R Tutorial for MKT 4320

Marketing Analytics

Jeffrey Meyer, Ph.D., Bowling Green State University

2025

Contents

Preface

About this book

This book is a companion resource for **MKT 4320: Marketing Analytics** at Bowling Green State University.

Its purpose is to help you **use R effectively for applied marketing analytics**, not to provide a comprehensive or theoretical treatment of the R language. The focus is on learning how to run analyses, interpret results, and apply methods to real marketing problems.

This book is intentionally practical and course-driven. Chapters are designed to support lab assignments, projects, and examples used throughout the semester.

What this book is (and is not)

This book **is**:

- A guided introduction to R and RStudio for marketing analytics
- A reference for the types of analyses used in MKT 4320
- A practical companion to lab assignments and course projects

This book **is not**:

- A complete R programming textbook
- A reference for every possible R function or package
- A substitute for reading lab instructions carefully

You are not expected to memorize R syntax. You *are* expected to run code, modify examples, interpret output, and explain results.

The MKT4320BGSU package

Many examples in this book rely on the **MKT4320BGSU** R package.

This package contains:

- Custom functions used in the course
- Datasets for labs and demonstrations
- Tools designed specifically for teaching marketing analytics

Because the package is actively maintained, this book may be updated over time to reflect changes in functions, workflows, or best practices.

How to use this book

Most weeks, you will use this book alongside a lab assignment. A typical workflow looks like this:

1. Read the relevant sections of the book
2. Open the corresponding lab file
3. Run the example code
4. Modify the code to answer lab questions
5. Interpret results and write conclusions

Code examples are meant to be run. Errors are normal and are part of the learning process.

A note on updates

This book is a **living document**. Content may be revised, expanded, or clarified as the course progresses or as the supporting R package evolves.

When updates occur, they are intended to improve clarity, consistency, or alignment with course objectives.

Acknowledgments

Portions of this material are adapted from a variety of sources, including R documentation, textbooks, online tutorials, and teaching materials developed over time. Where appropriate, sources are noted within individual chapters.

What's next

The first chapter covers **Setup**, including how to install and configure R, RStudio, and the required packages for the course.

From there, the book moves into core R fundamentals and then into the specific analytical methods used in marketing analytics.

Chapter 1

Setup

1.1 What you need

To work through this book, you will need:

- Access to R/RStudio
- An internet connection (for installing packages)
- A basic familiarity with saving and opening files on your computer

1.2 R and RStudio (quick overview)

R is the language that runs your analysis. RStudio is the interface you use to write code, run code, view plots, and manage files. You have three options for using R/RStudio:

1. Use OSC Classroom On Demand
   Using the OSC Classroom on Demand is by far the easiest way to use R/RStudio for this course. All packages (see below for more info on packages) should already be loaded and tested to ensure compatability.
2. Install on your own machine
   This option involves two steps. First, install "base" R from the Comprehensive R Archive Network (CRAN): https://cran.r-project.org/. Second, install RStudio: https://posit.co/download/rstudio-desktop/#download.
3. Use a computer in an on-campus computer lab.
   This option should be considered a last resort. I cannot guarantee all packages will be able to be installed on university machines, and you would need to install the packages every time you use the machine.

1.3 Navigating RStudio

RStudio's interface is typically organized into four main panes in a default layout: the Source Editor, the R Console, the Environment/History, and the Files/Plots/Packages/Help/Viewer pane.

- **Source** (*usually* top-left)
  This pane is where you write, edit, and save your R scripts or R Markdown files. It functions like a text editor with features like syntax highlighting and code completion. Code is not executed immediately upon typing. You must explicitly send lines or sections of code to the Console (commonly using the "Run" button or keyboard shortcuts like Ctrl+Enter on Windows/Linux or Cmd+Enter on Mac) to execute them. This pane only appears when you have a file open.
- **Console** (*usually* bottom-left)
  This pane is where commands are actually run and text-based output, warnings, or error messages are displayed. You can type R commands directly into the console for immediate execution. The console shows a `>` prompt when it is ready to accept a new command. Commands typed directly here are not saved automatically, which is why writing in the Source editor is recommended for complete analyses.
- **Environments** (*usually* top-right)
  This pane helps manage your current R session and the objects within it. It may contain several tabs, including:
  - Environment Tab: Displays all the active objects (e.g., data frames, variables, functions, vectors) you've created or loaded during your current R session. You can inspect brief summaries of these objects here.
  - History Tab: Provides a log of all the commands that have been successfully executed in the console, which can be useful for reviewing past work.
  - Other Tabs: May also include tabs for Connections, Build, or Tutorials, depending on your RStudio configuration.
- **Output** (*usually* top-right)
  This multi-purpose pane provides access to various tools for project management and output viewing Files: A file browser for navigating your computer's directory structure and managing files within your current working directory. Plots: Where all the visualizations and graphs you create with R code will be displayed. It also may contain several tab, including:
  - Packages: Lists all installed R packages and allows you to install new ones or load them into your current session.
  - Help: The built-in documentation browser for R functions and packages. You can search for help directly here or by using a command like `?function_name` in the console.

  - Viewer: Used for displaying local web content, such as interactive HTML outputs, Shiny apps, or interactive plots generated by certain packages.
  - Plots: Use for displaying static plots generated by code.

The layout of these panes can be customized via the *Tools -> Global Options -> Pane Layout* menu. For more information on the panes in RStudio, please visit https://docs.posit.co/ide/user/ide/guide/ui/ui-panes.html.

1.4 Installing and loading packages

Packages extend R. You typically:

1. Install a package once per computer
2. Load the package each time you start a new R session

1.4.1 Install packages (one-time)

Packages are installed using the `install.packages("package_name")` function. For example, run the code below in the Console. If you see messages scrolling by, that is normal.

```
install.packages("tidyverse")
install.packages("devtools")
```

1.4.2 Load packages (each session)

Packages are loaded using the `library(package_name)` function. You may see messages and/or warnings scroll by when loading package, which is normal behavior.

```
library(tidyverse)
library(devtools)
```

1.5 Installing the MKT4320BGSU package

The MKT4320BGSU package contains the functions and datasets used throughout this course.

1.5.1 Install from GitHub

```
devtools::install_github("jdmeyer73/MKT4320BGSU")
```

1.5.2 Load the package

```
library(MKT4320BGSU)
```

1.5.3 Verify everything works

If the chunks below run without errors, you are set.

```
data("directmktg")
head(directmktg)
```

```
# A tibble: 6 x 5
    userid   age buy   gender salary
     <dbl> <dbl> <fct> <fct>   <dbl>
1 15624510    19 No    Male       19
2 15810944    35 No    Male       20
3 15668575    26 No    Female     43
4 15603246    27 No    Female     57
5 15804002    19 No    Male       76
6 15728773    27 No    Male       58
```

1.6 Getting help when you're stuck

These are the most useful help tools in R:

- `?function_name` opens help for a function
- `help.search("keyword")` searches help files
- If an error happens, read the last line carefully (it often tells you what to fix)

Examples:

```
?mean
help.search("logistic regression")
```

1.7 How you'll use this book

Most weeks, your workflow will look like this:

1. Read the relevant sections of the book
2. Open the lab file for the week
3. Run the example code
4. Modify the code to answer lab questions
5. Interpret results and write conclusions

Errors are normal—debugging is part of learning analytics.

1.8 What's next

In the next chapter, we'll cover basic R fundamentals you'll use constantly, including:

- objects and assignment
- vectors and data frames
- common functions
- importing data and basic data manipulation

Chapter 2

R Basics

This chapter introduces the core R concepts you will use throughout the course. The goal is not to be exhaustive, but to give you enough familiarity with R's basic building blocks so that later analytical methods make sense.

By the end of this chapter, you should be comfortable creating objects, working with vectors and data frames, indexing data, and performing simple data manipulation tasks.

2.1 Basics of R Commands

- R is case sensitive
- When using the console, use the keyboard ↑ and ↓ arrow keys to easily cycle through previous commands typed.
- When using the text editor (i.e., a script file) in the source pane, use the `Ctrl`+`Enter` (Windows/Linux) or `Cmd` + `Enter` (Mac) keyboard shortcut to submit a line of code directly to the console.
  - The entire line does **not** need to be highlighted; the cursor needs to be anywhere on the line to be submitted.
- When using the text editor/script file, the "#" symbol signifies a comment
  - Everything after is ignored
  - It can be on the same line:

```
x <- 100   # Assign 100 to x
```

- It can be on separate lines:

Table 2.1: R Operators

Description	Operator
Mathematical	
addition	$+$
subtraction	$-$
multiplication	$*$
division	$/$
exponentiation	^ or $**$
Logical	
less than	<
less than or equal to	<=
greater than	>
greater than or equal to	>=
exactly equal to	==
not equal to	!=
Not x	!x
x OR y	x|y
x AND y	x&y
test if X is TRUE	isTRUE(x)

```
# Assign 100 to x
x <- 100
```

2.2 Operators

Mathematical and logical operators are used frequently.

2.3 Objects and assignment

In R, almost everything you work with is an **object**. Objects store values, data, or results from functions.

You create objects using the assignment operator `<-`. In an RStudio script file or in the console, you can use a keyboard shortcut to produce the assignment operator. For Windows/Linux, `Alt` + `-`. For Mac, `Option` + `-`.

```
x <- 10
x
```

```
[1] 10
```

You can overwrite objects by assigning a new value:

```
x <- 25
x
```

```
[1] 25
```

As stated before, object names are case sensitive

```
x <- 100
X
```

```
Error:
! object 'X' not found
```

2.4 Vectors

A **vector** is a collection of values of the same type.

2.4.1 Creating vectors

Vectors are aften created using the concatenate function, `c(item1, item2, ...)`

```
ages <- c(18, 21, 25, 30)
ages
```

```
[1] 18 21 25 30
```

Common vector types include:

- Numeric
- Character
- Logical

```
names <- c("Alex", "Jamie", "Taylor", "Pat")
passed <- c(TRUE, FALSE, TRUE, TRUE)
```

The class of a vector can be checked with the `class(ojbect_name)` or `str(object_name)` function.

```
class(ages)
```

```
[1] "numeric"
```

```
str(names)
```

```
 chr [1:4] "Alex" "Jamie" "Taylor" "Pat"
```

```
class(passed)
```

```
[1] "logical"
```

Vectors can only hold a single class/type of value. When multiple classes are included, the values are coerced to the most general type.

```
mixed <- c(1, FALSE, 3.5, "Hello!")
mixed
```

```
[1] "1"      "FALSE"  "3.5"    "Hello!"
```

```
class(mixed)
```

```
[1] "character"
```

The `c()` function can be used to add to existing vectors, or combine vectors. Type coercion will be applied as needed.

```
ages2 <- c(ages, 29, 24)
ages
```

```
[1] 18 21 25 30
```

```
ages2
```

```
[1] 18 21 25 30 29 24
```

```
ages_names <- c(ages, names)
ages_names
```

```
[1] "18"     "21"     "25"     "30"     "Alex"   "Jamie"  "Taylor" "Pat"
```

```
class(ages_names)
```

```
[1] "character"
```

2.4.2 Vectorized operations

R is vectorized, meaning operations apply to all elements at once.

```
ages
```

```
[1] 18 21 25 30
```

```
ages + 1
```

```
[1] 19 22 26 31
```

```
ages * 2
```

```
[1] 36 42 50 60
```

2.4.3 Vector Length

The number of items in a vector can be checked with the `length(object_name)` function, but can also be seen using the `str(object_name)` function from earlier.

```
length(ages2)
```

```
[1] 6
```

```
str(ages2)
```

```
 num [1:6] 18 21 25 30 29 24
```

2.5 Data frames

A **data frame** is a table where:

- Each column is a variable
- Each row is an observation

Data frames are the most common way to handle data sets and to provide data to statistical functions.

Data frames can be created using the `data.frame(objects)` function:

```
# Creating a data frame all in one step
students <- data.frame(
  id = 1:4,
  age = c(18, 21, 25, 30),
  major = c("MKT", "FIN", "MKT", "MKT"))
students
```

```
  id age major
1  1  18   MKT
2  2  21   FIN
3  3  25   MKT
4  4  30   MKT
```

```
# Creating a data frame by combining vectors
new_students <- data.frame(c(names,ages,passed))
new_students
```

```
   c.names..ages..passed.
1                    Alex
2                   Jamie
3                  Taylor
4                     Pat
5                      18
6                      21
```

```
7                      25
8                      30
9                    TRUE
10                  FALSE
11                   TRUE
12                   TRUE
```

You will work with data frames constantly in this course.

2.6 Indexing and sequencing

Indexing is used to obtain particular elements of a data structure (vectors, matrices, data frames). Sequences are useful for indexing and loops.

2.6.1 Indexing vectors

Use square brackets `[]` to select elements.

```
ages[1]
```

```
[1] 18
```

```
ages[2:4]
```

```
[1] 21 25 30
```

Logical indexing is also common:

```
ages[ages > 21]
```

```
[1] 25 30
```

2.6.2 Sequencing

Use the `#:#` coding or the `seq(from = , to = , by = )` function to create a sequence.

```
1:10
```

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

```
seq(from = 0, to = 1, by = 0.2)
```

```
[1] 0.0 0.2 0.4 0.6 0.8 1.0
```

```
seq(0,100,10)
```

```
 [1]   0  10  20  30  40  50  60  70  80  90 100
```

2.7 Common functions

Functions take inputs (arguments) and return outputs.

Examples of commonly used functions:

```
mean(ages)
```

```
[1] 23.5
```

```
min(ages)
```

```
[1] 18
```

```
max(ages)
```

```
[1] 30
```

```
summary(ages)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  18.00   20.25   23.00   23.50   26.25   30.00 
```

To learn about a function:

```
?mean
```

2.8 Missing (and Other Interesting) Values

In R, missing values are assigned a special constant, NA.

NA is not a character value, but a type of its own. Any math performed on a value of NA becomes NA.

```
ages_missing <- c(ages, NA, NA)
ages_missing
```

```
[1] 18 21 25 30 NA NA
```

```
mean(ages_missing)
```

```
[1] NA
```

However, many commands contain a option, `na.rm=TRUE`, to ignore NA data when performing the function.

```
mean(ages_missing, na.rm=TRUE)
```

```
[1] 23.5
```

R also has special types for infinity, Inf, and undefined numbers (i.e., "not a number"), NaN. To see this in action, take the natural log, `log()`, of certain numbers. Notice that R provides a warning when the NaN is found.

```
log(-1)   # Not a number
```

```
Warning in log(-1): NaNs produced
```

```
[1] NaN
```

```
log(0)    # Infinity
```

```
[1] -Inf
```

2.9 Factors

Character data can be converted into nominal **factors** using the `as.factor(object_name)` function. Each unique character value will be a level of the factor. Behind the scenes, R stores the values as integers, with a separate list of labels. When the data type is set as a factor, R knows how to handle it appropriately in the model. The levels can be accessed with the `levels(object_name)` function.

```
school_year <- c("JR", "SR", "SR", "SO", "FR", "JR")
class(school_year)
```

```
[1] "character"
```

```
school_year <- as.factor(school_year)
str(school_year)
```

```
 Factor w/ 4 levels "FR","JR","SO",..: 2 4 4 3 1 2
```

```
levels(school_year)
```

```
[1] "FR" "JR" "SO" "SR"
```

2.10 What's next

In the next chapter, we focus on **using functions effectively in R**.

You will learn how to:

- pass arguments to functions,
- work with positional versus named arguments,
- understand default values, and
- read function documentation more efficiently.

These skills are essential for working with both built-in R functions and the custom functions provided in the **MKT4320BGSU** package.

Chapter 3

Functions in R

Most of what you do in R involves **using functions**. A function takes inputs (called **arguments**), performs an operation, and returns an output.

You have already used several functions, such as `mean()`, `summary()`, and `seq()`.

```
mean(ages)
```

```
[1] 23.5
```

```
summary(ages)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  18.00   20.25   23.00   23.50   26.25   30.00
```

3.1 Function arguments

Functions often require one or more arguments. For example:

```
mean(ages)
```

```
[1] 23.5
```

Here, `ages` is passed to the function `mean()` as its first argument.

Some functions require multiple arguments:

```r
seq(from = 0, to = 1, by = 0.2)
```

```
[1] 0.0 0.2 0.4 0.6 0.8 1.0
```

3.2 Positional vs named arguments

Arguments can be passed to a function in **two ways**.

3.2.1 Positional arguments

If arguments are supplied **in the correct order**, you do not need to name them.

```r
seq(0, 1, 0.2)
```

```
[1] 0.0 0.2 0.4 0.6 0.8 1.0
```

This works because R knows the expected order of arguments for `seq()`:

1. `from`
2. `to`
3. `by`

Positional arguments are concise, but they can make code harder to read and easier to misuse.

3.2.2 Named arguments (recommended)

You can explicitly name arguments using `argument = value`.

```r
seq(from = 0, to = 1, by = 0.2)
```

```
[1] 0.0 0.2 0.4 0.6 0.8 1.0
```

Advantages of named arguments:

- Code is easier to read
- Order does not matter
- Fewer mistakes when functions have many arguments

For example, this works even though the order is different:

```
seq(by = 0.2, to = 1, from = 0)
```

```
[1] 0.0 0.2 0.4 0.6 0.8 1.0
```

In this course, **naming arguments is recommended**, especially for complex functions.

3.3 Default argument values

Many function arguments have **default values**. If you do not specify them, R uses the default.

Example:

```
mean(ages)
```

```
[1] 23.5
```

The function `mean()` has optional arguments such as `na.rm`, which defaults to `FALSE`.

```
mean(ages, na.rm = TRUE)
```

```
[1] 23.5
```

You only need to specify arguments when:

- You want a non-default behavior
- The function requires the argument

3.4 Mixing positional and named arguments

You can mix positional and named arguments, but **positional arguments must come first**.

This is valid

```
mean(ages, na.rm = TRUE)
```

```
[1] 23.5
```

This will result in an error.

```
mean(na.rm = TRUE, ages)
```

```
[1] 23.5
```

A good rule of thumb:

- Use positional arguments for the **main input**
- Use named arguments for **options and settings**

3.5 Viewing function documentation

To understand how a function works and what arguments it accepts, use the help system.

```
?mean
```

The help page shows:

- What the function does
- Required and optional arguments
- Default values
- Examples

When in doubt, **read the argument list first**.

3.6 Common mistakes to avoid

- Forgetting parentheses. This refers to the function itself, not the result.:

```
mean
```

```
function (x, ...)
UseMethod("mean")
<bytecode: 0x0000029c291e1a90>
<environment: namespace:base>
```

- Misspelling argument names:

```
mean(ages, na_remove = TRUE)
```

```
[1] 23.5
```

- Assuming argument order without checking documentation

3.7 Key takeaway

You do **not** need to memorize every function or argument. Instead, focus on:

- Understanding what a function expects as input
- Knowing when to use named arguments
- Reading help files when unsure

These habits will make your R code more readable, more reliable, and easier to debug.

3.8 What's next

In the next chapter, we turn to **working with course data**.

You will learn how to:

- inspect and understand real datasets used in the course,
- distinguish between numeric and factor variables,
- identify missing or problematic values, and
- perform basic data manipulation using the `dplyr` package.

These skills will allow you to move from isolated examples to analyzing complete datasets and will serve as the foundation for descriptive analysis, visualization, and modeling later in the semester.

