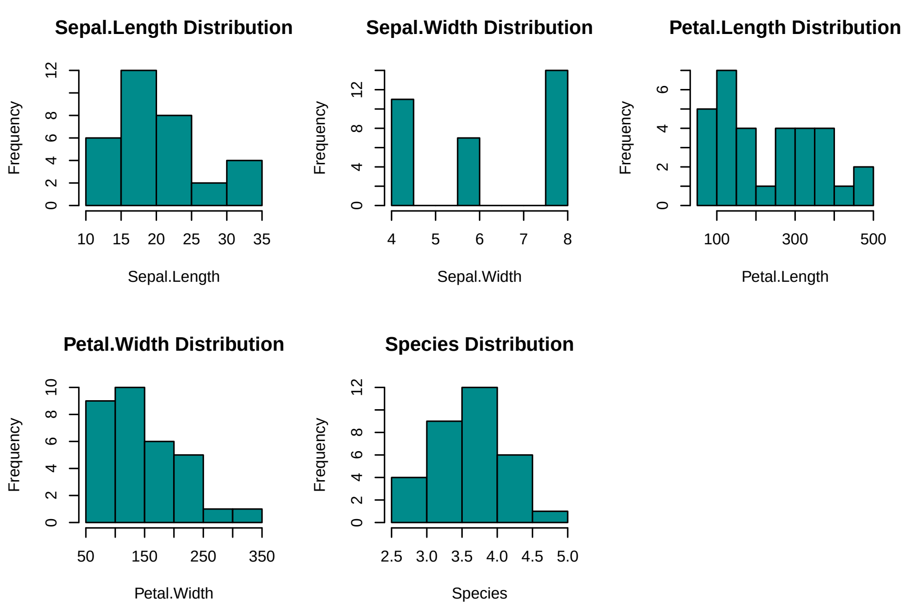
```
# Getting the means for each column in mtcars.

## Create an empty vector into which we will
##   store means.
x <- c()

## For each variable in mtcars...
for (i in seq_along(mtcars)) {

  ### Store the mean of that variable
  ###   into x.
  x[i] <- mean(mtcars[, i])

}

x
```

```
##  [1]  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250
```

```
## [7] 17.848750   0.437500   0.406250   3.687500   2.812500
```

There is actually a much better way to get the means of all columns in a dataset, which will be discussed in the *Functionals* chapter. In the meantime, the following is a more complex use-case of a `for` loop.

```
# set 4x3 canvas
par(mfrow = c(2, 3))

# For each column in the dataset iris...
for (i in seq_along(iris)) {

  # Plot a histogram.
  hist(mtcars[, i], # Get column vector.
       xlab = names(iris)[i], # Get name of column.
       ylab = 'Frequency',
       col  = 'cyan4',
       # Set the title to be based on the column name.
       main = paste(names(iris[i]), 'Distribution'))

}
```



For more on graphs, see the *Graphing* chapter.

5.2.2 `while` loop

In contrast to the `for` loop, the `while` loop iterates over data until the specified condition breaks (i.e., no longer true).

```
# Set an initial value for the while loop.
x <- 0

# While x is less than 10...
while (x < 10) {

  # Add 1 to it...
  x <- x + 1

  # And then print it to the console.
  print(x)

}
```

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

5.3 Summary

Table 5.1: Control Flow Statements

Statement.or.Function	Description	Example
if (condition) {output}	Control the flow of the R script.	if (x == 2) {'x is 2!'} else {'x is not 2!'}
ifelse(test, yes, no)	Control the flow of a vector.	ifelse(1:10 %% 2 == 0, 'even', 'odd')
for (statement) {output}	Iterate over each data element.	x <- c(); for (i in seq_along(mtcars)) { x[i] <- mean(mtcars[, i]) }; x
while (condition) {output}	Iterate over data until a condition breaks.	x <- 0; while (x < 10) { x <- x + 1 print(x) }

Chapter 6

Descriptive Statistics

There are various functions for descriptive statistics in R. The below subsections show a selected sample.

6.1 Centrality and Spread

Like in Microsoft Excel, we can cast centrality and spread functions on a variable.

```
k <- c(1, 5, 7, 9)
mean(k)
```

```
## [1] 5.5
```

```
# Use the $ operator for columns in a dataset
mean(mtcars$mpg)
```

```
## [1] 20.09062
```

If you want to use multiple functions on a single variable, the `with()` function can be useful, as it lets you define the local environment to be the desired dataset so that you do not have to use the `$` operator repeatedly.

```
with(mtcars, c(mean = mean(mpg), median = median(mpg), sd = sd(mpg)))
```

```
##      mean    median        sd
## 20.090625 19.200000  6.026948
```

6.2 Minimum and Maximum

To compute the minimum and maximum of a variable, we can use the `min()` and `max()` functions respectively.

```
x <- 1:10 # 1 through 10.

min(x)
```

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

```
max(x)
```

```
## [1] 10
```

6.3 Data Dimensions

To know the dimensions of an object in R, we can use `nrow()/NROW` for the number of rows; `ncol()/NCOL()` for the number of columns; and `dim()` for number of both rows and columns simultaneously.

```
NROW(mtcars)
```

```
## [1] 32
```

```
NCOL(mtcars)
```

```
## [1] 11
```

```
dim(mtcars)
```

```
## [1] 32 11
```

6.4 Data Summary

We can cast `summary()` on an object to capture summary information on an object. This function is useful following `str()`, as you can get a sense of what your dataset is like.

```
# Preview the dataset
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 .
```

```
# Summarize the dataset.
summary(iris)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
```

```
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

Note that because `Species` is a factor variable, we obtain counts by category for that column instead of quantiles and means like the others.

6.5 Frequency Tables

To get counts by groups, we can use the `table()` function, while using `prop.table()` on a `table()` computation produces proportions. The input of `table()` can be one to two columns and the output is a `table` class.

6.5.1 Single-variable Case

```
my_table <- table(iris$Species)

my_table
```

```
##
##     setosa versicolor  virginica
##         50         50         50
```

```
prop.table(my_table)
```

```
##
##     setosa versicolor  virginica
##  0.3333333  0.3333333  0.3333333
```

6.5.2 Multi-variable Case

```
my_table2 <- with(mtcars, table(am, gear))

my_table2
```

```
##    gear
## am   3  4  5
```

```
##   0 15  4  0
##   1  0  8  5
```

```
my_table3 <- with(mtcars, table(am, gear, cyl))

my_table3
```

```
## , , cyl = 4
## 
##    gear
## am   3  4  5
##   0  1  2  0
##   1  0  6  2
## 
## , , cyl = 6
## 
##    gear
## am   3  4  5
##   0  2  2  0
##   1  0  2  1
## 
## , , cyl = 8
## 
##    gear
## am   3  4  5
##   0 12  0  0
##   1  0  0  2
```

6.5.3 Converting to a Data Frame

If we apply the `as.data.frame()` function to an object of a `table` class, the output would be structured in a way such that we have a column (or columns) containing the group(s) and a column for the frequency. The structure is useful, as it is in a format that is acceptable for CSV output, for example.

```
freq <- table(iris$Species)
prop <- prop.table(freq)

as.data.frame(freq)
```

```
##         Var1 Freq
## 1     setosa   50
## 2 versicolor   50
## 3  virginica   50
```

```
as.data.frame(prop)
```

```
##         Var1      Freq
```

```
## 1     setosa 0.3333333
## 2 versicolor 0.3333333
## 3  virginica 0.3333333
```

```
my_table_df <- merge(as.data.frame(freq), as.data.frame(prop), by = 'Var1')

names(my_table_df) <- c('Species', 'Frequency', 'Percent')

my_table_df
```

```
##      Species Frequency   Percent
## 1     setosa        50 0.3333333
## 2 versicolor        50 0.3333333
## 3  virginica        50 0.3333333
```

```
write.csv(my_table_df, 'my_example_table.csv')
```

6.6 Summary

Table 6.1: Summary of Descriptive Statistics Functions

Function	Description	Example
mean(x)	Computes the mean.	mean(mtcars\$mpg)
sd(x)	Computes the standard deviation.	sd(mtcars\$mpg)
median(x)	Computes the median.	median(mtcars\$mpg)
min(x)	Computes the minimum.	min(mtcars\$mpg)
max(x)	Computes the maximum.	max(mtcars\$mpg)
nrow(x)/NROW(x)	Computes the number of rows.	nrow(mtcars); NROW(mtcars)
ncol(x)/NCOL(x)	Computes the number of columns.	ncol(mtcars); NCOL(mtcars)
dim(x)	Computes the number of rows and columns.	dim(mtcars)
length(x)	Computes the number of elements in a data object.	length(mtcars\$mpg)
summary(x)	Summarizes a dataset.	summary(mtcars)
table(x)	Generates a frequency table for one or more variables.	table(mtcars\$gear); with(mtcars, table(gear, am))
prop.table(table)	Generates a proportions table.	prop.table(table(mtcars\$gear))

Chapter 7

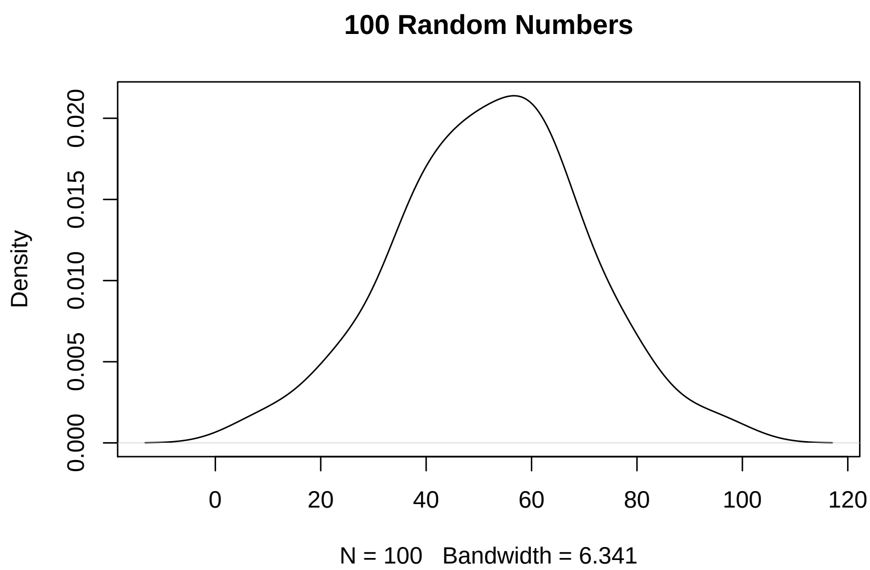
Probability Functions

This chapter will primarily focus on the normal distribution functions in R.

7.1 Generating Random Numbers

To calculate random numbers in R based on a normal distribution, we can use the `rnorm()` function. By default, the mean and sd respectively are 0 and 1; but we can change these parameters as necessary.