Chapter 4

Working with Data

This chapter focuses on working with real datasets used throughout the course. You will learn how course data are made available, how to inspect and understand variables, and how to perform basic data transformations using the `dplyr` package.

The goal is to help you move from small, isolated examples to working confidently with full datasets.

4.1 Course data vs importing data

In this course, many datasets are provided directly through the **MKT4320BGSU** package. These datasets are accessed using the `data()` function. For the next two chapters, we'll be using the `directmktg` dataset.

```
data("directmktg")
```

Using `data()` has several advantages:

- Everyone is working with the same dataset
- No file paths are required
- Fewer import-related errors

When possible, labs will rely on datasets loaded with `data()`.

4.1.1 Importing external data

In some cases, you may work with your own data files (e.g., CSV files).

```
mydata <- read.csv("mydata.csv")
```

Imported data behave the same way as course datasets once they are loaded into R. The key difference is how they enter your workspace.

As a general rule:

- Use `data()` for course-provided datasets
- Use `read.csv()` (or similar functions) for your own files

4.2 Inspecting datasets and variables

Before analyzing data, it is critical to understand what is in the dataset.

4.2.1 Viewing the data

The `head(data_object)` first few rows (the default is `n=6`) and helps you understand the structure of the data. There is a similar function, `tail()`, that looks at the last `n` rows.

```
head(directmktg)
```

```
    userid age buy gender salary
1 15624510  19  No   Male     19
2 15810944  35  No   Male     20
3 15668575  26  No Female     43
4 15603246  27  No Female     57
5 15804002  19  No   Male     76
6 15728773  27  No   Male     58
```

```
head(directmktg, n=5)
```

```
    userid age buy gender salary
1 15624510  19  No   Male     19
2 15810944  35  No   Male     20
3 15668575  26  No Female     43
4 15603246  27  No Female     57
5 15804002  19  No   Male     76
```

```
tail(directmktg, n=5)
```

```
      userid age buy gender salary
396 15691863  46 Yes Female     41
397 15706071  51 Yes   Male     23
398 15654296  50 Yes Female     20
399 15755018  36  No   Male     33
400 15594041  49 Yes Female     36
```

To see the entire data object, you can use the `View(data_object)` function, which will open the data in a spreadsheet-like format in the Source pane.

```
View(directmktg)
```

4.2.2 Dataset structure

The `str(object)` function is one of the most important commands in R. When used with a dataset object, it shows:

- Variable names
- Variable types (numeric, factor, character)
- A preview of values

```
str(directmktg)
```

```
'data.frame':   400 obs. of  5 variables:
 $ userid: num  15624510 15810944 15668575 15603246 15804002 ...
 $ age   : num  19 35 26 27 19 27 27 32 25 35 ...
 $ buy   : Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 2 1 1 ...
 $ gender: Factor w/ 2 levels "Male","Female": 1 1 2 2 1 1 2 2 1 2 ...
 $ salary: num  19 20 43 57 76 58 84 150 33 65 ...
```

4.2.3 Variable summaries

When used on the entire dataset, the `summary(object_name)` function provides a summary of all variables. The output depends on the variable type:

- Numeric variables: min, max, mean, quartiles
- Factor variables: counts by level

```
summary(directmktg)
```

```
     userid              age         buy        gender       salary      
 Min.   :15566689   Min.   :18.00   No :257   Male  :196   Min.   : 15.00  
 1st Qu.:15626764   1st Qu.:29.75   Yes:143   Female:204   1st Qu.: 43.00  
 Median :15694342   Median :37.00                          Median : 70.00  
 Mean   :15691540   Mean   :37.66                          Mean   : 69.74  
 3rd Qu.:15750363   3rd Qu.:46.00                          3rd Qu.: 88.00  
 Max.   :15815236   Max.   :60.00                          Max.   :150.00  
```

4.3 Data frames, variables, and indexing

A **data frame** is a table of rows and columns. In R, you will often need to reference a specific variable (column) or select a subset of rows/columns.

4.3.1 Accessing a variable (a column)

Use the $ operator to access a column by name.

```
directmktg$age
```

```
  [1] 19 35 26 27 19 27 27 32 25 35 26 26 20 32 18 29 47 45 46 48 45 47 48 45 46 47 49
 [39] 26 27 27 33 35 30 28 23 25 27 30 31 24 18 29 35 27 24 23 28 22 32 27 25 23 32 59
 [77] 18 22 28 26 30 39 20 35 30 31 24 28 26 35 22 30 26 29 29 35 35 28 35 28 27 28 32
[115] 42 40 35 36 40 41 36 37 40 35 41 39 42 26 30 26 31 33 30 21 28 23 20 30 28 19 19
[153] 31 36 40 31 46 29 26 32 32 25 37 35 33 18 22 35 29 29 21 34 26 34 34 23 35 25 24
[191] 24 19 29 19 28 34 30 20 26 35 35 49 39 41 58 47 55 52 40 46 48 52 59 35 47 60 49
[229] 40 42 35 39 40 49 38 46 40 37 46 53 42 38 50 56 41 51 35 57 41 35 44 37 48 37 50
[267] 40 37 47 40 43 59 60 39 57 57 38 49 52 50 59 35 37 52 48 37 37 48 41 37 39 49 55
[305] 40 42 51 47 36 38 42 39 38 49 39 39 54 35 45 36 52 53 41 48 48 41 41 42 36 47 38
[343] 38 47 47 41 53 54 39 38 38 37 42 37 36 60 54 41 40 42 43 53 47 42 42 59 58 46 38
[381] 42 48 44 49 57 56 49 39 47 48 48 47 45 60 39 46 51 50 36 49
```

A column is often a vector, which means you can use vector functions on it.

```
mean(directmktg$age)
```

```
[1] 37.655
```

```
summary(directmktg$age)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  18.00   29.75   37.00   37.66   46.00   60.00
```

4.3.2 Indexing a data frame

Data frames can be indexed (or subsetted) using:

- `data[rows, cols]`
- leave rows blank to keep all rows: `data[, cols]`
- leave cols blank to keep all columns: `data[rows, ]`

In addition, sequencing can be used to get multiple rows and/or columns. By default, subsetting a single column returns a vector. To retain it as a data frame, use the `drop = FALSE` argument.

```
directmktg[1, ]               # first row, all columns
directmktg[1:5, ]             # first 5 rows, all columns
directmktg[, 2]               # all rows, column 2 only (age), as a vector
directmktg[, 2, drop=FALSE]   # all rows, column 2 only (age), as a data frame
```

You can also access index columns using the column name columns. in two other ways. First, if you know the column number, you can using `[[ ]]` (useful when column names are stored in an object).

```
directmktg[1, "age"]                # first row, age column
directmktg[1:5, c("age", "gender")]  # first 5 rows, age and gender columns
```

4.3.3 Filtering rows using a condition (base R)

You can filter rows by using a logical condition inside the row index.

```
directmktg[directmktg$age >= 55, ]
```

```
      userid age buy gender salary
65  15605000  59  No Female     83
205 15660866  58 Yes Female    101
207 15654230  55 Yes Female    130
213 15707596  59  No Female     42
216 15779529  60 Yes Female    108
```

```
220 15732987  59 Yes   Male    143
224 15593715  60 Yes   Male    102
228 15685346  56 Yes   Male    133
244 15769596  56 Yes Female    104
248 15775590  57 Yes Female    122
259 15569641  58 Yes Female     95
263 15672821  55 Yes Female    125
272 15688172  59 Yes Female     76
273 15791373  60 Yes   Male     42
275 15692819  57 Yes Female     26
276 15727467  57 Yes   Male     74
281 15609669  59 Yes Female     88
293 15625395  55 Yes   Male     39
301 15736397  58 Yes Female     38
335 15814553  57 Yes   Male     60
337 15664907  58 Yes   Male    144
356 15606472  60 Yes   Male     34
366 15807525  59 Yes Female     29
367 15574372  58 Yes Female     47
371 15611430  60 Yes Female     46
372 15774744  60 Yes   Male     83
374 15708791  59 Yes   Male    130
380 15749381  58 Yes Female     23
385 15806901  57 Yes Female     33
386 15775335  56 Yes   Male     60
394 15635893  60 Yes   Male     42
```

This keeps only rows where `age >= 55`.

4.3.4 Specific variable summaries (base R)

You can use variable naming or indexing to get specific variable summaries using the `summary()` function.

```
summary(directmktg$age)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  18.00   29.75   37.00   37.66   46.00   60.00
```

```
summary(directmktg[,2:3])
```

```
      age          buy
 Min.   :18.00   No :257
 1st Qu.:29.75   Yes:143
 Median :37.00
 Mean   :37.66
 3rd Qu.:46.00
 Max.   :60.00
```

4.4 Data transformations (base R)

Data transformation refers to modifying or creating variables so they are more useful for analysis. This includes:

- creating new variables
- recoding variables

Base R can do all of these. Even if you prefer `dplyr` (covered later), it is helpful to understand what is happening "under the hood."

4.4.1 Creating a new variable

This example creates a **new variable** in the data frame using base R. The expression on the right-hand side, `directmktg$age / 10`, is evaluated first. Because `directmktg$age` is a vector, the division is applied to **each observation**, and the result is a new vector of the same length.

Using the assignment operator `<-`, this new vector is stored as a column named `age10` in the `directmktg` data frame.

```
directmktg$age10 <- directmktg$age / 10
head(directmktg, n=5)
```

```
    userid age buy gender salary age10
1 15624510  19  No   Male     19   1.9
2 15810944  35  No   Male     20   3.5
3 15668575  26  No Female     43   2.6
4 15603246  27  No Female     57   2.7
5 15804002  19  No   Male     76   1.9
```

4.4.2 Recoding with `ifelse()`

A common transformation is converting a categorical outcome into a numeric indicator.

```
directmktg$buy_binary <- ifelse(directmktg$buy == "Yes", 1, 0)
table(directmktg$buy, directmktg$buy_binary)
```

```
       0   1
  No  257   0
  Yes   0 143
```

Another common recoding is converting a numeric value into categories, which can be done with nested `ifelse()` statements.

```
directmktg$salary_cat <- ifelse(directmktg$salary <=50, "Low",
                                ifelse(directmktg$salary <=80, "Med", "High"))
head(directmktg[,c("salary","salary_cat")], 10)
```

```
   salary salary_cat
1      19        Low
2      20        Low
3      43        Low
4      57        Med
5      76        Med
6      58        Med
7      84       High
8     150       High
9      33        Low
10     65        Med
```

4.5 What's next

In the next chapter, we introduce the **`dplyr` package** for data manipulation. While the tasks performed with `dplyr` are similar to those you have already seen using base R, `dplyr` provides a more consistent and expressive way to work with data. These tools will be used extensively in later chapters for descriptive analysis, visualization, and modeling.

Chapter 5

dplyr Package

In this course, data transformations are primarily performed using the `dplyr` package (pronounced DEE ply er). This package makes data manipulation easier and more intuitive (for most). `dplyr` is built around the five main "verbs" shown below that make up a majority of data manipulation. However, there are other functions that dplyr uses to also help with data manipulation.

- `select` is used to subset columns
- `filter` is used to subset rows
- `mutate` is used to add new columns based on calculations (usually with other columns)
- `summarise` is use to perform summary calculations (e.g., mean, max, etc.) on data set
- `group_by` is used to group rows of a data frame with the same value in specified columns

In addition, `dplyr` uses the pipe, `%>%`, to string together a series of functions. Think of functions strung together as upstream and downstream functions. The function to the left of `%>%` is the upstream function, while the function to the right is the downstream function.

By default, the downstream function assumes the value coming from the upstream function is the first argument in its function. Therefore, the first argument can be omitted. If the downstream function needs to use the value from the upstream function assigned to a different argument, a `.` is simply put in the position of that argument

5.1 The *dplyr* workflow

The `dplyr` package is designed to make data manipulation clear and readable.

```
library(dplyr)
```

A typical `dplyr` workflow:

1. Start with a data frame
2. Apply a sequence of transformation verbs
3. Save or display the result

5.2 Selecting variables

Use `select()` to keep only the variables you need. This does not modify the original dataset unless you save the result.

```
directmktg %>%
  select(userid, age, gender) %>%
  head()
```

```
    userid age gender
1 15624510  19   Male
2 15810944  35   Male
3 15668575  26 Female
4 15603246  27 Female
5 15804002  19   Male
6 15728773  27   Male
```

"Negative" selection can also be done by using the `-` (minus sign) before a variable name or a vector of variable names.

```
directmktg %>%
  select(-gender) %>%
  head()
```

```
    userid age buy salary age10 buy_binary salary_cat
1 15624510  19  No     19   1.9          0        Low
2 15810944  35  No     20   3.5          0        Low
```

```
3 15668575  26  No     43   2.6          0        Low
4 15603246  27  No     57   2.7          0        Med
5 15804002  19  No     76   1.9          0        Med
6 15728773  27  No     58   2.7          0        Med
```

```
directmktg %>%
  select(-c(age,gender)) %>%
  head()
```

```
    userid buy salary age10 buy_binary salary_cat
1 15624510  No     19   1.9          0        Low
2 15810944  No     20   3.5          0        Low
3 15668575  No     43   2.6          0        Low
4 15603246  No     57   2.7          0        Med
5 15804002  No     76   1.9          0        Med
6 15728773  No     58   2.7          0        Med
```

5.3 Filtering observations

Use `filter()` to keep rows that meet certain conditions. (Note: the `nrow(object_name)` from base R provides the number of rows in the data frame).

```
nrow(directmktg)
```

```
[1] 400
```

```
directmktg %>%
  filter(age >= 35) %>%
  nrow()
```

```
[1] 254
```

Multiple conditions can be combined:

```
directmktg %>%
  filter(age >= 35,
         gender == "Male") %>%
  nrow()
```

```
[1] 124
```

5.4 Creating new variables

Use `mutate()` to create or transform variables. New variables are added to the data frame.

```
directmktg %>%
  select(userid, age) %>%
  mutate(age10 = age / 10) %>%
  head()
```

```
    userid age age10
1 15624510  19   1.9
2 15810944  35   3.5
3 15668575  26   2.6
4 15603246  27   2.7
5 15804002  19   1.9
6 15728773  27   2.7
```

5.5 Summaries with `summarise()`

The `summarise()` function is used to compute **summary statistics** from a data frame. It can be used **with or without grouping**.

When `summarise()` is used **without** `group_by()`, it computes summaries over the entire dataset.

```
directmktg %>%
  summarise(
    n = n(),
    mean_age = mean(age),
    buy_rate = mean(buy == "Yes")
  )
```

```
    n mean_age buy_rate
1 400   37.655   0.3575
```

Here's what this code is doing:

- `n()` counts the total number of observations in the dataset
- `mean(age)` computes the overall average age

- `mean(buy == "Yes")` computes the overall purchase rate

The result is a data frame with **one row**, where each column represents a summary statistic for the full dataset.

5.6 Grouped summaries with `group_by()` and `summarise()`

Often, you want to compute summaries **separately for different groups**, such as customer segments or demographic categories.

In `dplyr`, this is done by combining `group_by()` with `summarize()`.

```
directmktg %>%
  group_by(gender)
```

```
# A tibble: 400 x 8
# Groups:   gender [2]
     userid   age buy   gender salary age10 buy_binary salary_cat
      <dbl> <dbl> <fct> <fct>   <dbl> <dbl>      <dbl> <chr>
 1 15624510    19 No    Male       19   1.9          0 Low
 2 15810944    35 No    Male       20   3.5          0 Low
 3 15668575    26 No    Female     43   2.6          0 Low
 4 15603246    27 No    Female     57   2.7          0 Med
 5 15804002    19 No    Male       76   1.9          0 Med
 6 15728773    27 No    Male       58   2.7          0 Med
 7 15598044    27 No    Female     84   2.7          0 High
 8 15694829    32 Yes   Female    150   3.2          1 High
 9 15600575    25 No    Male       33   2.5          0 Low
10 15727311    35 No    Female     65   3.5          0 Med
# i 390 more rows
```

The `group_by()` function does not change the data values. Instead, it tells R how the data should be **temporarily divided into groups** for the next operation.

At this point, no calculations have been performed. Once the data are grouped, `summarise()` computes statistics **within each group**.

```
directmktg %>%
  group_by(gender) %>%
  summarise(
    n = n(),
    mean_age = mean(age),
    buy_rate = mean(buy == "Yes")
  )
```

```
# A tibble: 2 x 4
  gender     n mean_age buy_rate
  <fct>  <int>    <dbl>    <dbl>
1 Male     196     36.9    0.337
2 Female   204     38.4    0.377
```

Step by step:

- `group_by(gender)` splits the data into separate groups based on gender
- `n()` counts observations within each group
- `mean(age)` computes the average age within each group
- `mean(buy == "Yes")` computes the purchase rate within each group

The result is a data frame with **one row per group** and **one column per summary statistic**.

After `summarise()` runs, the grouping structure is automatically dropped, so the result behaves like a regular data frame.

5.7 Connecting `summarize()` to base R

Conceptually, grouped summaries in `dplyr` perform the same task as a multi-step process in base R:

1. Split the data into groups
2. Compute summary statistics for each group
3. Combine the results into a table

For example, in base R you might compute group means using functions such as `aggregate()` or by manually subsetting the data.

The advantage of `group_by()` and `summarise()` is that these steps are expressed **explicitly and readably**, making your data transformations easier to follow, debug, and modify.

5.8 Combining transformations

As you've seen, one of the main advantages of `dplyr` is that multiple steps can be chained together.

```
directmktg_clean <- directmktg %>%
  filter(age >= 35) %>%
  mutate(age10 = age / 10,
         buy_binary = ifelse(buy == "Yes", 1, 0)) %>%
  select(userid, age10, gender, buy_binary)

head(directmktg_clean)
```

```
    userid age10 gender buy_binary
1 15810944   3.5   Male          0
2 15727311   3.5 Female          0
3 15733883   4.7   Male          1
4 15617482   4.5   Male          1
5 15704583   4.6   Male          1
6 15621083   4.8 Female          1
```

This approach keeps data preparation transparent and reproducible.

5.9 Key takeaway

Before any modeling or visualization, you should be able to: - load data reliably, - inspect variable types and values, - identify missing or problematic data, and - transform data into a usable analytical form.

These steps are essential for sound marketing analytics.

5.10 What's next

In the next chapter, we will use cleaned and well-understood data to perform **descriptive analysis**, including frequency tables, crosstabs, measures of central tendency and dispersion, and correlation.

Chapter 6

Descriptive Analysis

Descriptive analysis summarizes and helps you understand your data **numerically**. In marketing analytics, it is often your first "sanity check" before modeling or visualization.

In this chapter you will learn how to:

- inspect a dataset and its variables,
- create frequency tables and crosstabs,
- compute measures of central tendency and dispersion,
- and compute and interpret correlations.

For this chapter, we'll be using the `airlinesat_small` dataset from the `MKT4320BGSU`.

```
data(airlinesat_small)
```

6.1 Descriptive Analysis

A quick descriptive workflow often looks like:

1) **Confirm the data structure** (rows/columns, variable types)
2) **Look for missing values and odd ranges**
3) **Summarize key variables** (categorical → counts; numeric → mean/SD, etc.)
4) **Compare segments** (e.g., by treatment, gender, region, etc.)

6.1.1 Inspecting the dataset

The following functions are useful for inspecting the dataset.