```
set.seed(1) # Remember our random numbers
x <- rnorm(100,       # 100 random numbers
           mean = 50, # with a mean of 50
           sd = 20)   # and SD of 20.

plot(density(x),
     main = '100 Random Numbers')
```



See more on plots in the *Graphing* chapter.

7.2 Sampling

We can take a random sampling of a vector with `sample()`.

```
set.seed(1) # Remember our random values.

# 10 random numbers from
#   a vector of 100 values.
sample(1:100, size = 10, replace = TRUE)
```

```
##  [1] 68 39  1 34 87 43 14 82 59 51
```

For a dataset, we can do the following:

```
set.seed(1) # Remember our random values.

# Random 5 rows
mtcars[sample(1:NROW(mtcars), 5), ]
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Pontiac Firebird 19.2   8  400 175 3.08 3.845 17.05  0  0    3    2
## Hornet 4 Drive   21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Duster 360       14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
## Mazda RX4        21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
```

```
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
```

7.3 Others

See `?rnorm`, `?rchisq`, and `?rpois` for more information on normal, chi-square, and Poisson probability distributions

7.4 Summary

Table 7.1: Summary of Probability Distributions

Function	Description	Example
rnorm(x)	x random numbers based on a normal distribution.	rnorm(10)
sample(x, size)	Sample a vector with a specified size.	sample(1:100, size = 10)
Other probability functions.	See '?rnorm', '?rchisq', and '?rpois'	

Chapter 8

Function Writing

Writing functions allows us to condense a process into a single function.

8.1 Univariate Case

If we wanted to index a variable by its mean, we could simply type `x/mean(x)`, where `x` is our vector. However, what if there were a function called `index()` that makes this process more clear? There is not one inherently in R, but we are able to create it:

```
index <- function(x) { # the formals/arguments

  x/mean(x) # The body

}

index(mtcars$mpg)
```

```
##  [1] 1.0452636 1.0452636 1.1348577 1.0651734 0.9307824 0.9009177 0.7117748
##  [8] 1.2144968 1.1348577 0.9556696 0.8859854 0.8163011 0.8610981 0.7565718
## [15] 0.5176544 0.5176544 0.7316846 1.6126925 1.5131436 1.6873542 1.0701509
## [22] 0.7715041 0.7565718 0.6620003 0.9556696 1.3588427 1.2941359 1.5131436
## [29] 0.7864365 0.9805569 0.7466169 1.0651734
```

We can cast this new function over all columns in `mtcars` with `sapply()`.[1]

```
# Get only a few rows.
head(sapply(mtcars, index))
```

```
##                  mpg       cyl      disp        hp      drat        wt      qsec
```

[1]See the *Functionals* chapter for more on `sapply()` and its bretheren.

```
## [1,] 1.0452636 0.9696970 0.6934756 0.7498935 1.0843688 0.8143601 0.9221934
## [2,] 1.0452636 0.9696970 0.6934756 0.7498935 1.0843688 0.8936203 0.9535682
## [3,] 1.1348577 0.6464646 0.4680961 0.6340009 1.0704666 0.7211128 1.0426500
## [4,] 1.0651734 0.9696970 1.1182295 0.7498935 0.8563733 0.9993006 1.0891519
## [5,] 0.9307824 1.2929293 1.5603202 1.1930124 0.8758363 1.0692361 0.9535682
## [6,] 0.9009177 0.9696970 0.9752001 0.7158074 0.7673994 1.0754526 1.1328524
##            vs       am      gear      carb
## [1,] 0.000000 2.461538 1.0847458 1.4222222
## [2,] 0.000000 2.461538 1.0847458 1.4222222
## [3,] 2.285714 2.461538 1.0847458 0.3555556
## [4,] 2.285714 0.000000 0.8135593 0.3555556
## [5,] 0.000000 0.000000 0.8135593 0.7111111
## [6,] 2.285714 0.000000 0.8135593 0.3555556
```

8.2 Multivariate Case

In the univariate case, we indexed a vector by its mean. What if we wanted to use the median instead? We would simply need to replace `mean` with `median`. Alternatively, we can add an additional input into our function that specifies what aggregation function to use in the indexing.

```
index2 <- function(x, f) {

  x/f(x)

}
```

Now we can use any function in the `f` input.

```
head(index2(mtcars$mpg, mean)) # show only a few elements
```

```
## [1] 1.0452636 1.0452636 1.1348577 1.0651734 0.9307824 0.9009177
```

```
head(index2(mtcars$mpg, median)) # show only a few elements
```

```
## [1] 1.0937500 1.0937500 1.1875000 1.1145833 0.9739583 0.9427083
```

```
head(index2(mtcars$mpg, max)) # show only a few elements
```

```
## [1] 0.6194690 0.6194690 0.6725664 0.6312684 0.5516224 0.5339233
```

Our `index2` function is actually special in that it is not only a multivariate function but a *functional*, which is a function that takes another function as an input–see the *Functionals* chapter for more details.

8.3 Summary

Table 8.1: Summary of Function Writing

Function	Description	Example
function(x)	Write a function, which consists of arguments and the body.	index <- function(x) x/mean(x)

Chapter 9

Functionals

Functionals are functions that take a function as an input and output a value. They are useful for casting a function over all columns in a dataset or elements in a list.

This chapter will demonstrate a select handful of functionals–see `?lapply` for more information.

9.1 `lapply()`

The `lapply()` function ("list apply") casts a function over a dataset and outputs a list.

```
lapply(mtcars, mean)
```

```
## $mpg
## [1] 20.09062
##
## $cyl
## [1] 6.1875
##
## $disp
## [1] 230.7219
##
## $hp
## [1] 146.6875
##
## $drat
## [1] 3.596563
##
## $wt
```

```
## [1] 3.21725
##
## $qsec
## [1] 17.84875
##
## $vs
## [1] 0.4375
##
## $am
## [1] 0.40625
##
## $gear
## [1] 3.6875
##
## $carb
## [1] 2.8125
```

9.2 sapply()

The `sapply()` function ("simplified apply") casts a function over a dataset and outputs a matrix (or list, depending on the function).

```r
sapply(mtcars, mean)
```

```
##        mpg        cyl       disp         hp       drat         wt       qsec
##  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750
##         vs         am       gear       carb
##   0.437500   0.406250   3.687500   2.812500
```

9.3 apply()

The `apply()` function can cast a function over a dataset row-wise or column-wise, returning a matrix.

```r
# Row-wise means.
# show only a few with head().
head(apply(mtcars, 1, mean))
```

```
##         Mazda RX4     Mazda RX4 Wag        Datsun 710    Hornet 4 Drive
##          29.90727          29.98136          23.59818          38.73955
## Hornet Sportabout           Valiant
##          53.66455          35.04909
```

```r
# Column-wise means.
apply(mtcars, 2, mean)
```

```
##        mpg        cyl       disp         hp       drat         wt       qsec
```

```
## 20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750
##         vs         am       gear       carb
##   0.437500   0.406250   3.687500   2.812500
```

9.4 vapply()

The `vapply()` function ("vectorized apply") works similarly as `sapply()`; however, there is a type-checking component to it. In other words, one can set whether the output should be numeric or character, for example, beforehand. If the output does not match the set type, an error will occur. This function is useful for type-checking your results (i.e., making sure the output matches your expectations).

```
# Mean of all mtcars columns
# Type-check whether it is a numeric vector.
vapply(mtcars, mean, numeric(1))
```

```
##        mpg        cyl       disp         hp       drat         wt       qsec
## 20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750
##         vs         am       gear       carb
##   0.437500   0.406250   3.687500   2.812500
```

```
# Mean of all mtcars columns
# Type-check whether it is a character vector.
vapply(mtcars, mean, character(1))
```

```
## Error in vapply(mtcars, mean, character(1)): values must be type 'character',
##  but FUN(X[[1]]) result is type 'double'
```

9.5 mapply()/Map()

The functions `mapply()` and `Map()` allow us to compute a function iteratively over one or more data inputs.

9.5.1 Univariate Case

In the univariate case, `mapply()`/`Map()` work similarly as `sapply()`/`lapply()`.

```
mapply(mean, mtcars)
```

```
##        mpg        cyl       disp         hp       drat         wt       qsec
## 20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750
##         vs         am       gear       carb
##   0.437500   0.406250   3.687500   2.812500
```

```
head(Map(mean, mtcars)) # Just show a few.
```

```
## $mpg
```

```
## [1] 20.09062
##
## $cyl
## [1] 6.1875
##
## $disp
## [1] 230.7219
##
## $hp
## [1] 146.6875
##
## $drat
## [1] 3.596563
##
## $wt
## [1] 3.21725
```

9.5.2 Multivariate Case

In the multivariate case, we can have multiple data inputs.