- `str()` shows each variables' type, displays factor levels if present, and gives a compact preview of values for each variable
- `dim()` returns the number of rows and columsn to quickly tell you sample size and number of variables
- `names()` provides all column names, which helps with selecting variables for writing code or formulas
- `head()` gives the first six rows of the dataset to allow for visual inspection of raw values

```
# Take a look at the data frame
str(airlinesat_small)
```

```
'data.frame':   1065 obs. of  13 variables:
 $ age           : num  30 55 56 43 44 40 39 41 33 51 ...
 $ country       : Factor w/ 5 levels "at","ch","de",..: 2 2 2 4 2 2 2 2 2 3 ...
 $ flight_class  : Factor w/ 3 levels "Business","Economy",..: 2 1 2 2 1 3 2 1 2 1 ...
 $ flight_latest : Factor w/ 6 levels "within the last 12 months",..: 4 3 5 3 6 5 6 3 3
 $ flight_purpose: Factor w/ 2 levels "Business","Leisure": 2 1 1 2 1 2 1 1 2 1 ...
 $ flight_type   : Factor w/ 2 levels "Domestic","International": 1 2 1 1 2 2 1 2 1 2
 $ gender        : Factor w/ 2 levels "female","male": 2 2 1 1 1 2 2 2 2 2 ...
 $ language      : Factor w/ 3 levels "English","French",..: 2 1 1 2 1 3 2 2 2 3 ...
 $ nflights      : num  2 6 8 7 25 16 35 9 3 4 ...
 $ status        : Factor w/ 3 levels "Blue","Gold",..: 1 2 1 1 2 2 1 2 1 2 ...
 $ nps           : num  6 10 8 8 6 7 8 7 8 8 ...
 $ overall_sat   : num  2 6 2 4 2 4 4 4 4 3 ...
 $ reputation    : num  3 6 4 6 5 3 3 4 2 4 ...
```

```
# Dimensions (rows, columns)
dim(airlinesat_small)
```

```
[1] 1065   13
```

```
# Variable names
names(airlinesat_small)
```

```
 [1] "age"           "country"       "flight_class"   "flight_latest"  "flight_purpos
 [7] "gender"        "language"      "nflights"       "status"         "nps"
[13] "reputation"
```

```
# First few rows
head(airlinesat_small)
```

```
  age country flight_class            flight_latest flight_purpose   flight_type gender language
1  30      ch      Economy within the last 6 months        Leisure      Domestic   male   French
2  55      ch     Business within the last 3 months       Business International   male  English
3  56      ch      Economy    within the last month       Business      Domestic female  English
4  43      fr      Economy within the last 3 months        Leisure      Domestic female   French
5  44      ch     Business     within the last week       Business International female  English
6  40      ch        First    within the last month        Leisure International   male   German
  overall_sat reputation
1           2          3
2           6          6
3           2          4
4           4          6
5           2          5
6           4          3
```

6.1.2 Missing values (quick checks)

It is also important to check for missing values, because they can have an adverse effect on some analyses. The `is.na()` function is often used for this, which returns a `TRUE` if a value is `NA`. Ultimately, this will tell which values in the data are missing to help decide how to handle them.

```
# Missing values by variable
colSums(is.na(airlinesat_small))
```

```
          age       country  flight_class  flight_latest flight_purpose   flight_type
            0             0             0              0              0             0
     nflights        status           nps    overall_sat     reputation
            0             0             0              0              0
```

```
# Total missing values in the dataset
sum(is.na(airlinesat_small))
```

```
[1] 0
```

6.2 Frequency Tables

Frequency tables summarize **categorical** variables (and sometimes binned numeric variables).

6.2.1 One-way frequency table

To create a frequency table, use the `table()` function in R. Counts are often converted into proportions or rates to make results easier to interpret and compare.

In R, the `proportions()` function is used to convert frequency tables into proportions. Multiply the table by 100 to get percentages.

```r
# Frequency table for a categorical variable
table(airlinesat_small$gender)
```

```
female   male
   280    785
```

```r
# Add proportions (shares)
proportions(table(airlinesat_small$gender))
```

```
   female      male
0.2629108 0.7370892
```

```r
# Add percentages
100 * proportions(table(airlinesat_small$gender))
```

```
  female     male
26.29108 73.70892
```

6.3 Crosstabs

A crosstab is a frequency table for **two categorical variables**. Crosstabs are used frequently in marketing to compare segments (e.g., purchase by gender).

6.3.1 Base R

Base R does not do a great job of easily creating crosstabs and testing for independence of the two variables. Instead, a multistep process is required:

- Create the two-way frequency table using the `table(rowvar, colvar)` function and assign it to a separate object
- Display the two-way freq table by just using the table name

```
# Counts
ct <- table(airlinesat_small$flight_class, airlinesat_small$gender)
ct
```

```
          female male
  Business     39  146
  Economy     239  626
  First         2   13
```

- Use the function `proportions(tablename, margin)` on the newly created object to get column, row, or total percentages
  - `proportions(tablename)` gives total percentages
  - `proportions(tablename, 1)` gives row percentages
  - `proportions(tablename, 2)` gives column percentages

```
proportions(ct)             # Total proportions: what pprportion of all respondents
```

```
                female        male
  Business 0.036619718 0.137089202
  Economy  0.224413146 0.587793427
  First    0.001877934 0.012206573
```

```
proportions(ct, margin=1)   # Row proportions: how are genders distributed within a row
```

```
             female      male
  Business 0.2108108 0.7891892
  Economy  0.2763006 0.7236994
  First    0.1333333 0.8666667
```

```
proportions(ct, margin=2)   # Column proportions: how is th outcome distributed within e
```

```
                female        male
Business 0.139285714 0.185987261
Economy  0.853571429 0.797452229
First    0.007142857 0.016560510
```

- Use the function `chisq.test(tablename)` on the newly created object to run the test of independence

```
chisq.test(ct)
```

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

```
    Pearson's Chi-squared test

data:  ct
X-squared = 4.6912, df = 2, p-value = 0.09579
```

6.3.2 Using package *sjPlot*

The **sjPlot** package can print crosstabs with nicer formatting. Use the function `tab_xtab(var.row=, var.col=, show.col.prc=TRUE)` to get a standard crosstab with column percentages and the chi-square test of independence.

```
tab_xtab(airlinesat_small$flight_class,
         airlinesat_small$gender,
         show.col.prc = TRUE)
```

flight_class

gender

Total

female

male

Business

3913.9 %

14618.6 %

18517.4 %

Economy

23985.4 %

62679.7 %

86581.2 %

First

20.7 %

131.7 %

151.4 %

Total

280100 %

785100 %

1065100 %

2=4.691 · df=2 · Cramer's V=0.066 · Fisher's p=0.095

6.4 Measures of Central Tendency and Dispersion

For **numeric** variables, the most common summaries are:

- Central tendency: mean, median, mode
- Dispersion: variance, standard deviation, IQR, range

6.4.1 Base R

Any individual summary statistic can be easily calculated using Base R with functions such as:

- `mean(var)` for mean
- `sd(var)` for standard deviation
- `quantile(var, .percentile)` for percentiles (e.g., '.50' would be median)

For summary statistics except for standard deviation, the `summary(object)` function can be used, where object can be a single variable or an entire data frame

```
# Base R summary (min, quartiles, median, mean, max)
summary(airlinesat_small$age)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  19.00   42.00   50.00   50.42   58.00  101.00
```

```
summary(airlinesat_small)
```

```
      age          country    flight_class                    flight_latest  flight_purpo
 Min.   : 19.00   at:108   Business:185   within the last 12 months:139   Business:525
 1st Qu.: 42.00   ch: 66   Economy :865   within the last 2 days   : 57   Leisure :540
 Median : 50.00   de:695   First   : 15   within the last 3 months :296
 Mean   : 50.42   fr:  1                  within the last 6 months :187
 3rd Qu.: 58.00   us:195                  within the last month    :253
 Max.   :101.00                           within the last week     :133
    gender       language      nflights         status         nps          overall_sat
 female:280   English:233   Min.   :  1.00   Blue  :677   Min.   : 1.00   Min.   :1.00
 male  :785   French : 10   1st Qu.:  4.00   Gold  :143   1st Qu.: 6.00   1st Qu.:3.00
              German :822   Median :  8.00   Silver:245   Median : 8.00   Median :4.00
                            Mean   : 13.42                Mean   : 7.52   Mean   :3.74
                            3rd Qu.: 16.00                3rd Qu.: 9.00   3rd Qu.:5.00
                            Max.   :457.00                Max.   :11.00   Max.   :7.00
```

```
# Mean and SD (remove missing values if present)
mean(airlinesat_small$age, na.rm = TRUE)
```

```
[1] 50.41972
```

```
sd(airlinesat_small$age, na.rm = TRUE)
```

```
[1] 12.27464
```

If you want a single, custom summary:

```
x <- airlinesat_small$age

c(n = sum(!is.na(x)),
  mean = mean(x, na.rm = TRUE),
  sd = sd(x, na.rm = TRUE),
  median = median(x, na.rm = TRUE),
  iqr = IQR(x, na.rm = TRUE),
  min = min(x, na.rm = TRUE),
  max = max(x, na.rm = TRUE))
```

```
         n       mean         sd     median        iqr        min        max
1065.00000   50.41972   12.27464   50.00000   16.00000   19.00000  101.00000
```

6.4.2 Using package *dplyr*

With **dplyr**, you can summarize many variables and/or do summaries by groups.

Overall summaries:

```
airlinesat_small %>%
  summarise(n = n(),
            mean_age = mean(age, na.rm = TRUE),
            sd_age = sd(age, na.rm = TRUE),
            median_age = median(age, na.rm = TRUE),
            iqr_age = IQR(age, na.rm = TRUE),
            mean_nflights = mean(nflights, na.rm = TRUE),
            sd_nflights = sd(nflights, na.rm = TRUE))
```

```
     n mean_age   sd_age median_age iqr_age mean_nflights sd_nflights
1 1065 50.41972 12.27464         50      16      13.41878    20.22647
```

Group summaries (example: by `buy`):

```
airlinesat_small %>%
  group_by(status) %>%
  summarise(n = n(),
            mean_age = mean(age, na.rm = TRUE),
            sd_age = sd(age, na.rm = TRUE),
            mean_nflights = mean(nflights, na.rm = TRUE),
            sd_nflights = sd(nflights, na.rm = TRUE))
```

```
# A tibble: 3 x 6
  status     n mean_age sd_age mean_nflights sd_nflights
  <fct>  <int>    <dbl>  <dbl>         <dbl>       <dbl>
1 Blue     677     50.6   13.3          8.23        19.2
2 Gold     143     53     10.0         24.6         20.9
3 Silver   245     48.3   10.0         21.2         17.2
```

6.4.3 Using package *vtable*

The **vtable** package can generate clean, compact descriptive tables with the `sumtable(data, vars=c(""))` function. For factor variables, it will provide the counts and percents of each level.

```
sumtable(airlinesat_small, c("age", "nflights", "status"),
         add.median=TRUE,    # 'add.median=TRUE' includes a 50th percentile column
         title=NA)
```

Variable	N	Mean	Std. Dev.	Min	Pctl. 25	Pctl. 50	Pctl. 75	Max
age	1065	50	12	19	42	50	58	101
nflights	1065	13	20	1	4	8	16	457
status	1065							
... Blue	677	64%						
... Gold	143	13%						
... Silver	245	23%						

The `sumtable()` function can also provide summary statistics by a grouping variable.

```
sumtable(airlinesat_small, c("age", "nflights", "status"),
         add.median=TRUE,
         group="gender",
         title=NA)
```

gender	female				male			
Variable	N	Mean	SD	Median	N	Mean	SD	Median
age	280	51	13	51	785	50	12	50
nflights	280	9.3	12	5	785	15	22	9
status	280				785			
... Blue	225	80%			452	58%		
... Gold	16	6%			127	16%		
... Silver	39	14%			206	26%		

6.5 Correlation

Correlation measures the strength of a **linear relationship** between two numeric variables. The most common is Pearson correlation (the default).

6.5.1 Base R

Base R can easily provide a correlation matrix of many variables, and it can provide a correlation test between two variables at a time, but it cannot produce a correlation matrix with p-values.

To get a correlation matrix, use the `cor()` function with a dataframe of the variables desired or using indexing on variable names. By default, it uses all observations, which can create `NA` values if any missing values exists. Therefore, the preference is to add the option `use = "pairwise.complete.obs"` to only calculate the correlation on non-missing values for each pair of variables.

```
mycorr_df <- airlinesat_small %>%
  select(age, nflights, nps, overall_sat, reputation)
cor(mycorr_df, use = "pairwise.complete.obs")
```

```
                  age    nflights         nps overall_sat  reputation
age          1.00000000 -0.11576301  0.09867319  0.05903446  0.06082991
nflights    -0.11576301  1.00000000 -0.08949782 -0.05366975 -0.06364290
nps          0.09867319 -0.08949782  1.00000000  0.29961310  0.50712230
overall_sat  0.05903446 -0.05366975  0.29961310  1.00000000  0.17748688
reputation   0.06082991 -0.06364290  0.50712230  0.17748688  1.00000000
```

```
cor(airlinesat_small[,c("age", "nflights", "nps", "overall_sat", "reputation")],
    use = "pairwise.complete.obs")
```

```
                  age    nflights         nps overall_sat  reputation
age          1.00000000 -0.11576301  0.09867319  0.05903446  0.06082991
nflights    -0.11576301  1.00000000 -0.08949782 -0.05366975 -0.06364290
nps          0.09867319 -0.08949782  1.00000000  0.29961310  0.50712230
overall_sat  0.05903446 -0.05366975  0.29961310  1.00000000  0.17748688
reputation   0.06082991 -0.06364290  0.50712230  0.17748688  1.00000000
```

To get the correlation test for any one pair of variables, use the `cor.test(var1, var2)` function. By default, it includes only observations that are non-missing in both variables.

```
cor.test(airlinesat_small$age, airlinesat_small$nflights)
```

```
    Pearson's product-moment correlation

data:  airlinesat_small$age and airlinesat_small$nflights
t = -3.7998, df = 1063, p-value = 0.000153
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.17461941 -0.05608231
sample estimates:
      cor
-0.115763
```

6.5.2 Using package *Hmisc*

The **Hmisc** package is useful for correlation matrices with p-values using the `rcorr()` function. This function expects a matrix (or data frame coerced to matrix).

```
rcorr(as.matrix(mycorr_df))
```

```
             age nflights   nps overall_sat reputation
age          1.00    -0.12  0.10        0.06       0.06
nflights    -0.12     1.00 -0.09       -0.05      -0.06
nps          0.10    -0.09  1.00        0.30       0.51
overall_sat  0.06    -0.05  0.30        1.00       0.18
reputation   0.06    -0.06  0.51        0.18       1.00

n= 1065


P
            age    nflights nps    overall_sat reputation
age                0.0002   0.0013 0.0541      0.0472
nflights    0.0002          0.0035 0.0800      0.0378
nps         0.0013 0.0035          0.0000      0.0000
overall_sat 0.0541 0.0800   0.0000             0.0000
reputation  0.0472 0.0378   0.0000 0.0000
```

6.5.3 Using package *sjPlot*

The **sjPlot** package can produce formatted correlation tables for reports using the `tab_corr()` function. By default, it uses listwise (or casewise) deletion, which removes an entire case (or row) if any value is `NA`, which may result in major data loss. Use the option `na.deletion = "pairwise")` to prevent this.

```
tab_corr(mycorr_df,
         triangle = "lower",
         show.p = TRUE,
         na.deletion = "pairwise")
```

age

nflights

nps

overall_sat

reputation

age

nflights

-0.116***

nps

0.099**

-0.089**

overall_sat

0.059

-0.054

0.300***

reputation

0.061*

-0.064*

0.507***

0.177***

Computed correlation used pearson-method with pairwise-deletion.

6.6 Why descriptive analysis matters

Descriptive statistics help you to:

- detect unusual values,
- understand typical behavior,
- compare groups,
- and check whether results are plausible.

They also guide decisions about which models or visualizations are appropriate.

6.7 What's next

In the next chapter, we move from **numbers** to **visuals**. You will learn how to create effective data visualizations using a small amount of Base R, but primarily `ggplot2`. Visualizations allow you to see patterns, distributions, and relationships that are often difficult to detect from tables alone.

Together, descriptive statistics and visualization form the foundation of **exploratory data analysis**, which prepares you for modeling and inference later in the course.

Chapter 7

Data Visualization

Descriptive statistics summarize data numerically, but visualizations often reveal patterns, trends, and anomalies more quickly. This chapter introduces basic data visualization techniques in R. We begin with Base R graphics to understand core plotting concepts, then move to ggplot2, which provides a more flexible and powerful system for creating graphics.

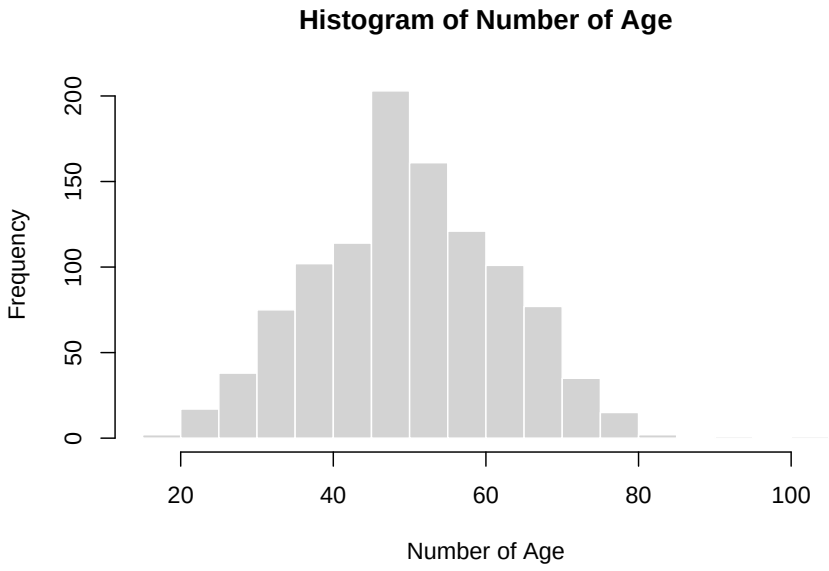
7.1 Base R Visualizations

Base R graphics are built into R and require no additional packages. They are useful for quick exploratory analysis and for understanding how plotting works at a fundamental level.

7.1.1 Histogram (Base R)

A histogram shows the distribution of a numeric variable. Histograms are useful for assessing the shape, spread, and potential outliers in a numeric variable.

```
hist(airlinesat_small$age,
     main = "Histogram of Number of Age",
     xlab = "Number of Age",
     col = "lightgray",
     border = "white")
```



7.1.2 Box-and-Whisker Plot (Base R)

A box-and-whisker plot (often called a boxplot) summarizes the distribution of a numeric variable using five key values:

- Minimum
- First quartile (25th percentile)
- Median
- Third quartile (75th percentile)
- Maximum

Boxplots are especially useful for:

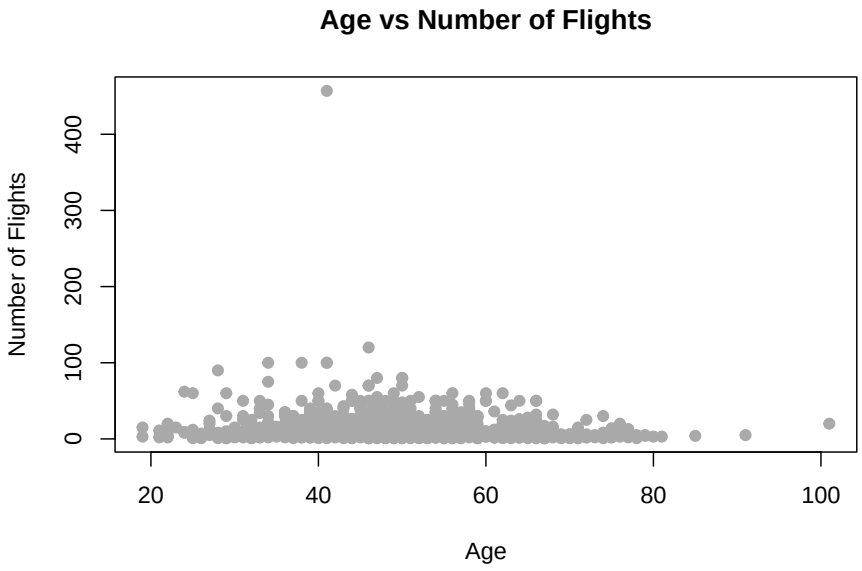
- Comparing distributions across groups
- Identifying skewness
- Detecting potential outliers

```
boxplot(airlinesat_small$age,
        main = "Distribution of Age",
        ylab = "Age",
        col  = "lightgray")
```

7.1.3 Scatterplot (Base R)

A scatterplot displays the relationship between two numeric variables. Scatterplots are commonly used to detect relationships, nonlinear patterns, and outliers.

```
plot(airlinesat_small$age,
     airlinesat_small$nflights,
     main = "Age vs Number of Flights",
     xlab = "Age",
     ylab = "Number of Flights",
     pch = 19,
     col = "darkgray")
```



7.1.4 Line Chart (Base R)

Line charts are typically used for ordered data, such as time series or values summarized across an ordered variable. Line charts emphasize change across an ordered dimension rather than individual observations.

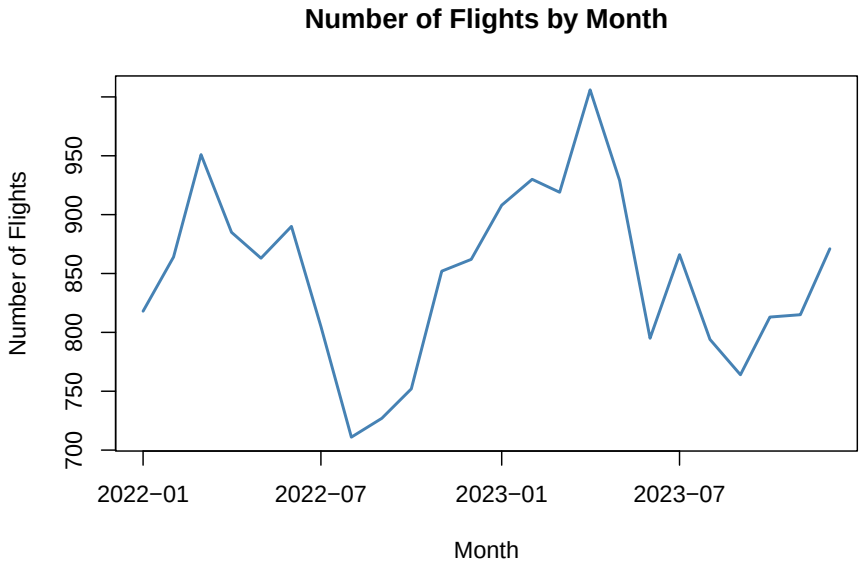
First, we'll simulate some time series data, and then we'll create the line chart.

```
# Simulate monthly flight data
set.seed(123)
months <- seq(from = as.Date("2022-01-01"), to = as.Date("2023-12-01"), by = "month")
n_months <- length(months)
flights <- round(800 +
    seq(0, 100, length.out = n_months) +          # upward trend
    80 * sin(2 * pi * (1:n_months) / 12) +         # seasonality
    rnorm(n_months, mean = 0, sd = 40))              # random noise

flight_ts <- data.frame(month = months, flights = flights)

# Line chart of flights by month
plot(flight_ts$month,
     flight_ts$flights,
     type = "l",
     main = "Number of Flights by Month",
```

```
     xlab = "Month",
     ylab = "Number of Flights",
     col  = "steelblue",
     lwd  = 2)
```

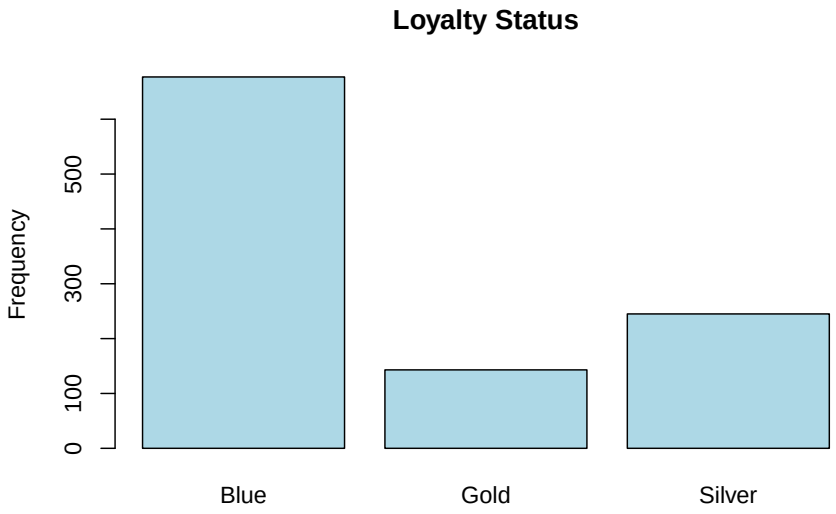


7.1.5 Bar Chart (Base R)

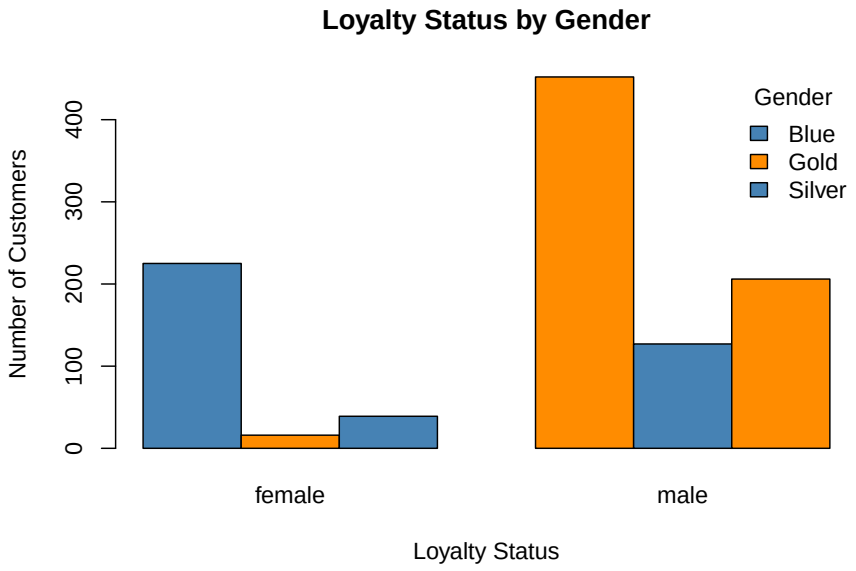
Bar charts are used for categorical variables. They are useful for comparing counts or proportions across categories. They can also show results for different categories of another variable (i.e., a side-by-side bar chart).

```
status_counts <- table(airlinesat_small$status)

barplot(
  status_counts,
  main = "Loyalty Status",
  ylab = "Frequency",
  col = "lightblue"
)
```




```r
status_gender_tab <- table(airlinesat_small$status, airlinesat_small$gender)
barplot(status_gender_tab,
        beside = TRUE,
        col = c("steelblue", "darkorange"),
        main = "Loyalty Status by Gender",
        xlab = "Loyalty Status",
        ylab = "Number of Customers",
        legend.text = TRUE,
        args.legend = list(title = "Gender", x = "topright", bty = "n"))
```



7.2 Moving Beyond Base R

While Base R graphics are useful, they can become cumbersome when creating more complex plots or when consistent styling is needed. The `ggplot2` package provides a structured approach to visualization based on the **g**rammar of **g**raphics.

7.3 Introduction to *ggplot2*

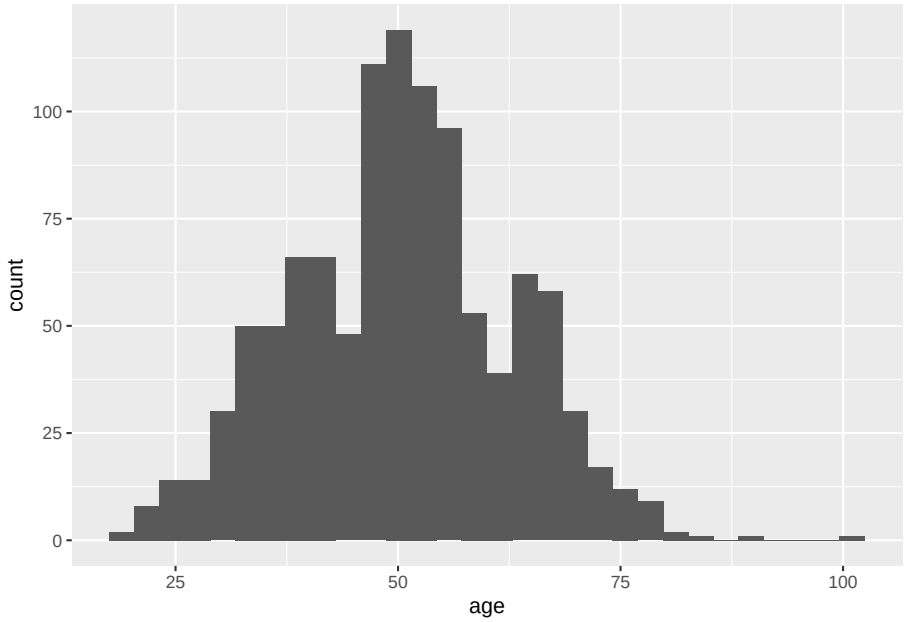
In `ggplot2`, plots are built in layers from three main components: data, aesthetic mappings (i.e., a coordinate system identifying `x` and `y` variables), and geometric objects (i.e., how the data should be displayed). In addition, the plot can be enhanced by adding additional layers using the `+` operator. `ggplot2` also works very well with `dplyr` when data manipulation is needed prior to creating the plot.

7.3.1 Histogram

Histograms in `ggplot` use the `geom_histogram()` layer. As with many geoms in `ggplot2`, no options are *required* in the geom. Here is a basic histogram.

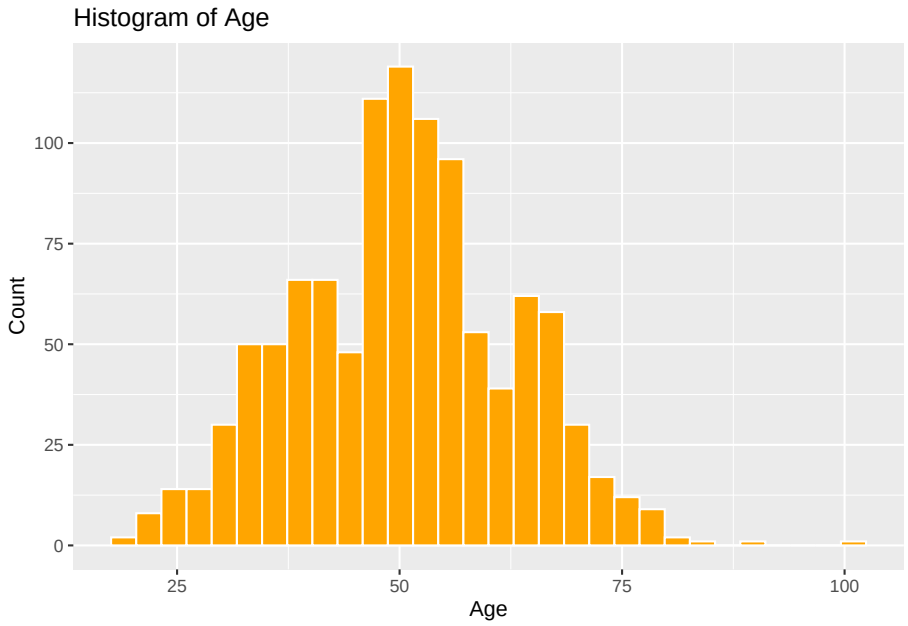
```
ggplot(airlinesat_small, aes(x = age)) +
  geom_histogram()
```

```
`stat_bin()` using `bins = 30`. Pick better value `binwidth`.
```



One of the benefits of `ggplot2` is the ease of making the plot look more visually appealing often more informative. Here is the histogram with additional options for `bins`, the `fill` color, and the outline `color`, along with labels using the `labs` layer.

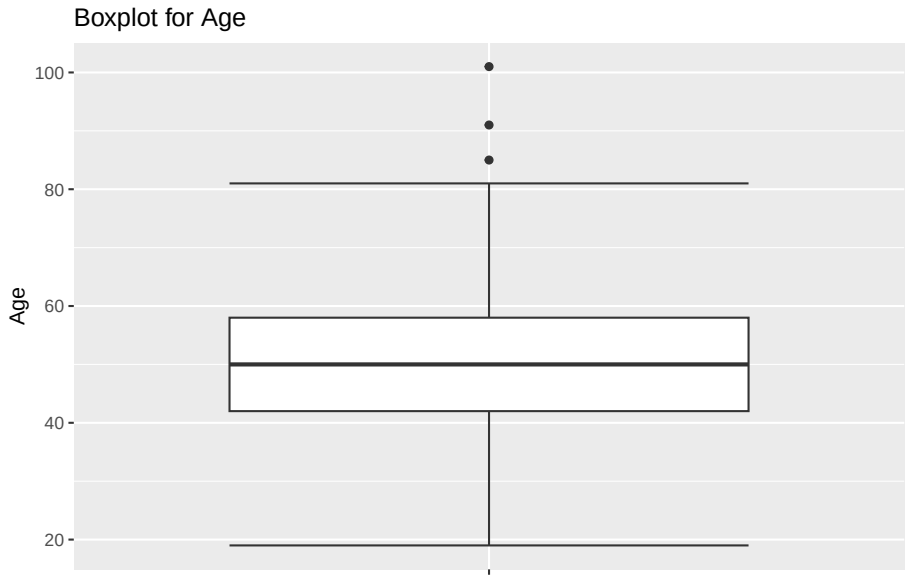
```
ggplot(airlinesat_small, aes(x = age)) +
  geom_histogram(bins = 30,
                 fill = "orange",
                 color = "white") +
  labs(title = "Histogram of Age",
       x = "Age",
       y= "Count")
```



7.3.2 Box-and-Whiskers Plot

Box Plots are drawn with the `geom_boxplot()` geom, which by default creates a box plot for a continuous `y` variable, but for each level of a discrete `x` variable. In addition, the standard box plot does not contain "whiskers". To get a box plot for only the continuous `y` variable, use `x = ""` as the discrete `x` variable. To add whiskers, include a `staplewidth = 1` within the `geom_boxplot()`.

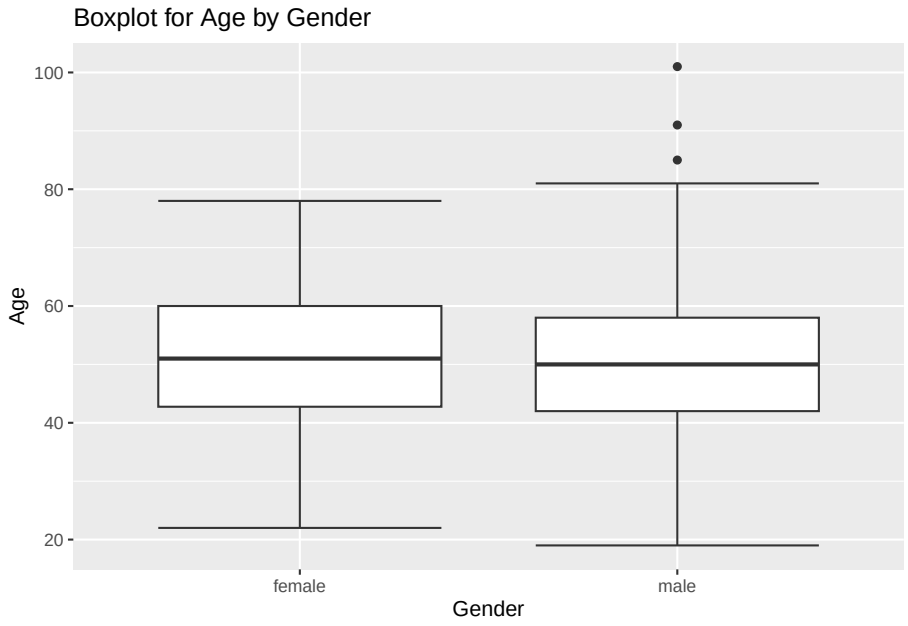
```
ggplot(airlinesat_small, aes(x = "", y=age)) +
  geom_boxplot(staplewidth = 1) +
  labs(title="Boxplot for Age",
       x = "",
       y = "Age")
```



7.3.2.1 Side-by-Side Boxplot

To create side-by-side boxplots for a discrete variable, simply repace the `x = ""` with a variable name.

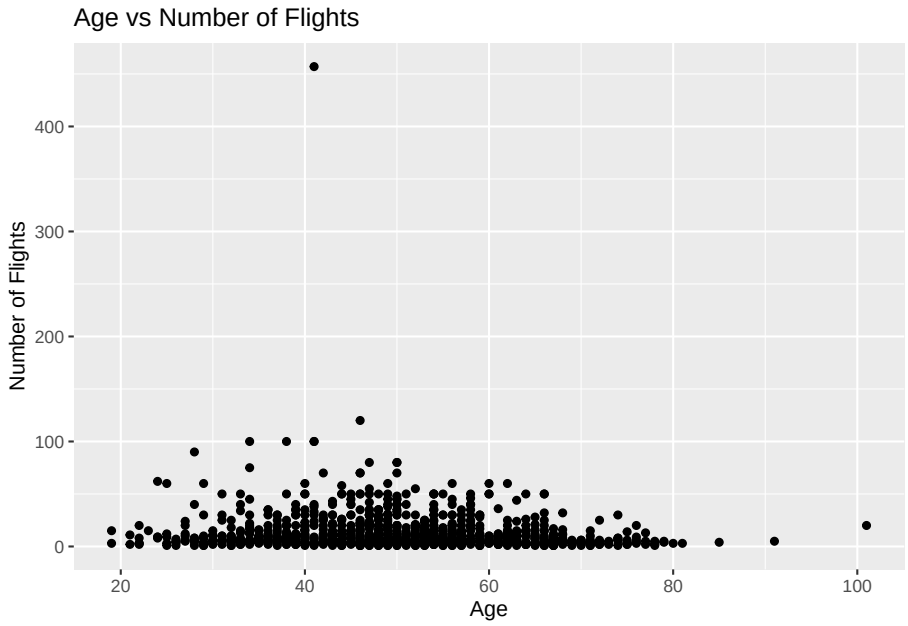
```
ggplot(airlinesat_small, aes(x = gender, y=age)) +
  geom_boxplot(staplewidth = 1) +
  labs(title="Boxplot for Age by Gender",
       x = "Gender",
       y = "Age")
```



7.3.3 Scatterplot

Scatterplots are drawn with the `geom_point()` geom and are used to show the relationship between two continuous variables.