```r
# Row bind mpg and wt from mtcars.
# Output = matrix
# Show only a few columns.
mapply(rbind, mtcars$mpg, mtcars$wt)[, 1:5]
```

```
##       [,1]   [,2]  [,3]   [,4]  [,5]
## [1,] 21.00 21.000 22.80 21.400 18.70
## [2,]  2.62  2.875  2.32  3.215  3.44
```

```r
# Row bind mpg and wt from mtcars.
# Output = list.
# Show only a few rows.
head(Map(rbind, mtcars$mpg, mtcars$wt))
```

```
## [[1]]
##       [,1]
## [1,] 21.00
## [2,]  2.62
##
## [[2]]
##        [,1]
## [1,] 21.000
## [2,]  2.875
##
## [[3]]
##       [,1]
```

```
## [1,] 22.80
## [2,]  2.32
##
## [[4]]
##        [,1]
## [1,] 21.400
## [2,]  3.215
##
## [[5]]
##       [,1]
## [1,] 18.70
## [2,]  3.44
##
## [[6]]
##       [,1]
## [1,] 18.10
## [2,]  3.46
```

9.6 rapply()

The `rapply()` function allows one to iterate over a list of datasets recursively. In other words, it allows us to compute the means for all datasets in a list simultaneously, for example.

```
my_list <- list(mtcars, airquality, iris)

rapply(my_list, # For this list...
       # Get all means...
       mean,
       # Remove missing values...
       na.rm = TRUE,
       # Calculate only for numeric columns
       classes = 'numeric')
```

```
##          mpg          cyl         disp           hp         drat           wt
##    20.090625     6.187500   230.721875   146.687500     3.596563     3.217250
##         qsec           vs           am         gear         carb         Wind
##    17.848750     0.437500     0.406250     3.687500     2.812500     9.957516
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
```

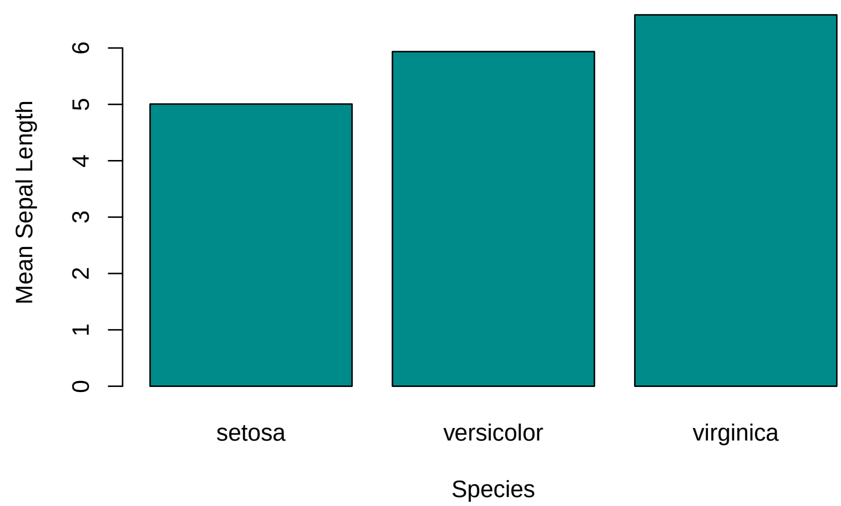
9.7 tapply()

The function `tapply()` makes group-wise computations, outputting a vector as a result. The output being a vector can be useful when passing to other functions, such as `barplot()`. As such, you may want to use `tapply()` when (1) you want your grouped-computation output to be a vector of values and (2) you want to interact the output values with another function.

```
# Let's use iris, a pre-loaded dataset in R.
means <- with(iris, tapply(Sepal.Length, Species, mean))

means
```

```
##     setosa versicolor  virginica
##      5.006      5.936      6.588
```

```
barplot(means, col = 'cyan4', ylab = 'Mean Sepal Length', xlab = 'Species')
```



9.8 aggregate()

Similar to `tapply()`, the function `aggregate()` allows you to make group-wise calculations; however, the output is a data frame rather than a vector. Additionally, you can input multiple independent variables (i.e. variables on the right-hand side of the formula syntax, `y ~ x`). This function may be preferred over `tapply()` when (1) you want multiple grouping variables and (2) you want your output to be in a 2-dimensional format.

```
# Ge thte mean MPG by gear and am.
my_agg <- aggregate(mpg ~ gear + am, mtcars, mean)
```

```r
my_agg
```

```
##   gear am      mpg
## 1    3  0 16.10667
## 2    4  0 21.05000
## 3    4  1 26.27500
## 4    5  1 21.38000
```

9.9 Summary

Table 9.1: Summary of Functionals

Function	Description	Example
lapply(X, FUN)	Compute a function over data and output a list.	lapply(mtcars, mean)
sapply(X, FUN)	Compute a function over data and output a matrix (sometimes a list, depending on the function being passed).	sapply(mtcars, mean)
apply(X, MARGIN, FUN)	Compute a function row-wise or column-wise.	apply(mtcars, 1, mean); apply(mtcars, 2, mean)
vapply(X, FUN, FUN.VALUE)	Compute a function over data and check if the output matches a pre-specified type.	vapply(mtcars, mean, numeric(1))
mapply(FUN, ...)	Compute a function over one or more data inputs and output an array (vector or matrix).	mapply(rbind, mtcars\$mpg, mtcars\$wt)
Map(f, ...)	Compute a function over one or more data inputs and output a list.	Map(rbind, mtcars\$mpg, mtcars\$wt)
rapply(object, f, classes)	Recursively compute a function over data and output a vector or list.	rapply(iris, mean, classes = "numeric")
tapply(X, INDEX, FUN)	Generate grouped computations and output a vector.	with(iris, tapply(Sepal.Length, Species, mean))
aggregate(formula, data, FUN)	Generate grouped computations and output a data frame.	aggregate(mpg ~ gear, mtcars, mean)

Chapter 10

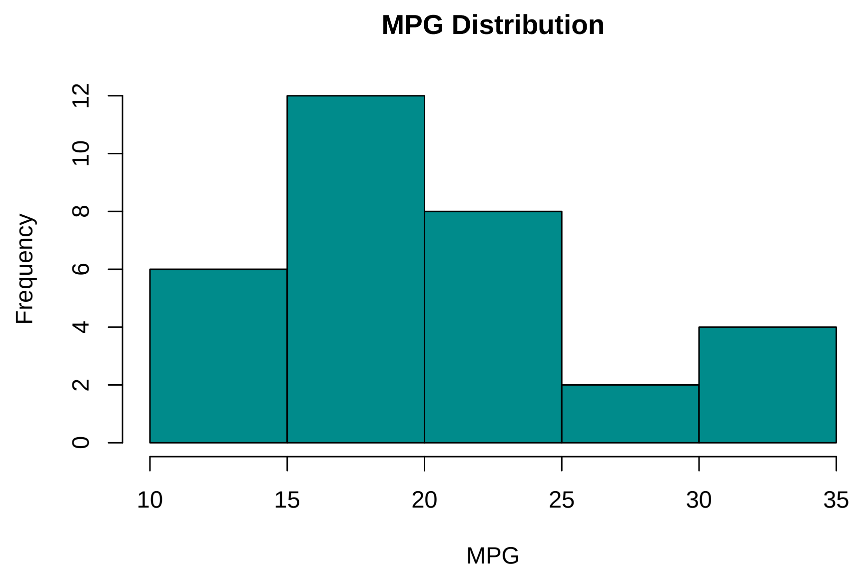
Graphing

The following subsections show examples of how to create certain types of graphs.

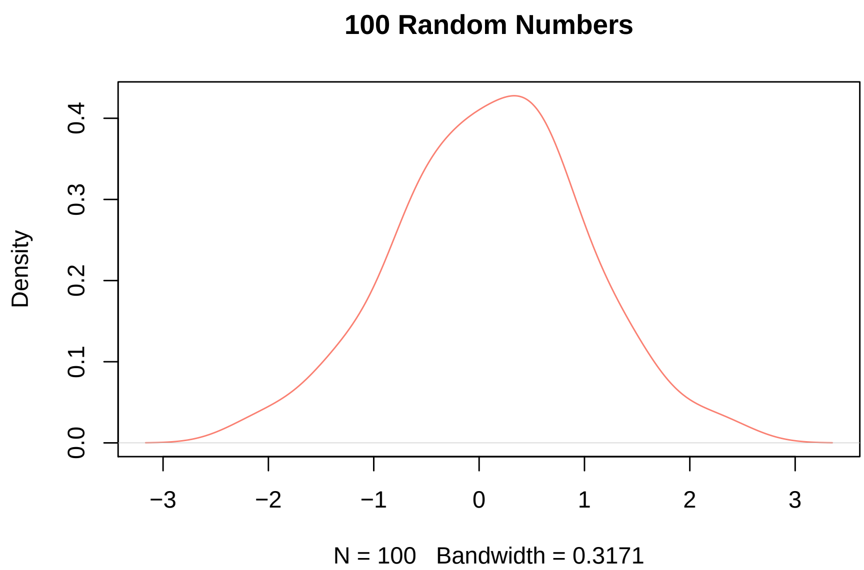
10.1 Histograms

All we need to make a histogram is to pass a vector into the `hist()` function.

```
hist(mtcars$mpg,
     col = 'cyan4',
     xlab = 'MPG',
     ylab = 'Frequency',
     main = 'MPG Distribution')
```




```r
set.seed(1) # Remember our random numbers.

x <- density(rnorm(100))

plot(x,
     main = '100 Random Numbers',
     col  = 'salmon')
```

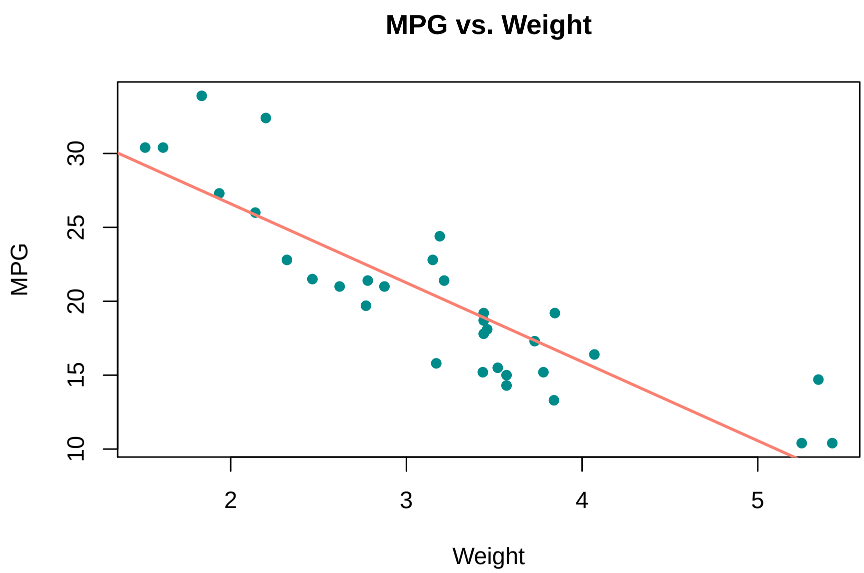


10.3 Scatter Plots

To make a scatter plot, we make use of the `plot(formula)` function, where `formula` input is of the syntax `y ~ x` (y relates to the y-axis and relates to the x-axis).

10.3.1 Simple Scatter Plot

```
# Draw the scatter plot
with(mtcars,
     plot(mpg ~ wt,
          ylab = 'MPG',
          xlab = 'Weight',
          main = 'MPG vs. Weight',
          col = 'cyan4',
          pch = 16)) # pch determines point type.

# Draw a trend line over the scatter plot
abline(lm(mpg ~ wt, mtcars),
       col = 'salmon',
       lwd = 2) # line width
```

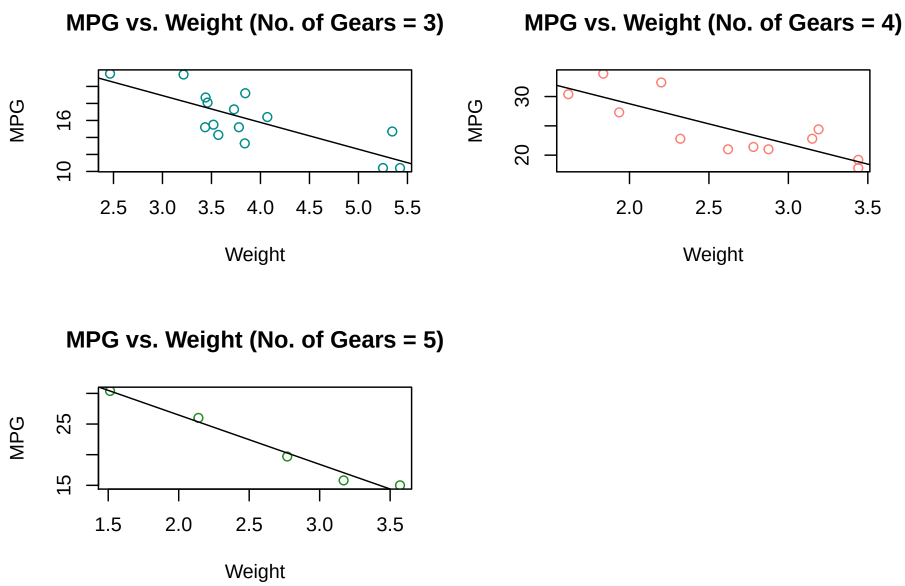


10.3.2 Multiple Scatter Plots

For a more complex example, let's make multiple scatter plots via a `for` loop.

```
# Set up a 2x2 canvas
par(mfrow = c(2,2))

# Set parameters
unique_gears <- sort(unique(mtcars$gear))

mycolors <- c('cyan4', 'salmon','forestgreen', 'purple')

# Begin plot loop
for (i in seq_along(unique_gears)) {

  # Subset by number of gears
  ss <- subset(mtcars, gear == unique_gears[i])

  # Plot a scatter points
  with(ss,
       plot(mpg ~ wt,
            col = mycolors[i],
            ylab = 'MPG',
            xlab = 'Weight',
            main = paste0('MPG vs. Weight (No. of Gears = ',
```

```
                                  unique_gears[i],
                                  ')')))

  # Generate a trendline for each subset.
  abline(lm(mpg ~ wt, ss))

}
```



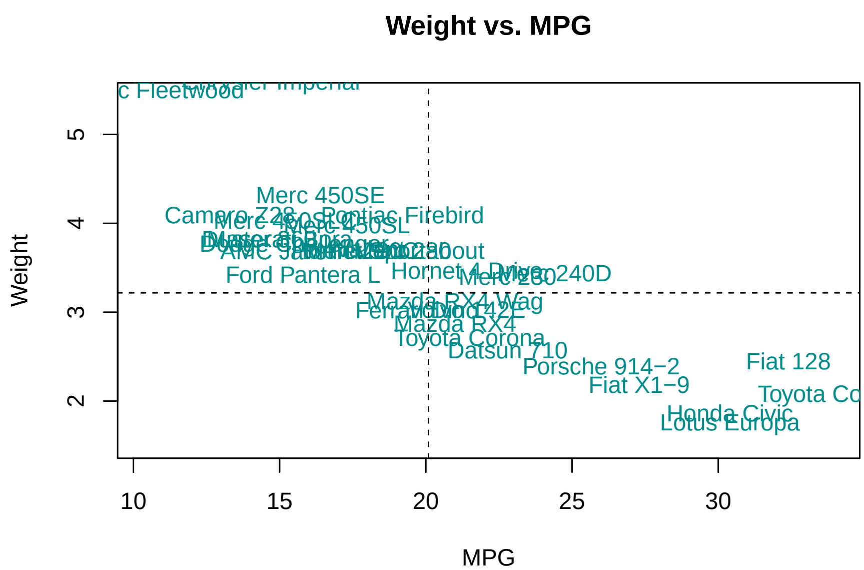
Notice how "purple" isn't used in graphs, as there are only three sub-graphs to plot

10.3.3 Text Plot

To make a text plot, we just turn off the points in the `plot()` function via `type = 'n'` and then use the `text()` function to label them on the graph.

```
# Set up basic plot.
with(mtcars, plot(wt ~ mpg, pch = 1, type = 'n',
                  xlab = 'MPG',
                  ylab = 'Weight',
                  main = 'Weight vs. MPG'))

# Plot the labels on the graph.
with(mtcars, text(mpg, # x coordinate for words
                  wt,  # y coordinate for words
                  row.names(mtcars), # Words to use.
```

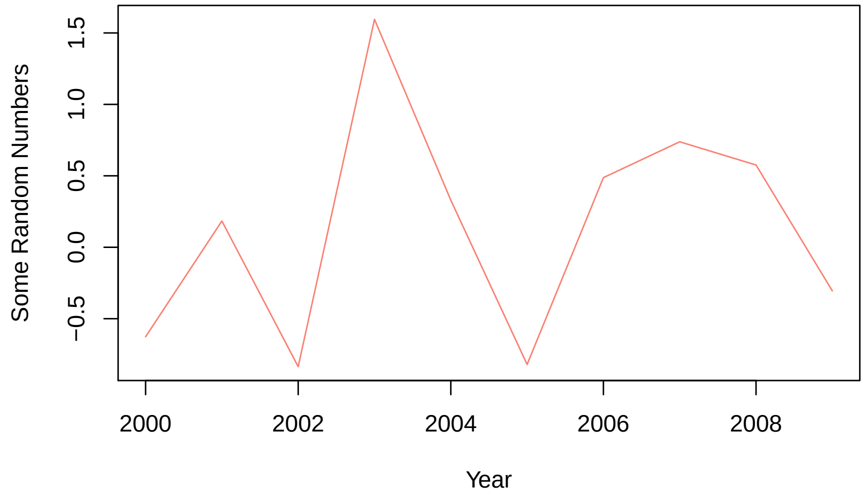
```
                   pos = 3,
                   cex = 0.0,
                   col = 'cyan4'))

# Add some mean lines for flair.
abline(h = mean(mtcars$wt), v = mean(mtcars$mpg),
       lty = 2)
```



10.4 Line Plots

Making a line plot is similar to making a scatter plot except that we set `type = 'l'` as an additional input.

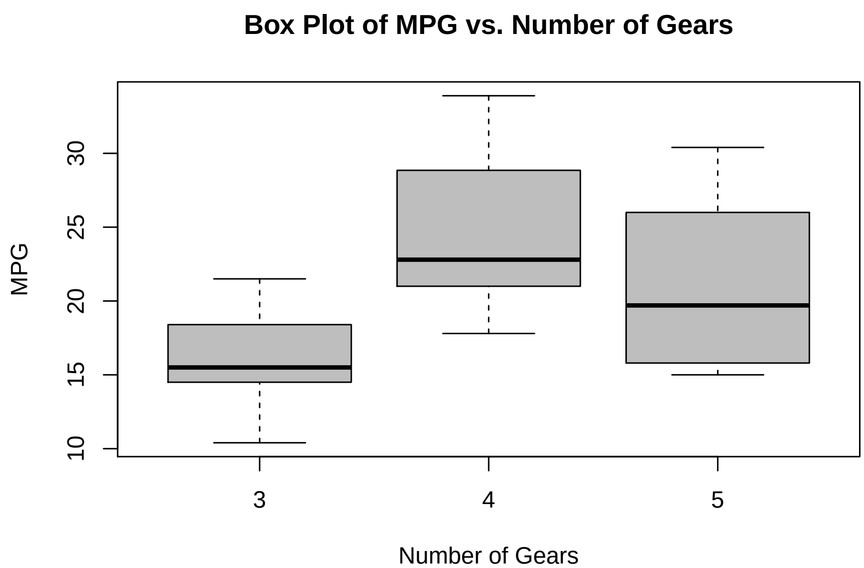
```
# Suppose we had this dataset:
set.seed(1) # Remember our random numbers.
df <- data.frame(y = rnorm(10),
                 x = 2000:2009)

# Plot this dataset
with(df, plot(y ~ x,
              type = 'l',
              col = 'salmon',
              ylab = 'Some Random Numbers',
              xlab = 'Year'))
```



10.5 Box Plots

Constructing a box plot with the `boxplot()` function is similar to making a scatter plot with `plot()`: we pass a formula of vectors into it.