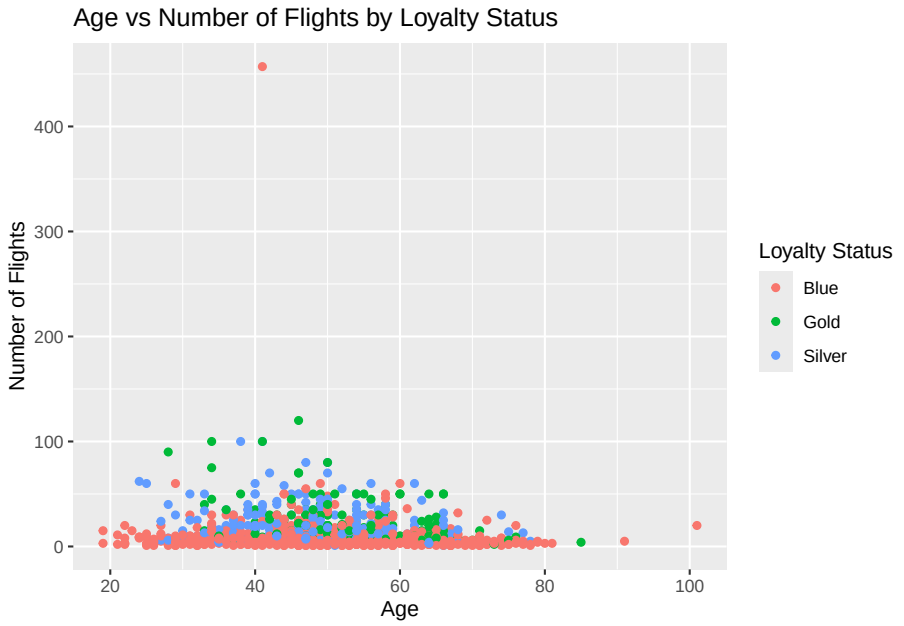
```
ggplot(airlinesat_small, aes(x = age, y = nflights)) +
  geom_point() +
  labs(title = "Age vs Number of Flights",
       x = "Age",
       y = "Number of Flights")
```



7.3.3.1 Scatterplot with a Categorical Variable

More interesting scatterplots can be created by changing the color of the points by a third, discrete variable. This means adding a new aesthetic in the `aes()` part.

```
ggplot(airlinesat_small, aes(x = age, y = nflights, color=status)) +
  geom_point() +
  labs(title = "Age vs Number of Flights by Loyalty Status",
       x = "Age",
       y = "Number of Flights",
       color = "Loyalty Status")
```

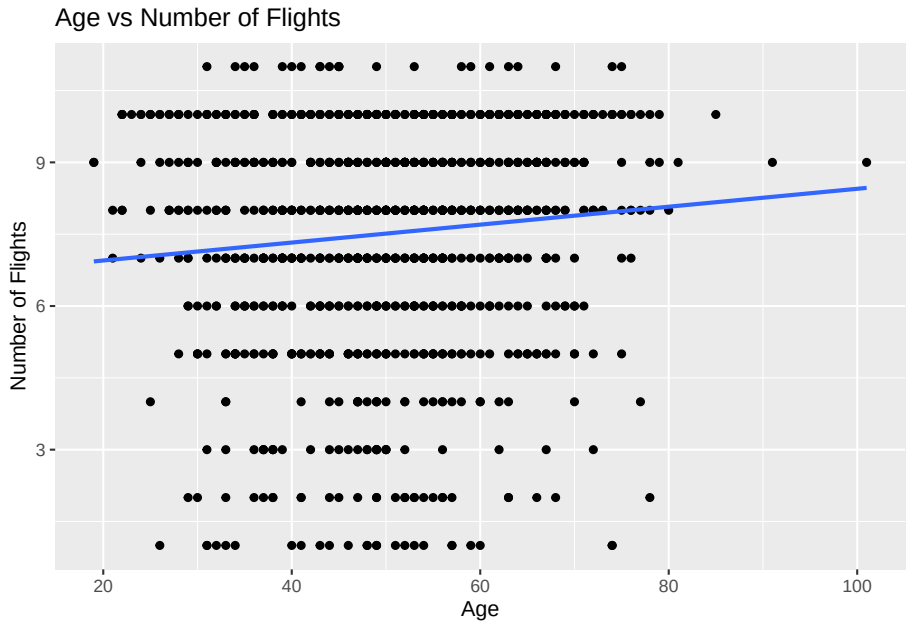


7.3.3.2 Scatterplot with a Trendline

Scatterplots become more helpful when we add a trend line. The most common trend line is a simple regression line, although others can be used. Use `geom_smooth(method = "lm", se = FALSE)` to add a linear trend line.

```
ggplot(airlinesat_small, aes(x = age, y = nps)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
  labs(title = "Age vs Number of Flights",
       x = "Age",
       y = "Number of Flights")
```

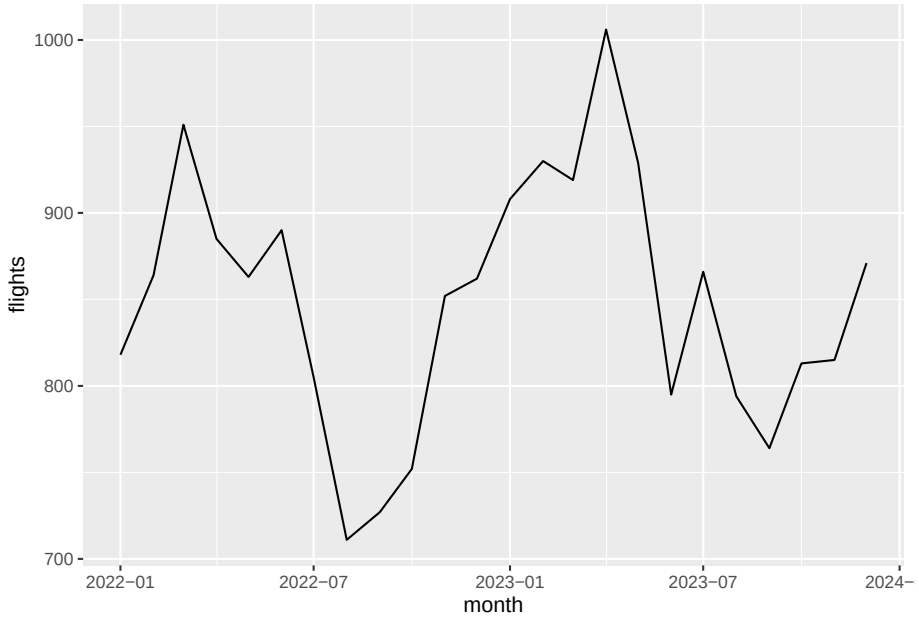
```
`geom_smooth()` using formula = 'y ~ x'
```



7.3.4 Line Chart

Line charts are drawn with the `geom_line()` geom and are used to emphasize change across an ordered dimension rather than individual observations. We'll use the simulated data from the line chart in base R created above.

```
ggplot(flight_ts, aes(x = month, y = flights)) +
  geom_line()
```

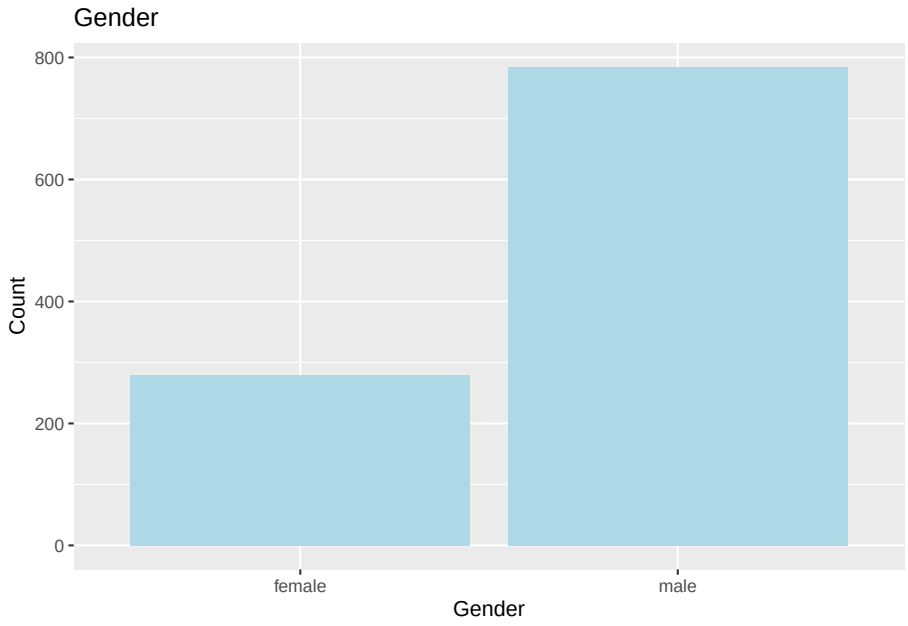


7.3.5 Bar Chart

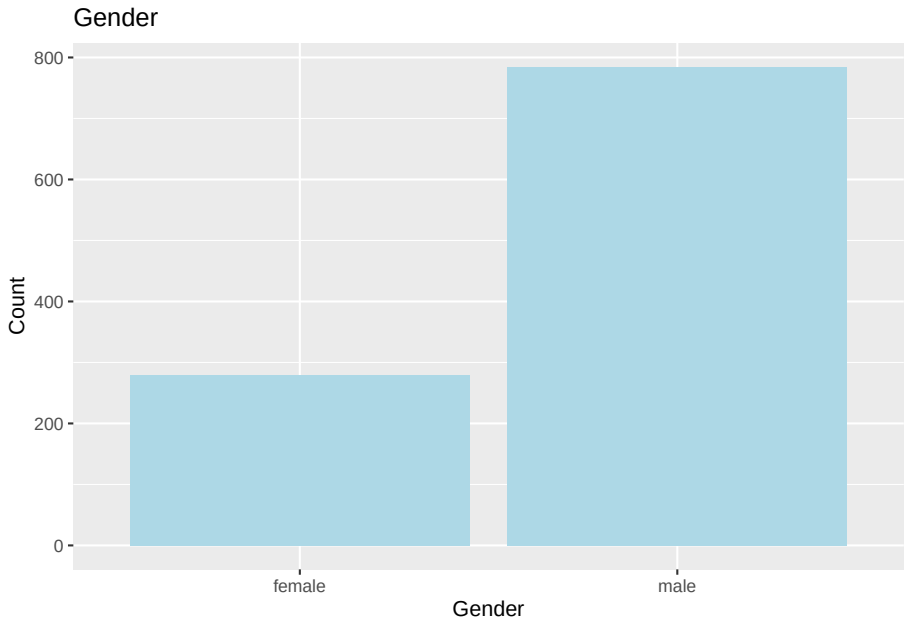
In `ggplot2`, bar charts, `geom_bar()`, are often used for plotting a single discrete variable, while column charts, `geom_col()`, are used for plotting a discrete variable on the x axis and a continuous variable on the y axis. However, `geom_col()` can be used for both if the data is "preformatted", which is easy with `dplyr`. Other than the first example below, all other examples use `geom_col()` as it tends to be more flexible when combined with `dplyr`.

The second example below first organizes the data using `dplyr` and then passes that result to `ggplot`.

```
ggplot(airlinesat_small, aes(x = gender)) +
  geom_bar(fill = "lightblue") +
  labs(title = "Gender",
       x = "Gender",
       y = "Count")
```



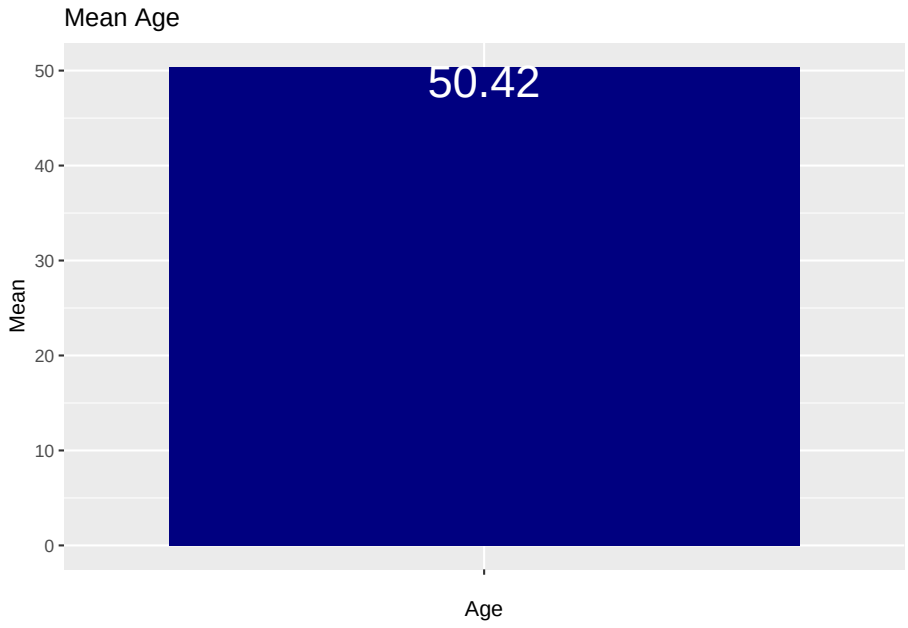

```
airlinesat_small %>%
  group_by(gender) %>%
  summarise(n=n()) %>%
  ggplot(aes(x = gender, y = n)) +
  geom_col(fill = "lightblue") +
  labs(title = "Gender",
       x = "Gender",
       y = "Count")
```



7.3.5.1 Bar Charts for Continuous Variables

Bar charts can be used to show a summary statistic (e.g., mean, median, etc.) of a continuous variable. This is easily done with `dplyr`. We can also add labels using the `geom_text()` or `geom_label()` geoms. `geom_text()` tends to be more flexible. By default, `ggplot` places the label at the exact height of the value, but that will overlap with the bar itself. Use the `vjust =` option to change the position of the label vertically. Usually `vjust = 0.95` puts the label just inside the top of the bar, while `vjust = -.05` puts the label just outside the top of the bar. You can also change the color of the label with `color =`, the size of the label with `size =`, and you can round the label to a specific number of digits with `round()`.

```
airlinesat_small %>%
  summarise(mean_age=mean(age)) %>%
  ggplot(aes(x = "", y = mean_age)) +
  geom_col(fill = "navyblue") +
  geom_text(aes(label = round(mean_age,2)),
            vjust = .95,
            color="white",
            size=8) +
  labs(title = "Mean Age",
       x = "Age",
       y = "Mean")
```



7.3.6 Grouped Bar Chart

When used with `dplyr`, it is easy to create side-by-side or stacked bar charts. In the code below, first a table is created and displayed using `dplyr`. Ultimately, we want to take those results pass them to a `ggplot` to create the chart.

7.3.6.1 Stacked Bar Chart with Counts

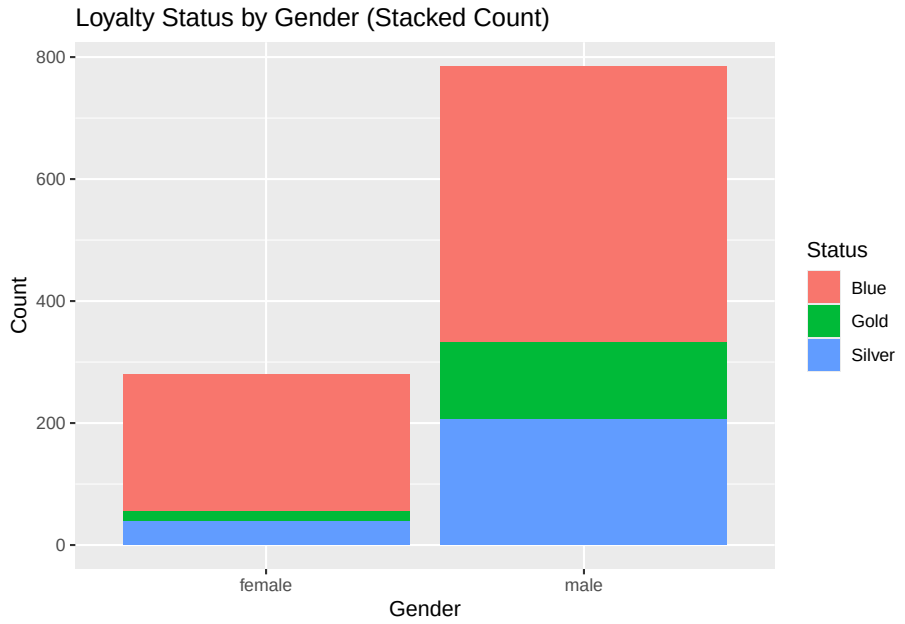
The default is to "stack" the counts.

```
airlinesat_small %>%
  group_by(gender, status) %>%
  summarise(n=n())
```

```
# A tibble: 6 x 3
# Groups:   gender [2]
  gender status     n
  <fct>  <fct>  <int>
1 female Blue     225
2 female Gold      16
3 female Silver    39
4 male   Blue     452
5 male   Gold     127
```

```
6 male   Silver   206
```

```
airlinesat_small %>%
  group_by(gender, status) %>%
  summarise(n=n()) %>%
  ggplot(aes(x = gender, y = n, fill = status)) +
  geom_col() +
  labs(title = "Loyalty Status by Gender (Stacked Count)",
       x = "Gender",
       y = "Count",
       fill = "Status")
```



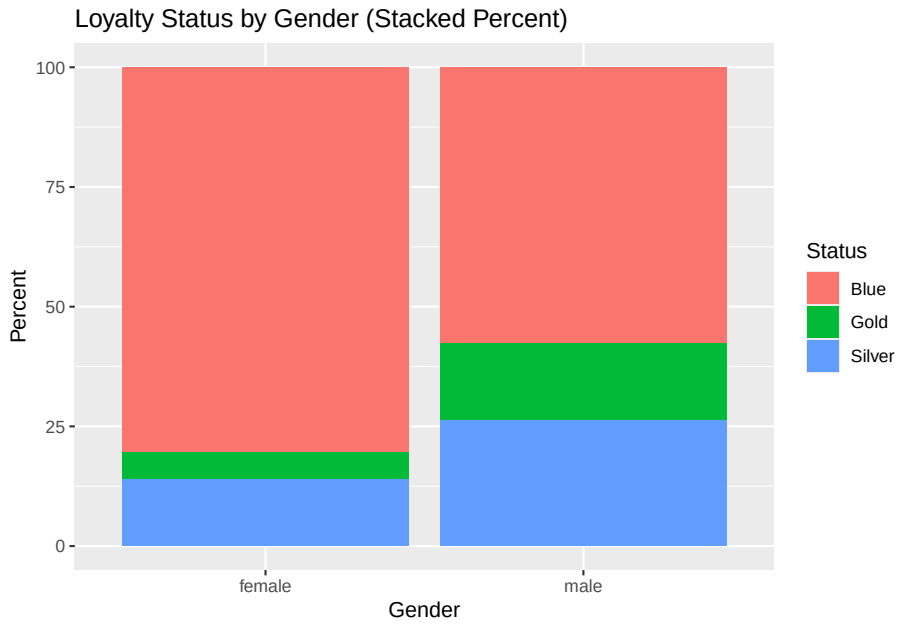
7.3.6.2 Stacked Bar Chart with Percentages