```
with(mtcars,
     boxplot(mpg ~ gear,
             ylab = 'MPG',
             xlab = 'Number of Gears',
             main = 'Box Plot of MPG vs. Number of Gears',
             col  = 'grey'))
```



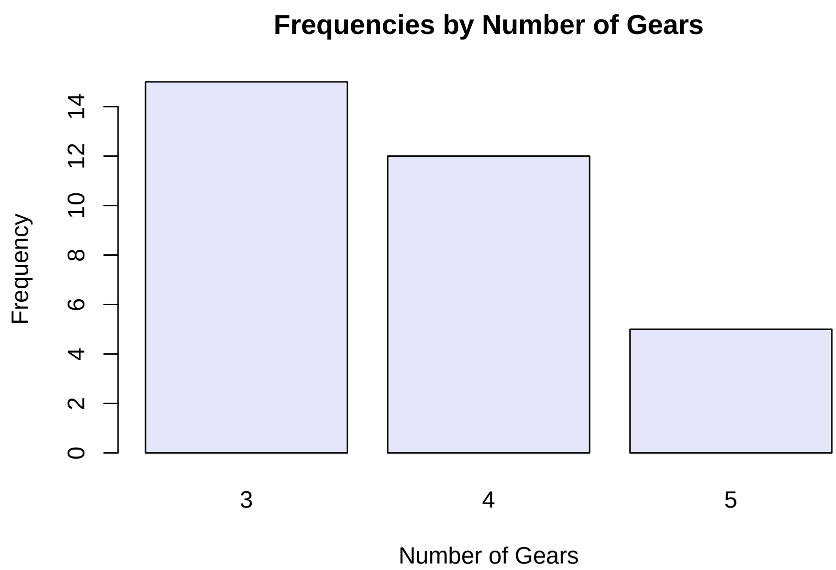
10.6 Bar Plots

For a bar plot, we pass a vector (usually one of counts) or aggregation to `barplot()`.

10.6.1 Frequency Chart

For a frequency chart, we have to calculate a table of frequencies with the `table()` function before passing it to `barplot()`.

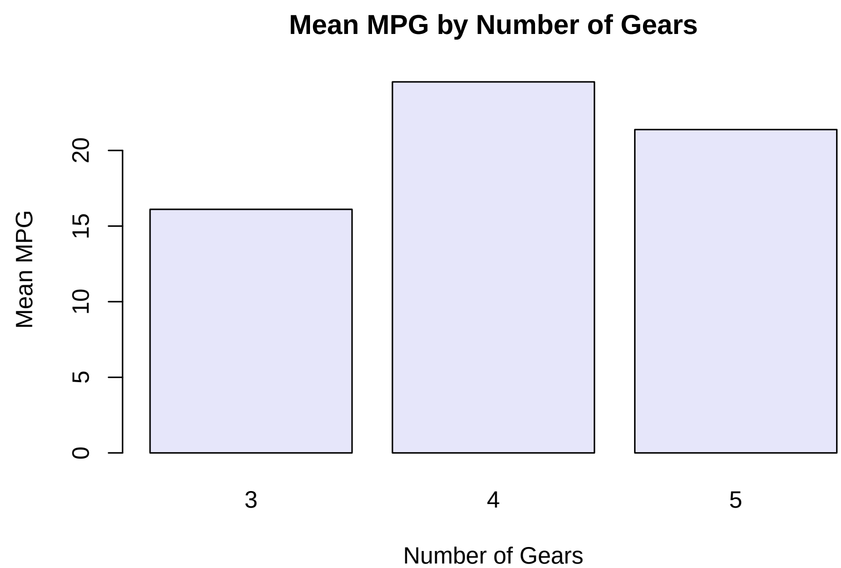
```
my_table <- table(mtcars$gear)

barplot(my_table,
        ylab = 'Frequency',
        xlab = 'Number of Gears',
        col  = 'lavender',
        main = 'Frequencies by Number of Gears')
```



10.6.2 Grouped Mean Comparisons

For grouped mean comparisons, we have to aggregate data with `aggregate()` (see the `Functionals` chapter for more details) before passing it to `barplot()`.

```
my_agg <- aggregate(mpg ~ gear, mtcars, mean)

with(my_agg,
     barplot(mpg ~ gear,
             beside = TRUE, # Set to FALSE to stack bars.
             ylab = 'Mean MPG',
             xlab = 'Number of Gears',
             main = 'Mean MPG by Number of Gears',
             col  = 'lavender'))
```



10.7 Summary

Table 10.1: Summary of Graphing Functions

Function	Description	Example
hist(x)	Histogram	hist(mtcars\$mpg)
plot(density(x))	Density plot	plot(density(rnorm(100)))
plot(y ~ x)	Scatter plot	with(mtcars, plot(mpg ~ wt))
plot(y ~ x, type = 'l')	Line plot	with(Orange, plot(circumference ~ age, type = 'l'))
boxplot(y ~ x)	Box plot	with(mtcars, boxplot(mpg ~ wt))
barplot(x)	Bar plot	barplot(table(mtcars\$gear))

Chapter 11

Hypothesis Testing

In this chapter, we will cover how to conduct a t-test of means and chi-square test of frequencies.

11.1 t-test

To conduct a t-test, we use the `t.test()` function. What we input into this function depends on whether we want to compute a one-sample or two-sample test.

11.1.1 One-sample t-test

To conduct a 1-sample t-test, we pass a vector and a `mu` value into the `t.test()` function. The `mu` value is the number against which we will compare the vector's mean to determine whether there is a statistically significant difference.

```
# Testing whether the mean MPG is statistically equal to 17.
t.test(mtcars$mpg, mu = 17)
```

```
##
##  One Sample t-test
##
## data:  mtcars$mpg
## t = 2.9008, df = 31, p-value = 0.006788
## alternative hypothesis: true mean is not equal to 17
## 95 percent confidence interval:
##  17.91768 22.26357
## sample estimates:
## mean of x
##  20.09062
```

11.1.2 Two-sample t-test

To conduct a two-sample t-test, we use the formula syntax of `y ~ x`, where `y` is our continuous dependent variable and `x` is our categorical independent variable. Then, we pass this formula into `t.test()`.

```
# Compare mean MPG by transmission type
with(mtcars, t.test(mpg ~ am))
```

```
##
##  Welch Two Sample t-test
##
## data:  mpg by am
## t = -3.7671, df = 18.332, p-value = 0.001374
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -11.280194  -3.209684
## sample estimates:
## mean in group 0 mean in group 1
##        17.14737        24.39231
```

11.2 Chi-square test

To conduct a Chi-square test, we pass a two-way `table` into the `chisq.test()` function.

```
mytable <- with(mtcars, table(gear, am))

mytable
```

```
##     am
## gear  0  1
##    3 15  0
##    4  4  8
##    5  0  5
```

```
chisq.test(mytable)
```

```
## Warning in chisq.test(mytable): Chi-squared approximation may be incorrect
```

```
##
##  Pearson's Chi-squared test
##
## data:  mytable
## X-squared = 20.945, df = 2, p-value = 2.831e-05
```

11.3 Summary

Table 11.1: Summary of Hypothesis Testing

Function	Description	Example
t.test(x, mu)	Test of mean against mu.	t.test(mtcars$mpg, mu = 17)
t.test(y ~ x)	Test of group means.	with(mtcars, t.test(mpg ~ am))
chisq.test(table)	Test of two-way frequencies.	with(mtcars, chisq.test(table(gear, am)))

Chapter 12

Linear Modeling

In this chapter, we will examine Pearson correlations, ANOVA, Ordinary Least Squares, and logistic regression.

12.1 Pearson Correlations

To estimate a Pearson correlation for all variables in a dataset, we pass a `matrix` or `data frame` into the `cor()` function.

```
# Pearson correlation coefficient matrix
cor(mtcars)
```

```
##             mpg        cyl       disp         hp        drat         wt
## mpg   1.0000000 -0.8521620 -0.8475514 -0.7761684  0.68117191 -0.8676594
## cyl  -0.8521620  1.0000000  0.9020329  0.8324475 -0.69993811  0.7824958
## disp -0.8475514  0.9020329  1.0000000  0.7909486 -0.71021393  0.8879799
## hp   -0.7761684  0.8324475  0.7909486  1.0000000 -0.44875912  0.6587479
## drat  0.6811719 -0.6999381 -0.7102139 -0.4487591  1.00000000 -0.7124406
## wt   -0.8676594  0.7824958  0.8879799  0.6587479 -0.71244065  1.0000000
## qsec  0.4186840 -0.5912421 -0.4336979 -0.7082234  0.09120476 -0.1747159
## vs    0.6640389 -0.8108118 -0.7104159 -0.7230967  0.44027846 -0.5549157
## am    0.5998324 -0.5226070 -0.5912270 -0.2432043  0.71271113 -0.6924953
## gear  0.4802848 -0.4926866 -0.5555692 -0.1257043  0.69961013 -0.5832870
## carb -0.5509251  0.5269883  0.3949769  0.7498125 -0.09078980  0.4276059
##             qsec         vs          am       gear        carb
## mpg   0.41868403  0.6640389  0.59983243  0.4802848 -0.55092507
## cyl  -0.59124207 -0.8108118 -0.52260705 -0.4926866  0.52698829
## disp -0.43369788 -0.7104159 -0.59122704 -0.5555692  0.39497686
## hp   -0.70822339 -0.7230967 -0.24320426 -0.1257043  0.74981247
## drat  0.09120476  0.4402785  0.71271113  0.6996101 -0.09078980
## wt   -0.17471588 -0.5549157 -0.69249526 -0.5832870  0.42760594
```

```
## qsec  1.00000000  0.7445354 -0.22986086 -0.2126822 -0.65624923
## vs    0.74453544  1.0000000  0.16834512  0.2060233 -0.56960714
## am   -0.22986086  0.1683451  1.00000000  0.7940588  0.05753435
## gear -0.21268223  0.2060233  0.79405876  1.0000000  0.27407284
## carb -0.65624923 -0.5696071  0.05753435  0.2740728  1.00000000
```

To perform a correlation test in which we produce a p-value, we pass two vectors into the `cor.test()` function.