We can create "100%" stacked bar charts by calculating the percentages in the `dplyr` code. With the code below, the percentages will add up to 100% for the first `group_by` variable.

```
airlinesat_small %>%
  group_by(gender, status) %>%
  summarise(n=n()) %>%
  mutate(perc = 100 * n/sum(n))
```

```
# A tibble: 6 x 4
# Groups:   gender [2]
  gender status     n  perc
  <fct>  <fct>  <int> <dbl>
1 female Blue     225 80.4
2 female Gold      16  5.71
3 female Silver    39 13.9
4 male   Blue     452 57.6
5 male   Gold     127 16.2
6 male   Silver   206 26.2
```

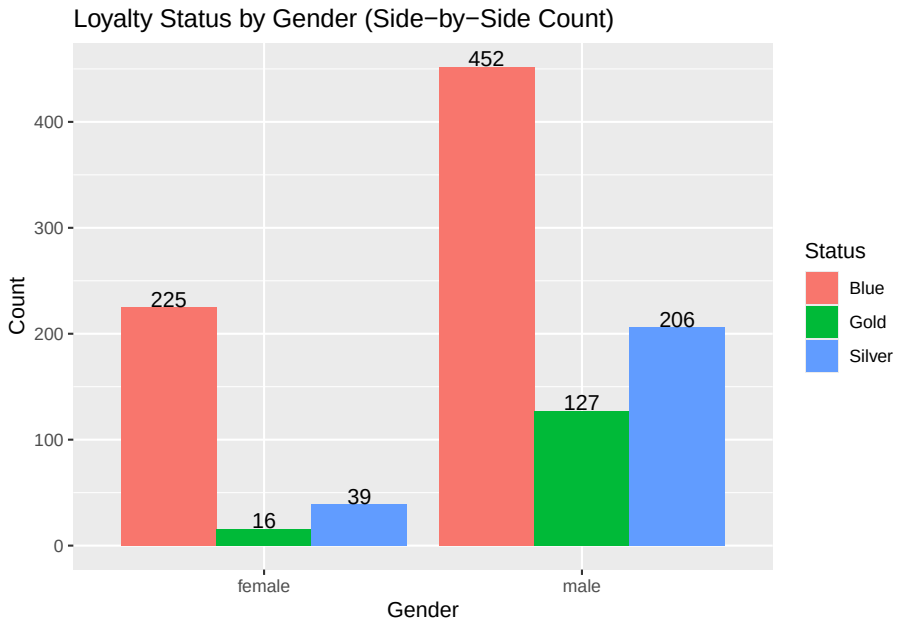
```
airlinesat_small %>%
  group_by(gender, status) %>%
  summarise(n=n()) %>%
  mutate(perc = 100 * n/sum(n)) %>%
  ggplot(aes(x = gender, y = perc, fill = status)) +
  geom_col() +
  labs(title = "Loyalty Status by Gender (Stacked Percent)",
       x = "Gender",
       y = "Percent",
       fill = "Status")
```



7.3.6.3 Side-by-Side Bar Chart with Counts/Percentages

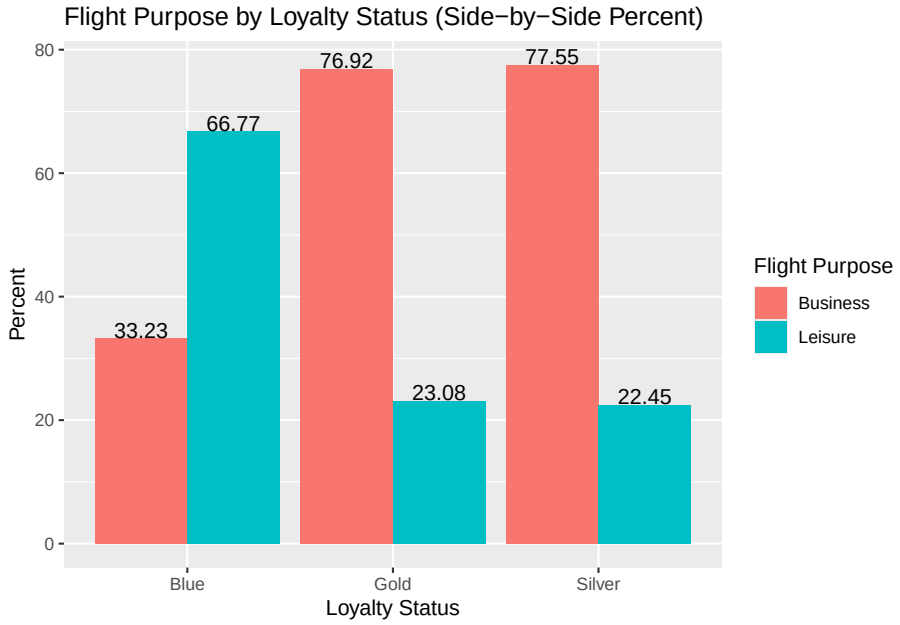
To create side-by-side bar charts (vs. stacked), use the `position = position_dodge(width=.9)` option within the `geom_col()`. If you add labels, you need to use `position = position_dodge(width=.9)` in the `geom_text`, otherwise the labels will be on top of each other.

```
airlinesat_small %>%
  group_by(gender, status) %>%
  summarise(n=n()) %>%
  ggplot(aes(x = gender, y = n, fill = status)) +
  geom_col(position = position_dodge(width=.9)) +
  geom_text(aes(label = n),
            position = position_dodge(width=.9),
            vjust = -.05) +
  labs(title = "Loyalty Status by Gender (Side-by-Side Count)",
       x = "Gender",
       y = "Count",
       fill = "Status")
```




```
airlinesat_small %>%
  group_by(status, flight_purpose) %>%
  summarise(n = n()) %>%
  mutate(perc = 100*n/sum(n)) %>%
```

```
  ggplot(aes(x = status, y = perc, fill = flight_purpose)) +
  geom_col(position = position_dodge(width=.9)) +
  geom_text(aes(label = round(perc, 2)),
            position = position_dodge(width=.9),
            vjust = -.05) +
  labs(title = "Flight Purpose by Loyalty Status (Side-by-Side Percent)",
       x = "Loyalty Status",
       y = "Percent",
       fill = "Flight Purpose")
```

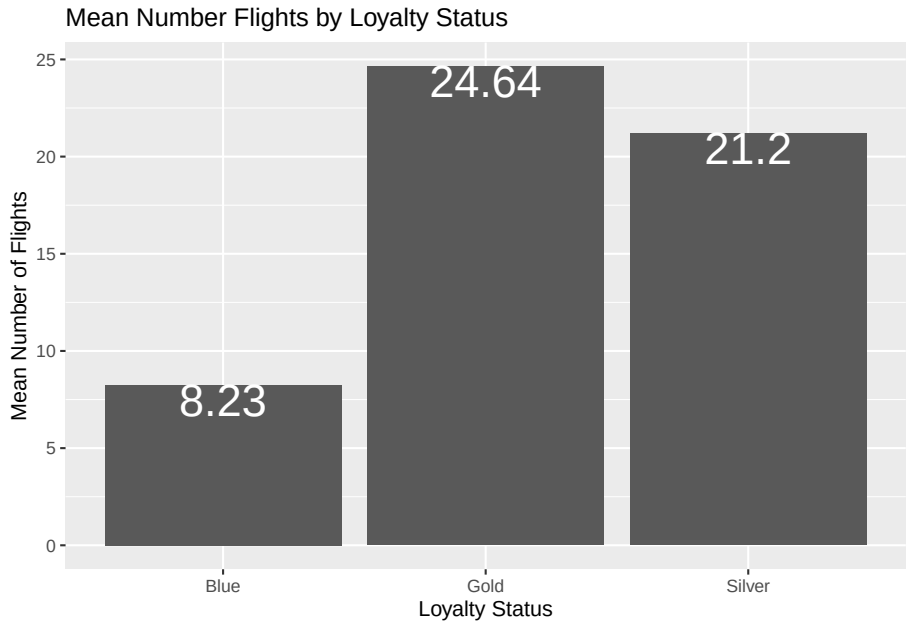


7.3.6.4 Side-by-Side Bar Chart with Continuous Variable

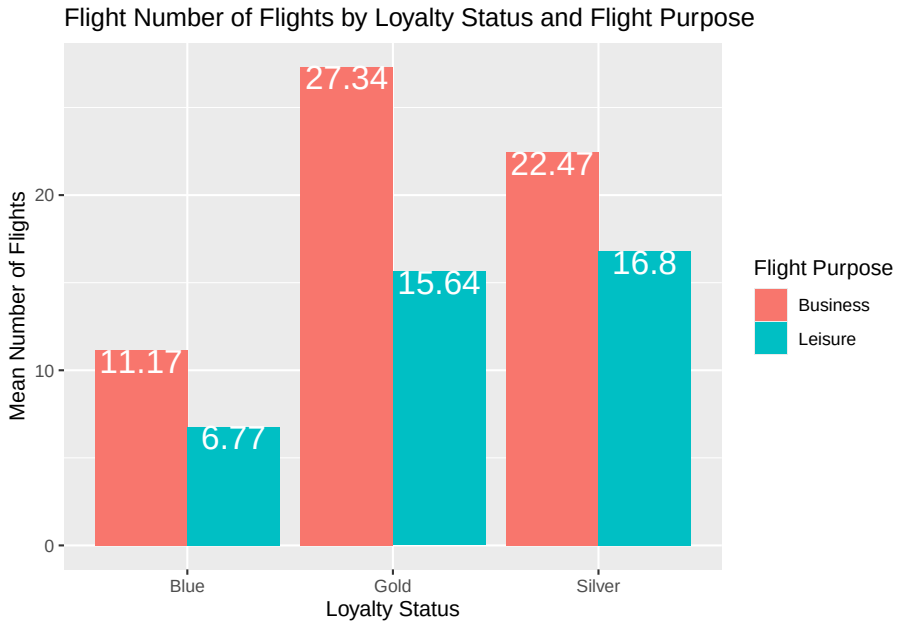
We can also calculate a summary statistic (e.g., mean, median, etc.) for a continuous variable and use that to create a side-by-side bar chart of a discrete variable or two discrete variables.

```
airlinesat_small %>%
  group_by(status) %>%
  summarise(mean=mean(nflights)) %>%
  ggplot(aes(x = status, y = mean)) +
  geom_col(position = position_dodge(width=.9)) +
  geom_text(aes(label = round(mean, 2)),
            position = position_dodge(width=.9),
```

```
            vjust = .95,
            color="white",
            size=8) +
  labs(title = "Mean Number Flights by Loyalty Status",
       x = "Loyalty Status",
       y = "Mean Number of Flights")
```




```
airlinesat_small %>%
  group_by(status, flight_purpose) %>%
  summarise(mean = mean(nflights)) %>%
  ggplot(aes(x = status, y = mean, fill = flight_purpose)) +
  geom_col(position = position_dodge(width=.9)) +
  geom_text(aes(label = round(mean, 2)),
            position = position_dodge(width=.9),
            vjust = .95,
            color="white",
            size=6) +
  labs(title = "Flight Number of Flights by Loyalty Status and Flight Purpose",
       x = "Loyalty Status",
       y = "Mean Number of Flights",
       fill = "Flight Purpose")
```



7.4 Summary

Base R graphics provide a simple way to create quick plots, while `ggplot2` offers a more flexible and extensible framework for data visualization. In practice, Base R is useful for fast checks, and `ggplot2` is preferred for exploratory analysis and communication.

7.5 What's Next

In the next chapter, we move from visual summaries to formal modeling. You will learn how to use linear regression to quantify relationships between variables, estimate marginal effects, and make predictions while holding other factors constant.

Chapter 8

Linear Regression

Linear regression is one of the most widely used tools in marketing analytics. It allows us to quantify relationships between an outcome variable and one or more predictors, helping us explain variation, estimate marginal effects, and generate predictions.

In earlier chapters, we summarized data numerically and visually. In this chapter, we move beyond description to modeling relationships between variables.

Throughout this chapter, we use the `airlinesat_small` dataset and focus on interpretation rather than mathematical derivation.

8.1 The Linear Regression Model

A linear regression model relates an outcome variable to one or more predictors. In R, linear regression models are estimated using the `lm()` function.

The general structure is:

```
lm(outcome ~ predictors, data = dataset)
```

8.2 Simple Linear Regression

We begin with a simple linear regression model containing a single predictor.

In this example, we model Net Promoter Score (`nps`) as a function of the number of flights taken (`nflights`). If we don't ask for a summary we **only** get the coefficients. If we save the result as an object and then ask for a summary, we get the full, expected results.

```
lm(nps ~ nflights, data = airlinesat_small)
```

```
Call:
lm(formula = nps ~ nflights, data = airlinesat_small)

Coefficients:
(Intercept)     nflights
    7.65848     -0.01031
```

```
model_simple <- lm(nps ~ nflights, data = airlinesat_small)
summary(model_simple)
```

```
Call:
lm(formula = nps ~ nflights, data = airlinesat_small)

Residuals:
    Min      1Q  Median      3Q     Max
-6.6482 -1.4318  0.4137  1.5476  8.0512

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  7.658478   0.085356   89.72  < 2e-16 ***
nflights    -0.010306   0.003518   -2.93  0.00347 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.321 on 1063 degrees of freedom
Multiple R-squared:  0.00801,   Adjusted R-squared:  0.007077
F-statistic: 8.583 on 1 and 1063 DF,  p-value: 0.003465
```

Interpretation focuses on:

- The intercept: the predicted net promoter score (NPS) when the number of flights is zero
- The slope on `nflights`: the expected change in NPS for one additional flight
- R-squared: the proportion of variation in NPS explained by the model

8.2.1 "Nice" Output

For nicer looking output, we can use either the `summ` function from the `jtools` package. It easily allows us to adjust the number of digits shown (`digits = #` option). It can also show the standardized beta coefficients by using the `scale=TRUE` and `transform.response=TRUE` options together.

```
summ(model_simple, digits = 4)
```

Observations	1065
Dependent variable	nps
Type	OLS linear regression

F(1,1063)	8.5832
R^2	0.0080
Adj. R^2	0.0071

	Est.	S.E.	t val.	p
(Intercept)	7.6585	0.0854	89.7243	0.0000
nflights	-0.0103	0.0035	-2.9297	0.0035

Standard errors: OLS

```
summ(model_simple, digits = 4, scale = TRUE, transform.response = TRUE)
```

Observations	1065
Dependent variable	nps
Type	OLS linear regression

F(1,1063)	8.5832
R^2	0.0080
Adj. R^2	0.0071

	Est.	S.E.	t val.	p
(Intercept)	0.0000	0.0305	0.0000	1.0000
nflights	-0.0895	0.0305	-2.9297	0.0035

Standard errors: OLS; Continuous variables are mean-centered and scaled by 1 s.d.

8.3 Multiple Linear Regression

In practice, outcomes are influenced by more than one factor. Multiple linear regression allows us to include additional predictors to control for other characteristics.

8.3.1 Adding Additional Variables

```
model_multi <- lm(nps ~ nflights + age, data = airlinesat_small)
summ(model_multi, digits = 4)
```

Observations	1065
Dependent variable	nps
Type	OLS linear regression

F(2,1062)	8.5875
R^2	0.0159
Adj. R^2	0.0141

	Est.	S.E.	t val.	p
(Intercept)	6.7861	0.3106	21.8516	0.0000
nflights	-0.0091	0.0035	-2.5822	0.0100
age	0.0170	0.0058	2.9208	0.0036

Standard errors: OLS

When additional predictors are included, coefficients are interpreted as marginal effects holding other variables constant.

8.4 Categorical Predictors and Reference Groups

Regression models can include categorical predictors. In R, factor variables are automatically converted into indicator (dummy) variables.

8.4.1 Example: Flight Class

```
model_cat <- lm(nps ~ nflights + flight_class,  data = airlinesat_small)
summ(model_cat, digits = 4)
```

Observations	1065
Dependent variable	nps
Type	OLS linear regression

F(3,1061)	4.1443
R^2	0.0116
Adj. R^2	0.0088

	Est.	S.E.	t val.	p
(Intercept)	7.7273	0.1793	43.0949	0.0000
nflights	-0.0108	0.0035	-3.0691	0.0022
flight_classEconomy	-0.0946	0.1881	-0.5029	0.6151
flight_classFirst	1.0666	0.6232	1.7115	0.0873

Standard errors: OLS

One category is treated as the reference group. Coefficients for other categories represent differences relative to that reference. In the example above, "Business" is treated as the reference group.

To change the reference group, use `relevel()`. It can be changed temporarily within the call to `lm()`, or it can be changed permanently in the dataframe.

```
# Temporary level chance in formula
model_cat_relevel <- lm(nps ~ nflights + relevel(flight_class, ref="Economy"),
                        data = airlinesat_small)
summ(model_cat_relevel, digits = 4)
```

Observations	1065
Dependent variable	nps
Type	OLS linear regression

```
airlinesat_small$flight_class <- relevel(airlinesat_small$flight_class,
                                         ref = "Economy")
```

F(3,1061)	4.1443
R^2	0.0116
Adj. R^2	0.0088

	Est.	S.E.	t val.	p
(Intercept)	7.6327	0.0907	84.1145	0.0000
nflights	-0.0108	0.0035	-3.0691	0.0022
relevel(flight_class, ref = "Economy")Business	0.0946	0.1881	0.5029	0.6151
relevel(flight_class, ref = "Economy")First	1.1612	0.6052	1.9187	0.0553

Standard errors: OLS

8.5 Interaction Effects

An interaction allows the effect of one predictor to depend on the level of another predictor. Interactions can be included in the formula using `*` or `:`

- `*` will include the interaction term **AND** each main effect
- `:` will include **ONLY** the interaction term
- Examples:
  - `y ~ x1 + x2 * x3` is the same as:
    $y = x1 + x2 + x3 + (x2 \times x3)$
  - `y ~ x1 + x2:x3 is the same as: $y = x1 + (x2 \times x3)$` 0y ~ x1 + x2 + x2:x3‘ is the same as:
    $y = x1 + x2 + (x2 \times x3)$

8.5.1 Example: Flights and Frequent Flier Status

```
model_inter <- lm(nps ~ age + nflights * status, data = airlinesat_small)
summ(model_inter, digits = 4)
```

Observations	1065
Dependent variable	nps
Type	OLS linear regression

F(6,1058)	5.0753
R^2	0.0280
Adj. R^2	0.0225

	Est.	S.E.	t val.	p
(Intercept)	6.8771	0.3136	21.9308	0.0000
age	0.0154	0.0058	2.6320	0.0086
nflights	0.0003	0.0046	0.0565	0.9550
statusGold	0.1997	0.3155	0.6330	0.5269
statusSilver	0.0803	0.2528	0.3178	0.7507
nflights:statusGold	-0.0158	0.0104	-1.5241	0.1278
nflights:statusSilver	-0.0266	0.0097	-2.7348	0.0063

Standard errors: OLS

8.6 Margin Plots with `easy_mp`

Rather than using exploratory plots or manually creating margin plots, we use the `easy_mp()` function from the `MKT4320BGSU` package to visualize predicted values and marginal effects.

8.6.1 The `easy_mp()` Function

This function creates marginal effects plots for a focal predictor (with or without an interaction) from a linear regression (`lm`) or binary logistic regression (`glm` with `family = "binomial"`).

Usage: `easy_mp(model, focal, int = NULL)`

Arguments:

- `model` is a fitted `lm` model or binary logistic `glm` model (`family = "binomial"`).
- `focal` is the name of the focal predictor variable in quotations
- `int` is the name of the interaction variable in quotations. Can be excluded if only the focal variable is wanted or no interaction exists.

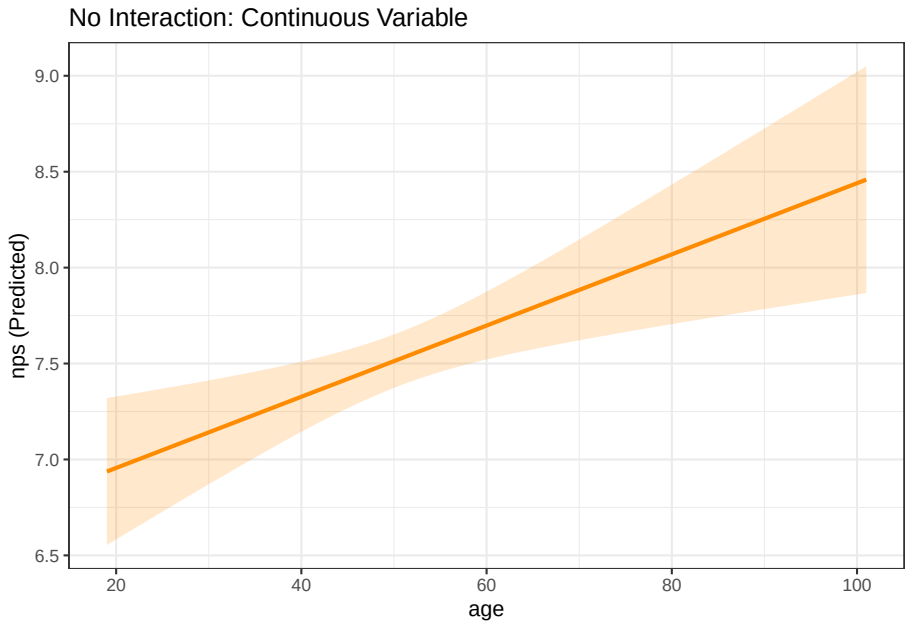
Returns: - `$plot` is the margin plot - `$ptable` is the marginal effects table used to produce the plot

8.6.2 WITHOUT Interactions

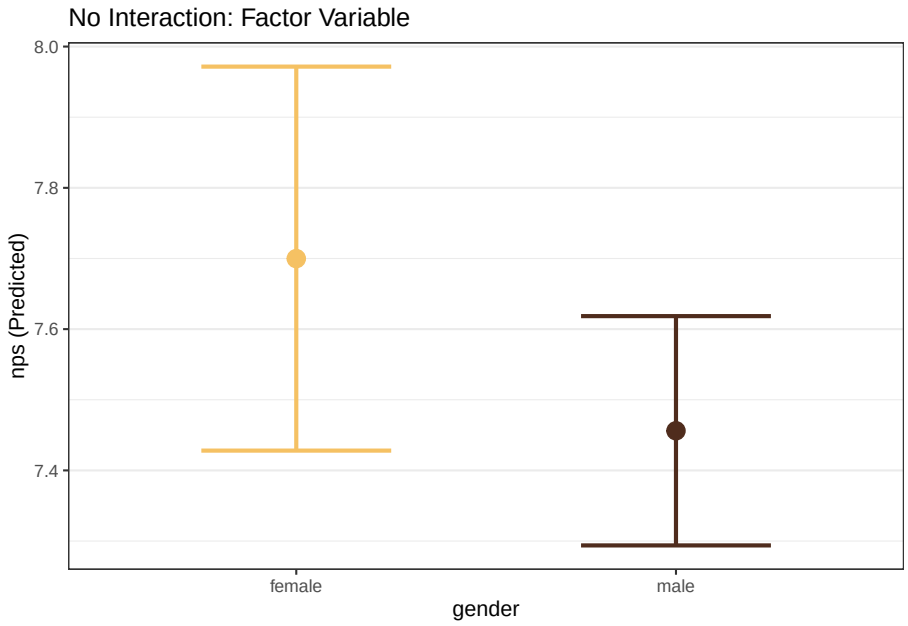
Below are examples of a margin plot for a continuous independent variable and for a categorical/factor independent variable. Note how the returned `$plot` object is a `ggplot` that can be modified by adding layers.

```
model_mp_nointer <- lm(nps ~ age + gender, data = airlinesat_small)

# Continuous Focal WITHOUT Interaction
mp_age <- easy_mp(model_mp_nointer, focal="age")
mp_age$plot +
  labs(title="No Interaction: Continuous Variable")
```




```
# Factor Focal WITHOUT Interaction
mp_gender <- easy_mp(model_mp_nointer, focal="gender")
mp_gender$plot +
  labs(title="No Interaction: Factor Variable")
```



8.6.3 WITH Interactions

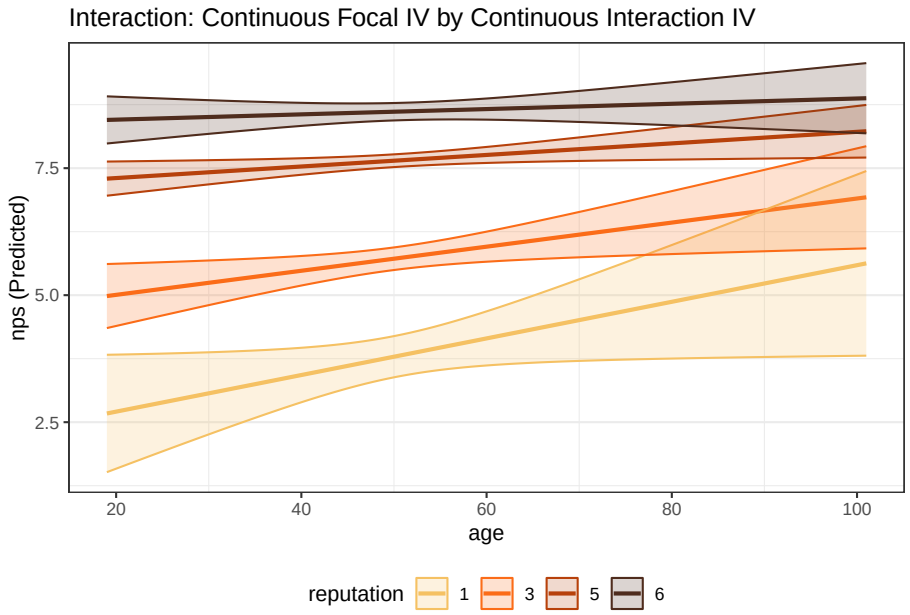
Ultimately, there are four types of margin plots that can be created depending on the focal variable type and the interaction variable type:

- Continuous Focal IV × Continuous Interaction IV
- Continuous Focal IV × Factor Interaction IV
- Factor Focal IV × Continuous Interaction IV
- Factor Focal IV × Factor Interaction IV

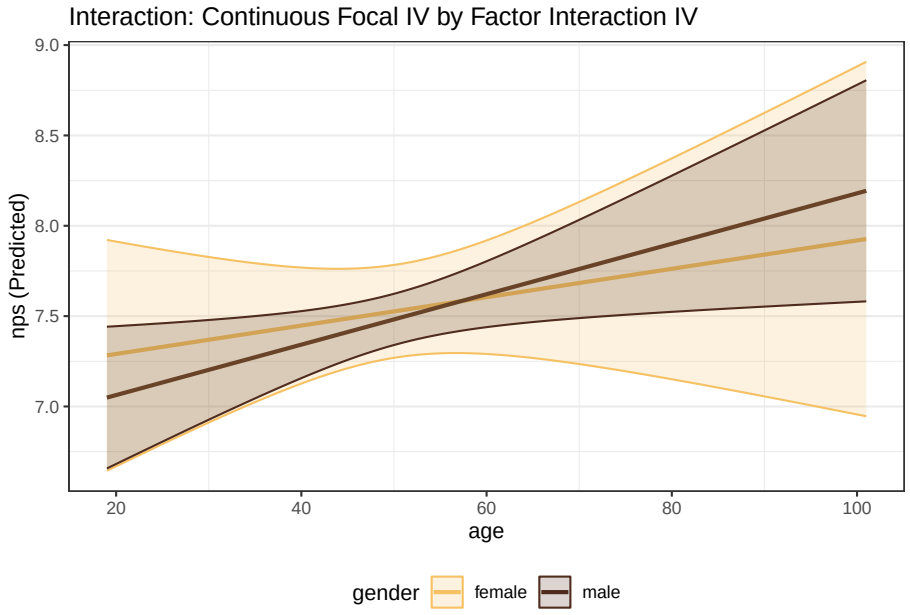
When the interaction term is continuous, the plot is created with representative values of the continuous variable (roughly the 1st percentile, the 99th percentile, and two evenly spaced values between those two).

```
model_mp_inter <- lm(nps ~ age*gender + age*reputation + gender*status, data = airlinesat_small)

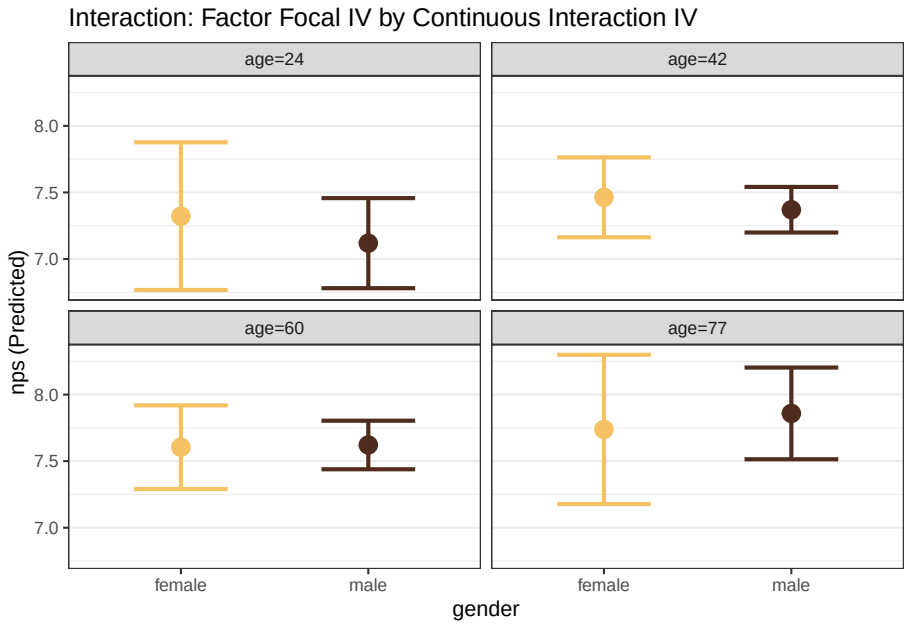
# Continuous Focal WITH Continuous Interaction
mp_age_reputation <- easy_mp(model_mp_inter, focal = "age", int = "reputation")
mp_age_reputation$plot +
  labs(title="Interaction: Continuous Focal IV by Continuous Interaction IV")
```




```
# Continuous Focal WITH Factor Interaction
mp_age_gender <- easy_mp(model_mp_inter, focal = "age", int = "gender")
mp_age_gender$plot +
  labs(title="Interaction: Continuous Focal IV by Factor Interaction IV")
```



```
# Factor Focal WITH Continuous Interaction
mp_gender_age <- easy_mp(model_mp_inter, focal = "gender", int = "age")
mp_gender_age$plot +
  labs(title="Interaction: Factor Focal IV by Continuous Interaction IV")
```




```
# Factor Focal WITH Factor Interaction
mp_gender_status <- easy_mp(model_mp_inter, focal = "gender", int = "status")
mp_gender_status$plot +
  labs(title="Interaction: Factor Focal IV by Factor Interaction IV")
```

8.7 Prediction

Regression models can also be used for prediction. The function `predict()` can be used to predict the DV based on values of the IVs. We can either: (1) predict the expected value of the DV for each observation in the data, or (2) predict the expected value of the DV for new values of the IV(s). To use this function for (2), we must pass a data frame of values to the function, where the data frame contains ALL of the IVs and the value for each IV that we want.

Suppose we wanted to predict, with a confidence interval, the `nps` of someone that is 45 years old and had 25 flights on the airline, and also someone that is 25 years old and had 45 flights on the airline, based on our `model_multi <- lm(nps ~ nflights + age, data = airlinesat_small)` from above. First, we create the data frame of values

```
values <- data.frame(age=c(45, 25), nflights=c(25, 45))
values
```

```
  age nflights
1  45       25
2  25       45
```

Second, the data frame is passed to the `predict()` function with confidence intervals requested.

```
predict(model_multi, values, interval="confidence")
```

```
       fit      lwr      upr
1 7.322601 7.153951 7.491252
2 6.800657 6.431058 7.170255
```

8.8 What's Next

In this chapter, you learned how to use linear regression to model relationships when the outcome variable is continuous, such as satisfaction or Net Promoter Score. Linear regression works well when predicted values can reasonably fall anywhere along a numeric scale.

Many marketing outcomes, however, are binary—for example, purchase vs. no purchase, churn vs. retention, or click vs. no click. In these cases, linear regression is no longer appropriate.

In the next chapter, we introduce binary logistic regression, a modeling approach designed specifically for yes/no outcomes. You will learn how to estimate probabilities, interpret coefficients and marginal effects, and evaluate model performance in a way that is well-suited to common marketing decision problems.

Chapter 9

Binary Logistic Regression

9.1 Why Logistic Regression in Marketing Analytics

Many important marketing decisions involve **binary outcomes**:

- Did the customer buy or not?
- Did the customer respond to a promotion?
- Did the customer churn?

In these cases, the dependent variable takes on only two possible values. A standard linear regression model is not appropriate because it can produce predicted values below 0 or above 1 and does not model probabilities correctly.

Binary logistic regression is designed specifically for situations where the outcome is binary. Rather than predicting the outcome directly, logistic regression models the **probability** that the outcome occurs.

From a marketing perspective, this is powerful: instead of simply predicting "buy" or "not buy," we can estimate how *likely* a customer is to buy and then use those probabilities for targeting and decision-making.

9.2 The Direct Marketing Data

In this chapter, we use the `directmktg` dataset, which contains information on a direct marketing campaign.

The dataset consists of **400 prospects** and the following variables:

- `userid`: Unique identifier for each prospect
- `age`: Prospect age in years
- `gender`: Prospect gender (coded as provided)
- `salary`: Prospect salary in thousands of dollars
- `buy`: Purchase decision ("Yes" or "No")

The marketing objective is:

Predict whether a prospect will purchase and estimate the probability of purchase.

```
data(directmktg)
```

9.3 Splitting Data into Training and Test Samples

To evaluate predictive performance, we split the data into training and test samples using `splitsample()` from the `MKT4320BGSU` package.

Usage:

- `splitsample(data, outcome = NULL, group = NULL, choice = NULL, alt = NULL, p = 0.75, seed = 4320)`
- where:
  - `data` is the data frame to split.
  - `outcome` is the outcome variable used for stratification. Required when group is `NULL`. Optional when group is provided. For binary logistic regression, it is required.
  - `group` is NOT USED FOR BINARY LOGISTIC
  - `choice` is NOT USED FOR BINARY LOGISTIC
  - `alt` is NOT USED FOR BINARY LOGISTIC
  - `p` is the proportion of observations to place in the training set. Must be strictly between 0 and 1. Default is 0.75.
  - `seed` is the random seed for reproducibility. Default is 4320.

```
sp <- splitsample(directmktg, outcome = "buy")
train <- sp$train
test  <- sp$test
```

The training data are used to estimate the model. The test data are reserved for out-of-sample evaluation.

9.4 Estimating a Binary Logistic Regression Model

We estimate a logistic regression model using `glm()` with `family = "binomial"`. Remember, we want to estimate the model with the `train` data we just created.

```
mod <- glm(buy ~ age + gender + salary, data = train, family = "binomial")
summary(mod)
```

```
Call:
glm(formula = buy ~ age + gender + salary, family = "binomial",
    data = train)

Coefficients:
               Estimate Std. Error z value Pr(>|z|)
(Intercept)  -13.166094   1.621695  -8.119 4.71e-16 ***
age            0.250215   0.032095   7.796 6.38e-15 ***
genderFemale  -0.406881   0.349811  -1.163    0.245
salary         0.040632   0.006742   6.026 1.68e-09 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 392.94  on 300  degrees of freedom
Residual deviance: 210.68  on 297  degrees of freedom
AIC: 218.68

Number of Fisher Scoring iterations: 6
```

The coefficients indicate how each variable affects the **log-odds** of purchase. The sign of each coefficient tells us whether the variable increases or decreases the likelihood of purchase.

For "nicer" looking results, the `summ()` function from the `jtools` package can be used.

```
library(jtools)
summ(mod, digits = 4)
```

Observations	301
Dependent variable	buy
Type	Generalized linear model
Family	binomial
Link	logit

$\chi^2(3)$	182.2574
p	0.0000
Pseudo-R² (Cragg-Uhler)	0.6231
Pseudo-R² (McFadden)	0.4638
AIC	218.6842
BIC	233.5126

	Est.	S.E.	z val.	p
(Intercept)	-13.1661	1.6217	-8.1187	0.0000
age	0.2502	0.0321	7.7961	0.0000
genderFemale	-0.4069	0.3498	-1.1631	0.2448
salary	0.0406	0.0067	6.0265	0.0000

Standard errors: MLE

9.5 Interpreting Odds Ratios

Because log-odds are difficult to interpret, we often convert coefficients to **odds ratios**. While we can do this by simply taking the natural exponents of the model coefficients (i.e., e^{coeff}), `exp(coef(mod))`, you can again use the `summ()` function from `jtools` and use the `exp = TRUE` option.

```
summ(mod, exp = TRUE, digits = 4)
```

Observations	301
Dependent variable	buy
Type	Generalized linear model
Family	binomial
Link	logit

$\chi^2(3)$	182.2574
p	0.0000
Pseudo-R^2 (Cragg-Uhler)	0.6231
Pseudo-R^2 (McFadden)	0.4638
AIC	218.6842
BIC	233.5126

	exp(Est.)	2.5%	97.5%	z val.	p
(Intercept)	0.0000	0.0000	0.0000	-8.1187	0.0000
age	1.2843	1.2060	1.3677	7.7961	0.0000
genderFemale	0.6657	0.3354	1.3215	-1.1631	0.2448
salary	1.0415	1.0278	1.0553	6.0265	0.0000

Standard errors: MLE

An odds ratio greater than 1 indicates higher odds of purchase as the predictor increases. An odds ratio less than 1 indicates lower odds.

9.6 Predicted Probabilities

Logistic regression naturally produces predicted probabilities.

```
train$phat <- predict(mod, type = "response")
head(train$phat)
```

```
           1            2            3            4            5            6
0.0004805918 0.0267022049 0.0109742358 0.0048495633 0.0170625797 0.0321671256
```

These probabilities are often more useful than hard classifications because they allow ranking and targeting. However, these probabilities are also used in evaluating model fit through the classification matrix. NOTE: You do not have to create these probabilities manually like shown above.

9.7 Classification and Cutoff Values

To classify prospects, we choose a probability cutoff (commonly 0.50). That is, a prospect/row is predicted to be "positive" if their predicted probability of the positive outcome is equal to 0.50 or greater. We use the `logistic_classify()` function from the `MKT4320BGSU` package to create classification matrices.