```
# Pearson correlation coefficient test
with(mtcars, cor.test(mpg, wt))
```

```
##
##  Pearson's product-moment correlation
##
## data:  mpg and wt
## t = -9.559, df = 30, p-value = 1.294e-10
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  -0.9338264 -0.7440872
## sample estimates:
##        cor
## -0.8676594
```

To get p-values from a correlation matrix for all variables, we will use the `Hmisc` package. We install it with `install.packages()` and then load it with `library()`. We use the library's `rcorr()` function to calculate the correlation and p-values matrices.

```
install.packages('Hmisc') # Install first.
```

```
# Load the library into the environment.
library(Hmisc)

my_corr <- rcorr(as.matrix(mtcars), type = 'pearson')

# Pearson correlation coefficients
my_corr$r
```

```
##             mpg        cyl       disp         hp        drat         wt
## mpg   1.0000000 -0.8521620 -0.8475514 -0.7761684  0.68117191 -0.8676594
## cyl  -0.8521620  1.0000000  0.9020329  0.8324475 -0.69993811  0.7824958
## disp -0.8475514  0.9020329  1.0000000  0.7909486 -0.71021393  0.8879799
## hp   -0.7761684  0.8324475  0.7909486  1.0000000 -0.44875912  0.6587479
## drat  0.6811719 -0.6999381 -0.7102139 -0.4487591  1.00000000 -0.7124406
## wt   -0.8676594  0.7824958  0.8879799  0.6587479 -0.71244065  1.0000000
## qsec  0.4186840 -0.5912421 -0.4336979 -0.7082234  0.09120476 -0.1747159
## vs    0.6640389 -0.8108118 -0.7104159 -0.7230967  0.44027846 -0.5549157
```

```
## am    0.5998324 -0.5226070 -0.5912270 -0.2432043  0.71271113 -0.6924953
## gear  0.4802848 -0.4926866 -0.5555692 -0.1257043  0.69961013 -0.5832870
## carb -0.5509251  0.5269883  0.3949769  0.7498125 -0.09078980  0.4276059
##             qsec         vs          am       gear        carb
## mpg   0.41868403  0.6640389  0.59983243  0.4802848 -0.55092507
## cyl  -0.59124207 -0.8108118 -0.52260705 -0.4926866  0.52698829
## disp -0.43369788 -0.7104159 -0.59122704 -0.5555692  0.39497686
## hp   -0.70822339 -0.7230967 -0.24320426 -0.1257043  0.74981247
## drat  0.09120476  0.4402785  0.71271113  0.6996101 -0.09078980
## wt   -0.17471588 -0.5549157 -0.69249526 -0.5832870  0.42760594
## qsec  1.00000000  0.7445354 -0.22986086 -0.2126822 -0.65624923
## vs    0.74453544  1.0000000  0.16834512  0.2060233 -0.56960714
## am   -0.22986086  0.1683451  1.00000000  0.7940588  0.05753435
## gear -0.21268223  0.2060233  0.79405876  1.0000000  0.27407284
## carb -0.65624923 -0.5696071  0.05753435  0.2740728  1.00000000
```

```
# p-values of the coefficients.
my_corr$P
```

```
##               mpg          cyl         disp           hp         drat
## mpg            NA 6.112688e-10 9.380328e-10 1.787835e-07 1.776240e-05
## cyl  6.112688e-10           NA 1.803002e-12 3.477861e-09 8.244636e-06
## disp 9.380328e-10 1.803002e-12           NA 7.142679e-08 5.282022e-06
## hp   1.787835e-07 3.477861e-09 7.142679e-08           NA 9.988772e-03
## drat 1.776240e-05 8.244636e-06 5.282022e-06 9.988772e-03           NA
## wt   1.293958e-10 1.217567e-07 1.222311e-11 4.145827e-05 4.784260e-06
## qsec 1.708199e-02 3.660533e-04 1.314404e-02 5.766253e-06 6.195826e-01
## vs   3.415937e-05 1.843018e-08 5.235012e-06 2.940896e-06 1.167553e-02
## am   2.850207e-04 2.151207e-03 3.662114e-04 1.798309e-01 4.726790e-06
## gear 5.400948e-03 4.173297e-03 9.635921e-04 4.930119e-01 8.360110e-06
## carb 1.084446e-03 1.942340e-03 2.526789e-02 7.827810e-07 6.211834e-01
##                wt         qsec           vs           am         gear
## mpg  1.293958e-10 1.708199e-02 3.415937e-05 2.850207e-04 5.400948e-03
## cyl  1.217567e-07 3.660533e-04 1.843018e-08 2.151207e-03 4.173297e-03
## disp 1.222311e-11 1.314404e-02 5.235012e-06 3.662114e-04 9.635921e-04
## hp   4.145827e-05 5.766253e-06 2.940896e-06 1.798309e-01 4.930119e-01
## drat 4.784260e-06 6.195826e-01 1.167553e-02 4.726790e-06 8.360110e-06
## wt             NA 3.388683e-01 9.798492e-04 1.125440e-05 4.586601e-04
## qsec 3.388683e-01           NA 1.029669e-06 2.056621e-01 2.425344e-01
## vs   9.798492e-04 1.029669e-06           NA 3.570439e-01 2.579439e-01
## am   1.125440e-05 2.056621e-01 3.570439e-01           NA 5.834043e-08
## gear 4.586601e-04 2.425344e-01 2.579439e-01 5.834043e-08           NA
## carb 1.463861e-02 4.536949e-05 6.670496e-04 7.544526e-01 1.290291e-01
##              carb
## mpg  1.084446e-03
## cyl  1.942340e-03
```

```
## disp 2.526789e-02
## hp   7.827810e-07
## drat 6.211834e-01
## wt   1.463861e-02
## qsec 4.536949e-05
## vs   6.670496e-04
## am   7.544526e-01
## gear 1.290291e-01
## carb           NA
```

12.2 ANOVA

To conduct ANOVA, we pass a formula and dataset into the `aov()` function. Note that the independent variables must be `factor` variables, so we must use the `factor()` function on our independent variables if they are not already factors.

```
my_anova <- aov(mpg ~ factor(gear) + factor(am), mtcars)

my_anova
```

```
## Call:
##    aov(formula = mpg ~ factor(gear) + factor(am), data = mtcars)
##
## Terms:
##                 factor(gear) factor(am) Residuals
## Sum of Squares      483.2432    72.8017  570.0023
## Deg. of Freedom            2          1        28
##
## Residual standard error: 4.511898
## Estimated effects may be unbalanced
```

```
summary(my_anova)
```

```
##              Df Sum Sq Mean Sq F value   Pr(>F)
## factor(gear)  2  483.2  241.62  11.869 0.000185 ***
## factor(am)    1   72.8   72.80   3.576 0.069001 .
## Residuals    28  570.0   20.36
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

To compare pairwise means, we use `TukeyHSD()` on our ANOVA model.

```
TukeyHSD(my_anova)
```

```
##   Tukey multiple comparisons of means
##     95% family-wise confidence level
##
```

```
## Fit: aov(formula = mpg ~ factor(gear) + factor(am), data = mtcars)
##
## $`factor(gear)`
##          diff        lwr       upr     p adj
## 4-3  8.426667  4.1028616 12.750472 0.0001301
## 5-3  5.273333 -0.4917401 11.038407 0.0779791
## 5-4 -3.153333 -9.0958350  2.789168 0.3999532
##
## $`factor(am)`
##         diff       lwr     upr     p adj
## 1-0 1.805128 -1.521483 5.13174 0.2757926
```

12.3 Ordinary Least Squares

To estimate a regression model, we pass a formula and a dataset into the `lm()` function.

```
# SYNTAX OF lm(): lm(y ~ x1 + x2 + ... xn, data)
my_ols <- lm(mpg ~ wt + hp + gear + am, mtcars)

# Return the coefficients
my_ols
```

```
##
## Call:
## lm(formula = mpg ~ wt + hp + gear + am, data = mtcars)
##
## Coefficients:
## (Intercept)           wt           hp         gear           am
##    32.55626     -2.79996     -0.03837      0.40299      1.68739
```

```
# Produce a summary table of the results.
summary(my_ols)
```

```
##
## Call:
## lm(formula = mpg ~ wt + hp + gear + am, data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.2986 -1.9652 -0.4584  1.1434  5.6766
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 32.55626    4.67171   6.969 1.72e-07 ***
## wt          -2.79996    0.94234  -2.971 0.006164 **
## hp          -0.03837    0.01004  -3.823 0.000706 ***
```

```
## gear         0.40299    1.06519   0.378 0.708145
## am           1.68739    1.74691   0.966 0.342651
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.577 on 27 degrees of freedom
## Multiple R-squared:  0.8407, Adjusted R-squared:  0.8171
## F-statistic: 35.63 on 4 and 27 DF,  p-value: 2.091e-10
```

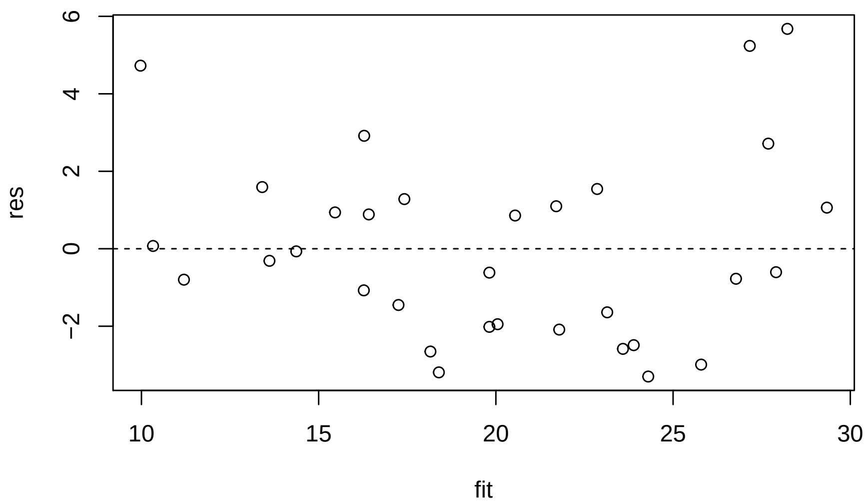
```
# Return the coefficient table from the summary regression table.
coef(summary(my_ols))
```

```
##                Estimate Std. Error    t value     Pr(>|t|)
## (Intercept) 32.55625619 4.67170949  6.9688101 1.723405e-07
## wt          -2.79995626 0.94234225 -2.9712732 6.164196e-03
## hp          -0.03837417 0.01003886 -3.8225618 7.063674e-04
## gear         0.40299281 1.06519249  0.3783286 7.081449e-01
## am           1.68739402 1.74690861  0.9659315 3.426513e-01
```

12.3.1 Residual diagnostics with OLS

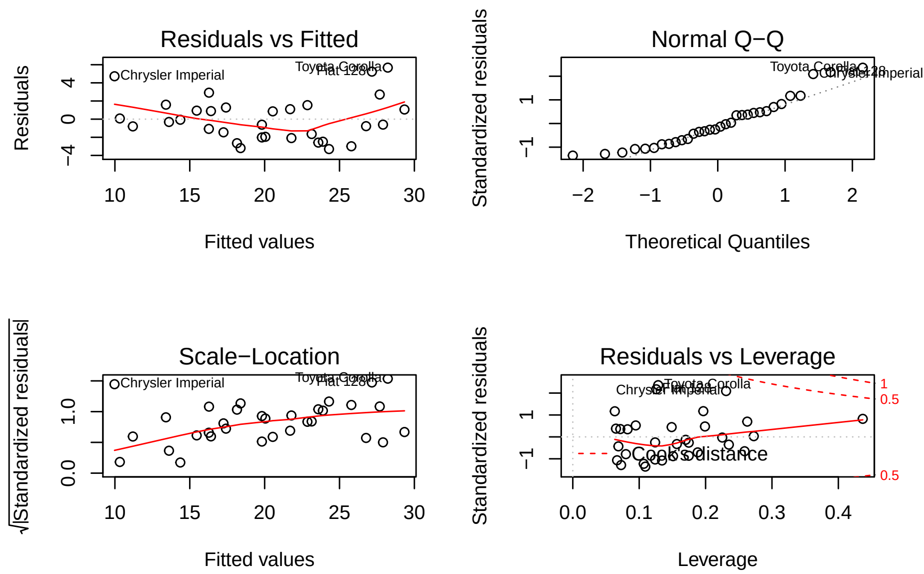
To analyze the performance of our models with respect to our residuals, we can calculate the predicted values with `predict()` and residuals with `resid()`. We can then plot them to see whether the residuals behave in a homoskedastic manner.

```
fit <- predict(my_ols)
res <- resid(my_ols)

plot(res ~ fit)
abline(lm(res ~ fit), lty = 2)
```



Alternatively, we can directly plot our model. Make sure to set a 2-by-2 canvas beforehand so that all the plots from `plot()` will generate simultaneously.

```
par(mfrow = c(2,2)) # Set 2x2 canvas

plot(my_ols)
```



12.4 Logistic Regression

Estimating a logistic regression is similar to estimating a model with OLS; however, we add an additional input in which we set the distribution family–in this case, it is the binomial one.

```
my_logit <- glm(am ~ mpg + wt + gear,
                mtcars,
                family = binomial(link = 'logit'))
```

```
## Warning: glm.fit: algorithm did not converge
```

```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
my_logit
```

```
##
## Call:  glm(formula = am ~ mpg + wt + gear, family = binomial(link = "logit"),
##     data = mtcars)
##
## Coefficients:
## (Intercept)          mpg           wt         gear
##     137.764       -6.548     -113.946       87.125
##
## Degrees of Freedom: 31 Total (i.e. Null);  28 Residual
## Null Deviance:       43.23
```

```
## Residual Deviance: 2.765e-09     AIC: 8
```

```r
summary(my_logit)
```

```
##
## Call:
## glm(formula = am ~ mpg + wt + gear, family = binomial(link = "logit"),
##     data = mtcars)
##
## Deviance Residuals:
##        Min          1Q      Median          3Q         Max
## -2.415e-05  -2.100e-08  -2.100e-08   2.100e-08   3.585e-05
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)   137.764 324199.947   0.000    1.000
## mpg            -6.548   8893.588  -0.001    0.999
## wt           -113.946  95316.944  -0.001    0.999
## gear           87.125  71730.620   0.001    0.999
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 4.3230e+01  on 31  degrees of freedom
## Residual deviance: 2.7646e-09  on 28  degrees of freedom
## AIC: 8
##
## Number of Fisher Scoring iterations: 25
```

12.5 Summary

Table 12.1: Summary of Linear Modeling

Function	Description	Example
cor(data)	Correlation matrix.	cor(mtcars)
rcorr(data)	Correlation matrix with p-values.	library(Hmisc); rcorr(as.matrix(mtcars), type = 'pearson'
aov(y ~ x, data)	ANOVA.	aov(mpg ~ factor(gear), mtcars)
TukeyHSD(anova)	Tukey HSD pairwise means.	TukeyHSD(aov(mpg ~ factor(gear), mtcars))
lm(y ~ x, data)	Linear Modeling / Ordinary Least Squares modeling.	lm(mpg ~ wt + gear, mtcars)
glm(y ~ x, data, family)	Generalized Linear Model.	glm(am ~ mpg + gear, mtcars, family = binomial(link = 'logit'))

Chapter 13

Recommended R Libraries

The following is a list of recommended R libraries to install–they can be helpful for data management, graphing, and formatting.

13.1 tidyverse

The `tidyverse` package is a metapackage consisting of other libraries. The most useful ones for a beginner, I believe, are `ggplot2`, `dplyr`, `tidyr`, and `purrr`.

For more information, see the `tidyverse` website.

```
install.packages('tidyverse')
```

```
library(tidyverse)
```

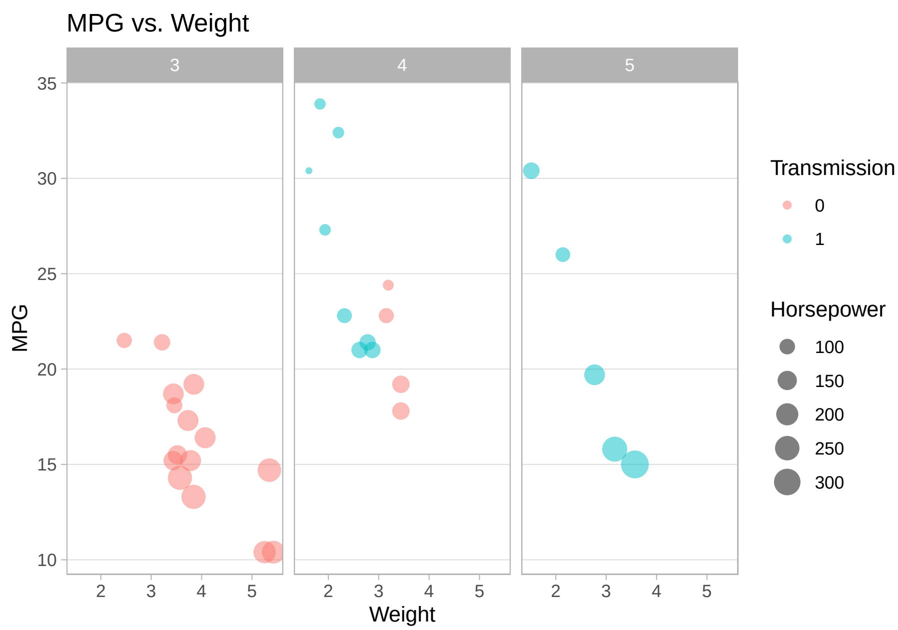
13.1.1 ggplot2

The library `ggplot2` offers visualization tools with a modern aesthetic. The following is an example of a small-multiples[1] scatter plot. For more information, see the `ggplot2` website.

```
ggplot(mtcars) +
  aes(y = mpg, x = wt, col = factor(am), size = hp) +
  geom_point(alpha = 0.5) +
  labs(y     = 'MPG',
       x     = 'Weight',
       col   = 'Transmission',
       size  = 'Horsepower',
       title = 'MPG vs. Weight') +
  facet_wrap(~ gear) +
```

[1] https://en.wikipedia.org/wiki/Small_multiple

```
  theme_light() +
  theme(panel.grid.minor   = element_blank(),
        panel.grid.major.x = element_blank())
```



13.1.2 dplyr

The `dplyr` library provides aggregation tools for data management. The following is an example of calculating the mean and median MPG by gear.

For more information, see the `dplyr` website.

```
my_agg <- mtcars %>%
  select(mpg, gear) %>%
  group_by(gear) %>%
  summarise(mean_mpg = mean(mpg),
            median_mpg = median(mpg))

my_agg
```

```
## # A tibble: 3 x 3
##    gear mean_mpg median_mpg
##   <dbl>    <dbl>      <dbl>
## 1     3     16.1       15.5
## 2     4     24.5       22.8
## 3     5     21.4       19.7
```

13.1.3 tidyr

The `tidyr` library provides pivoting tools to reshape your dataset. The following are examples of how to reformat an aggregation from `dplyr`'s functions.

For more information, see the `tidyr` website.

```
# Aggregation
my_agg <- mtcars %>%
  select(mpg, gear, am) %>%
  group_by(gear, am) %>%
  summarise(mean_mpg = mean(mpg))

# Pivot wide
my_agg2 <- my_agg %>%
  pivot_wider(id_cols     = gear, # rows
              names_from  = am, # columns
              values_from = mean_mpg) # values

my_agg2
```

```
## # A tibble: 3 x 3
## # Groups:   gear [3]
##    gear   `0`   `1`
##   <dbl> <dbl> <dbl>
## 1     3  16.1  NA  
## 2     4  21.0  26.3
## 3     5  NA    21.4
```

```
# Pivot long
my_agg2 %>%
  pivot_longer(2:3,
               names_to  = 'am',
               values_to = 'mpg',
               values_drop_na = TRUE) # drop NA values
```

```
## # A tibble: 4 x 3
## # Groups:   gear [3]
##    gear am      mpg
##   <dbl> <chr> <dbl>
## 1     3 0      16.1
## 2     4 0      21.0
## 3     4 1      26.3
## 4     5 1      21.4
```

13.1.4 purrr

The purrr library offers functionals similar to the *apply() functions (the former's map() operates similarly as the latter's lapply()); however, the former contains functions that maintain type consistency. For example, there is a function called map_dbl() that throws an error if the output is not a double vector (i.e., a numeric vector), which is useful when you want to catch your program's errors.

The following are some examples from purrr. For more information on how to use these and other functions within the library, see the purrr website.

```
map(mtcars, mean) # == lapply(mtcars, mean)
```

```
## $mpg
## [1] 20.09062
##
## $cyl
## [1] 6.1875
##
## $disp
## [1] 230.7219
##
## $hp
## [1] 146.6875
##
## $drat
## [1] 3.596563
##
## $wt
## [1] 3.21725
##
## $qsec
## [1] 17.84875
##
## $vs
## [1] 0.4375
##
## $am
## [1] 0.40625
##
## $gear
## [1] 3.6875
##
## $carb
## [1] 2.8125
```

```
map_dbl(mtcars, mean) # == sapply(mtcars, mean)
```

```
##        mpg        cyl       disp         hp       drat         wt       qsec
##  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750
##         vs         am       gear       carb
##   0.437500   0.406250   3.687500   2.812500
```

```
map_df(mtcars, mean) # Maintains data frame class.
```

```
## # A tibble: 1 x 11
##     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1  20.1  6.19  231.  147.  3.60  3.22  17.8 0.438 0.406  3.69  2.81
```

13.2 knitr

The `knitr` library is an "engine for dynamic report generation," which allows for better formatted tables and documentation capabilities when using R Markdown.[2] The following example demonstrates `kable()` to format a table.

```
install.packages('knitr')
```

```
library(knitr)

my_table <- with(mtcars, table(gear, am))

kable(my_table)
```

	0	1
3	15	0
4	4	8
5	0	5

13.3 stargazer

The `stargazer` library allows one to format a regression model to be closer to journal-quality guidelines.

For more information, see its documentation on CRAN.

```
install.packages('stargazer')
```

```
library(stargazer)
```

```
##
## Please cite as:
```

[2] https://yihui.org/knitr/

```
##  Hlavac, Marek (2018). stargazer: Well-Formatted Regression and Summary Statistics
##  R package version 5.2.2. https://CRAN.R-project.org/package=stargazer
```

```
my_ols <- lm(mpg ~ wt + hp + disp + gear + am, mtcars)
```

If you are using RGui or R Studio and not R Markdown, I recommend to set `type = 'text'` so that only textual output will be produced instead of LaTeX or HTML code.

```
# If NOT using R Markdown...
stargazer(my_ols, type = 'text')
```

```
##
## ===============================================
##                         Dependent variable:
##                     ---------------------------
##                                 mpg
## -----------------------------------------------
## wt                          -3.113**
##                              (1.179)
##
## hp                          -0.043***
##                              (0.014)
##
## disp                          0.005
##                              (0.012)
##
## gear                          0.652
##                              (1.212)
##
## am                            1.605
##                              (1.782)
##
## Constant                    32.108***
##                              (4.844)
##
## -----------------------------------------------
## Observations                   32
## R2                            0.842
## Adjusted R2                   0.812
## Residual Std. Error      2.616 (df = 26)
## F Statistic          27.709*** (df = 5; 26)
## ===============================================
## Note:               *p<0.1; **p<0.05; ***p<0.01
```

If you happen to use R Markdown, then set `type = 'html'` for HTML documents and omit `type` for PDF documents.

For more on R Markdown, see the R Markdown book by Yihui Xie, J. J. Allaire, and Garrett Grolemund.

```
# If using RMarkdown...
# stargazer(my_ols, type = 'html') # for html documents.
stargazer(my_ols) # for PDF documents.
```

% Table created by stargazer v.5.2.2 by Marek Hlavac, Harvard University. E-mail: hlavac at fas.harvard.edu % Date and time: Wed, May 13, 2020 - 3:53:42 PM

Table 13.1:

	Dependent variable:
	mpg
wt	-3.113^{**}
	(1.179)
hp	-0.043^{***}
	(0.014)
disp	0.005
	(0.012)
gear	0.652
	(1.212)
am	1.605
	(1.782)
Constant	32.108^{***}
	(4.844)
Observations	32
R^2	0.842
Adjusted R^2	0.812
Residual Std. Error	2.616 (df = 26)
F Statistic	27.709^{***} (df = 5; 26)
Note:	*p<0.1; **p<0.05; ***p<0.01

13.4 Summary

Table 13.2: Summary of Recommended Libraries

Library	Function	Description	Example
ggplot2	ggplot(data) + aes(y, x, ...) + geom_point()	Scatter plot with ggplot2.	ggplot(mtcars) + aes(y = mpg, x = wt, col = factor(am), size = hp) + geom_point(alpha = 0.5)
dplyr	select(), group_by(), summarise()	Select, group by, and summarise data.	mtcars %>% select(mpg, gear) %>% group_by(gear) %>% summarise(mean_mpg = mean(mpg), median_mpg = median(mpg))
tidyr	pivot_wider(), pivot_longer()	Pivot data long or wide.	my_agg <- mtcars %>% select(mpg, gear, am) %>% group_by(gear, am) %>% summarise(mean_mpg = mean(mpg)) my_agg2 <- my_agg %>% pivot_wider(id_cols = gear, names_from = am, values_from = mean_mpg)
purrr	map(.x, .f)	Apply a function over a data's elements iteratively.	map(mtcars, mean)
knitr	kable(x)	Format a table.	my_table <- with(mtcars, table(gear, am)) kable(my_table)
stargazer	stargazer(x)	Format a regression.	my_ols <- lm(mpg ~ wt + hp + disp + gear + am, mtcars) stargazer(my_ols, type = 'text')

Chapter 14

Conclusion

I hope that these chapters were helpful in teaching you the concepts and syntax structure of R functions. This book is the first time I am writing something akin to a textbook: most of my writing have been academic papers, documentation for my packages,[1] and blog posts,[2] so I hope you have learned at least as much on R as I have on writing this book!

For further reading, I recommend reviewing the *References* and *Resources* sections, as they provide packages, data, and a book for practicing with and learning about R.

Thank you for reading!

[1] https://github.com/robertschnitman

[2] https://robertschnitman.netlify.app/

References

`dplyr`. https://dplyr.tidyverse.org/

`ggplot2`. https://ggplot2.tidyverse.org/

Hlavac, Marek (2018). stargazer: Well-Formatted Regression and Summary Statistics Tables. https://CRAN.R-project.org/package=stargazer

`Hmisc`. https://www.rdocumentation.org/packages/Hmisc/versions/4.3-1

`knitr`. https://yihui.org/knitr/

`purrr`. https://purrr.tidyverse.org/

RStudio Cheat Sheets. *strings.* Github. https://github.com/rstudio/cheatsheets/blob/master/strings.pdf

Schnitman, Robert. Github Profile. https://github.com/robertschnitman

—. Profile and Services. https://robertschnitman.netlify.com/

`stringr`. https://stringr.tidyverse.org/

`tidyr`. https://tidyr.tidyverse.org/

`tidyverse`. https://www.tidyverse.org/

Wikipedia. Small multiple. https://en.wikipedia.org/wiki/Small_multiple

Xie, Yihui, J. J. Allaire, & Garrett Grolemund (2019). R Markdown: The Definitive Guide. https://bookdown.org/yihui/rmarkdown/

Resources

1. UNdata (United Nations' statistical database)

UNdata provides international statistics hosted by the United Nations Statistics Division. It provides general regional profiles that summarize basic demographic, economic, and health data of countries, as well as time-series tables for historical analyses. Highly recommended for social scientists, public policy analysts, and other similar professions.

2. Institute for Digital Research and Education (idre), University of California at Los Angeles

The Institute has tutorial videos, annotated command outputs, workshop notes, and more for those wanting to learn and improve their skills in Stata, SPSS, SAS, and R. They emphasize applications while explaining the statistical theories behind them. Highly recommended as introductory material to these software.

3. R for Data Science (Hadley Wickham & Garrett Grolemund, 2017)

Wickham and Grolemund's R for Data Science book teaches a select number of indispensable tools for data preparation, visualization, and reporting. Particularly, they demonstrate the dplyr library for transformations, ggplot2 for professional graphics, and R Markdown for presentable documentation. A must-read for anyone working with the R programming language.

4. LibreOffice: The Document Foundation (free open-source equivalent to Microsoft Office)

Microsoft Office is ubiquitous. While its cost is a non-issue for large organizations, for others, however, even its cheapest options are expensive. Fortunately, LibreOffice offers suites that function the same, such as its Writer Document (Word equivalent) and Calc Spreadsheet (Excel equivalent). Notably, the Math Formula suite incorporates a formula editor that makes users be able to type complex mathematical equations at a faster rate than the cumbersome point-and-click method in Word. Additionally, LibreOffice's Access-equivalent Base boasts formal SQL scripting abilities and Wizard functions that guide the database design process. Recommended for students, work-at-home users, and smaller organizations wishing to cut costs.

5. bookdown.org

Bookdown.org is a site containing free online books about R. Notably, the *bookdown* book teaches you how to create your own books in R with the `bookdown` package (this book you're reading was created with this package!). Highly recommended for R users of any level, beginner through expert.

About the Author

Hello, I'm Robert Schnitman! I am an independent contractor providing statistical consulting and data analysis services to organizations and individuals. My services include preparing statistical reports, restructuring datasets, and creating visualizations. On the side, I blog primarily about R and data analysis on my website at https://robertschnitman.netlify.app/.

Feel free to contact me using the following links!

Email: robertschnitman@gmail.com

LinkedIn: https://www.linkedin.com/in/rschnitman/

Github: https://github.com/robertschnitman/