Usage:

- `classify_logistic(MOD, DATA, POSITIVE, CUTOFF = 0.5, DATA2 = NULL,`
  `LABEL1 = "Sample 1", LABEL2 = "Sample 2", digits = 3, ft = FALSE)`
- where:
  - `MOD` is a fitted binary logistic regression glm object (i.e., family = "binomial").
  - `DATA` a data frame for which the classification matrix should be produced. Usually `train`.
  - `POSITIVE` is the level in the outcome variable representing the positive outcome.
  - `CUTOFF` is the pobability cutoff for classification (default = 0.5).
  - `DATA2` is an optional second data frame (e.g., test/holdout sample).
  - `LABEL1` is the label for the first data set (default = "Sample 1")
  - `LABEL2` is the label for the section data set, if provided (default = "Sample 2")
  - `digits` is the number of decimal places (default = 3) to show.
  - `ft` is a logical operator to return a nicer looking flextable (default = FALSE).

```
classify_logistic(MOD = mod, DATA = train, POSITIVE = "Yes")
```

```
Classification Matrix - Sample 1 (Cutoff = 0.50)
Accuracy = 0.844
PCC = 0.540

       No Yes Total
No    176  30   206
Yes    17  78    95
Total 193 108   301
```

```
Selected Statistics (Positive Class):
  Sensitivity (TPR): 0.722
  Specificity (TNR): 0.912
  Precision (PPV):   0.821
```

We can also evaluate classification performance on the test sample and get the output in a "nicer" table.

```
classify_logistic(MOD = mod, DATA = train, DATA2 = test, POSITIVE = "Yes",
                  LABEL1 = "Training", LABEL2 = "Test", ft=TRUE)
```

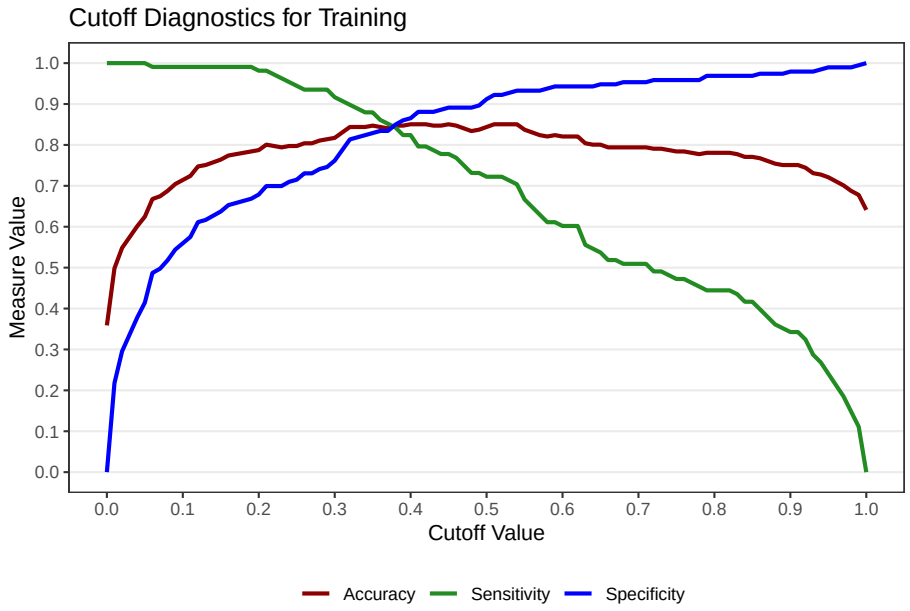
9.8 Choosing a Cutoff

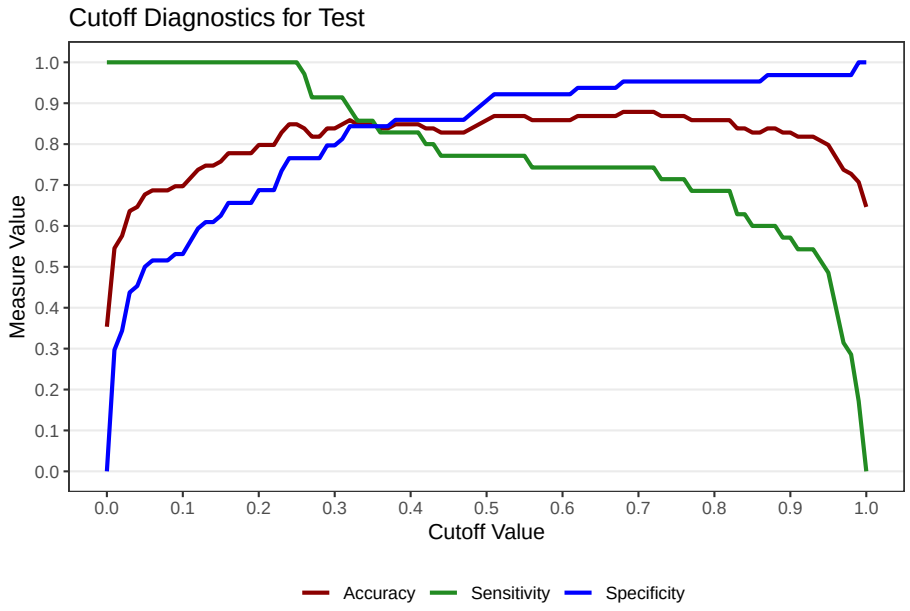
The default cutoff of 0.50 may not be optimal. The `cutoff_logistic()` function helps visualize tradeoffs between sensitivity, specificity, and accuracy.

Usage:

- `cutoff_logistic(MOD, DATA, POSITIVE, LABEL1 = "Sample 1", DATA2 = NULL, LABEL2 = "Sample 2")`
- where:
  - `MOD` is a fitted binary logistic regression model (glm with family = "binomial").
  - `DATA` a data frame for which the classification matrix should be produced. Usually `train`.
  - `POSITIVE` is the level in the outcome variable representing the positive outcome.
  - `LABEL1` is the label for the first data set (default = "Sample 1").
  - `DATA2` is an optional second data frame (e.g., test/holdout sample).
  - `LABEL2` is the label for the section data set, if provided (default = "Sample 2").

```
cut <- cutoff_logistic(MOD = mod, DATA = train, DATA2 = test, POSITIVE = "Yes",
                LABEL1 = "Training", LABEL2 = "Test")
```


Cutoff Diagnostics for Training
Measure Value
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
Cutoff Value
Accuracy
Sensitivity
Specificity


Cutoff Diagnostics for Test
Measure Value
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
Cutoff Value
Accuracy
Sensitivity
Specificity

9.9 ROC Curve and AUC

The ROC curve evaluates model discrimination across all possible cutoffs. The Area Under the Curve (AUC) summarizes overall classification performance. To get the ROC curve, use the `roc_logistic()` function from the `MKT4320BGSU` package.

Usage:

- `roc_logistic(MOD, DATA, LABEL1 = "Sample 1", DATA2 = NULL, LABEL2 = "Sample 2")`
- where:
  - `MOD` is a fitted binary logistic regression model (glm with family = "binomial").
  - `DATA` a data frame for ROC computation. Usually `train`.
  - `LABEL1` is the label for the first data set (default = "Sample 1").
  - `DATA2` is an optional second data frame (e.g., test/holdout sample) (default = NULL).
  - `LABEL2` is the label for the section data set, if provided (default = "Sample 2").

```
roc_logistic(MOD = mod, DATA = train, DATA2 = test,
             LABEL1 = "Training",LABEL2 = "Test")
```

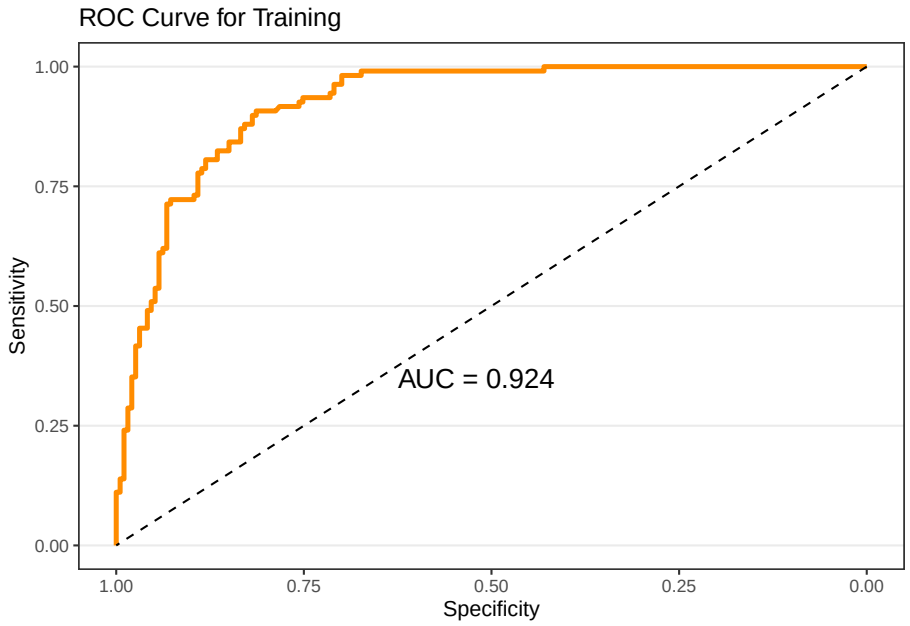
```
Setting levels: control = No, case = Yes
```

```
Setting direction: controls < cases
```

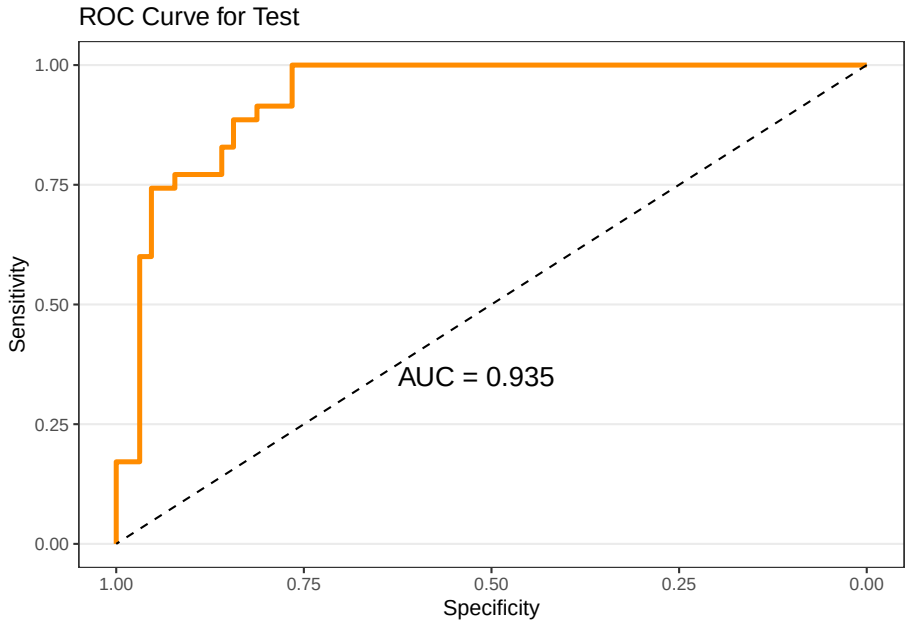
```
Setting levels: control = No, case = Yes
```

```
Setting direction: controls < cases
```

```
$sample1
```


ROC Curve for Training
Sensitivity
1.00
0.75
0.50
0.25
0.00
AUC = 0.924
1.00
0.75
0.50
0.25
0.00
Specificity

```
$sample2
```


ROC Curve for Test
Sensitivity
1.00
0.75
0.50
0.25
0.00
AUC = 0.935
1.00
0.75
0.50
0.25
0.00
Specificity

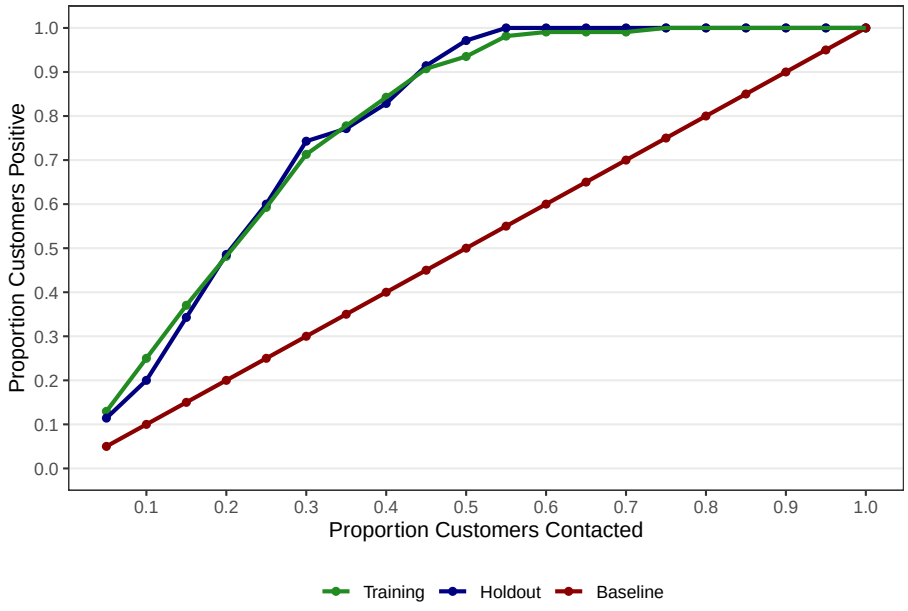
9.10 Gain and Lift Charts

Gain and lift charts are especially useful for targeting decisions. These plots (and tables) show how much better the model performs relative to random targeting. To get the gain and lift charts, use the `gainlift_logistic()` function from the `MKT4320BGSU` package.

Usage:

- `gainlift_logistic(MOD, TRAIN, TEST, POSITIVE)`
- where:
  - `MOD` is a fitted binary logistic regression model (glm with family = "binomial").
  - `TRAIN` a data frame with the training data (usually `train`).
  - `TEST` a data frame with the test/holdout data (usually `test`).
  - `POSITIVE` is the level in the outcome variable representing the positive outcome.

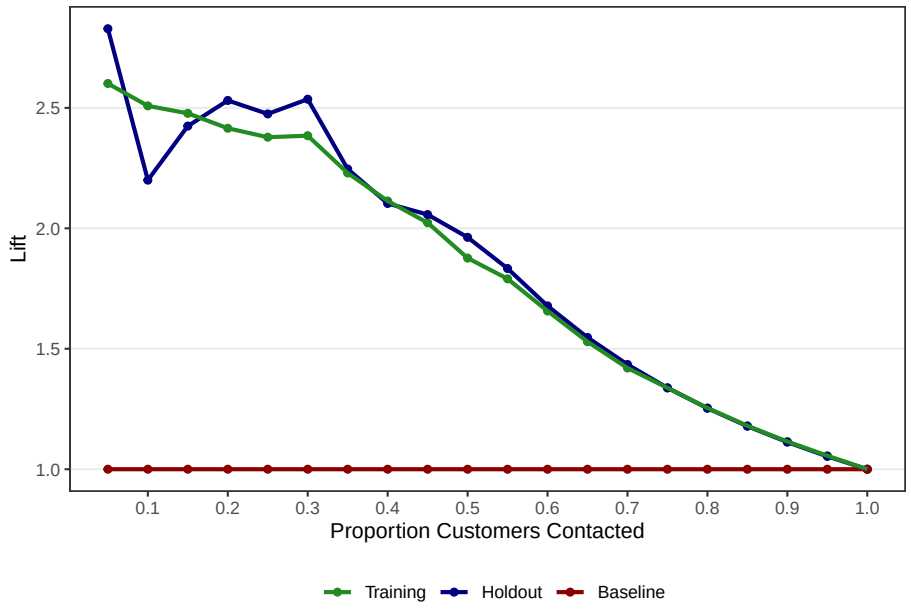
```
gl <- gainlift_logistic(MOD = mod, TRAIN = train, TEST = test, POSITIVE = "Yes")
gl$gainplot
```




```
gl$gaintable
```

```
# A tibble: 20 x 3
   `% Sample` Training Holdout
        <dbl>    <dbl>   <dbl>
 1       0.05    0.130   0.114
 2       0.1     0.25    0.2
 3       0.15    0.370   0.343
 4       0.2     0.481   0.486
 5       0.25    0.593   0.6
 6       0.3     0.713   0.743
 7       0.35    0.778   0.771
 8       0.4     0.843   0.829
 9       0.45    0.907   0.914
10       0.5     0.935   0.971
11       0.55    0.981   1
12       0.6     0.991   1
13       0.65    0.991   1
14       0.7     0.991   1
15       0.75    1       1
16       0.8     1       1
17       0.85    1       1
18       0.9     1       1
19       0.95    1       1
20       1       1       1
```

```
gl$liftplot
```



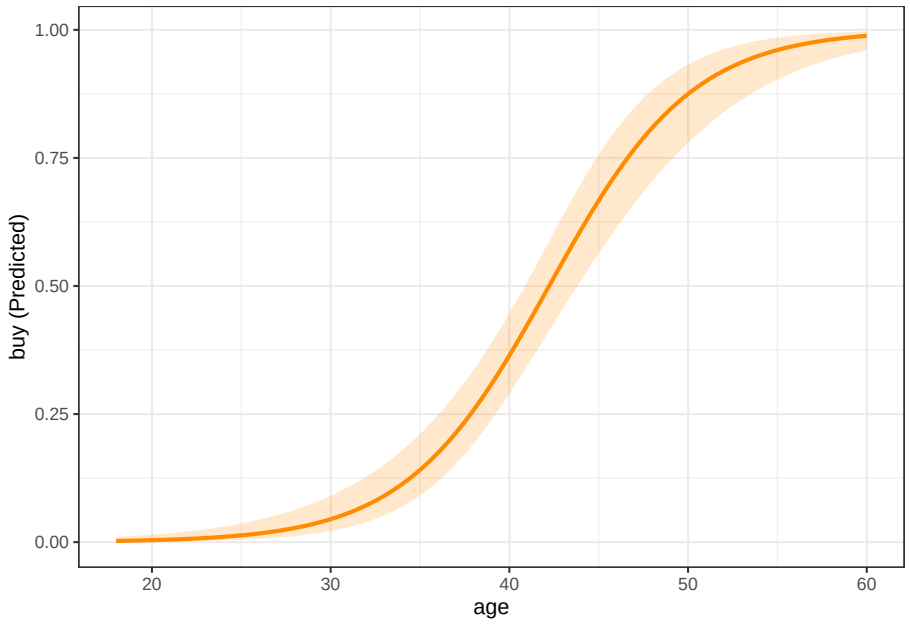
```
gl$lifttable
```

```
# A tibble: 20 x 3
   `% Sample` Training Holdout
        <dbl>    <dbl>   <dbl>
 1       0.05     2.60    2.83
 2       0.1      2.51    2.2
 3       0.15     2.48    2.42
 4       0.2      2.42    2.53
 5       0.25     2.38    2.48
 6       0.3      2.38    2.54
 7       0.35     2.23    2.25
 8       0.4      2.11    2.10
 9       0.45     2.02    2.06
10       0.5      1.88    1.96
11       0.55     1.79    1.83
12       0.6      1.66    1.68
13       0.65     1.53    1.55
14       0.7      1.42    1.43
15       0.75     1.34    1.34
16       0.8      1.25    1.25
17       0.85     1.18    1.18
18       0.9      1.11    1.11
19       0.95     1.06    1.05
20       1        1       1
```

9.11 Margin Plots with easy_mp()

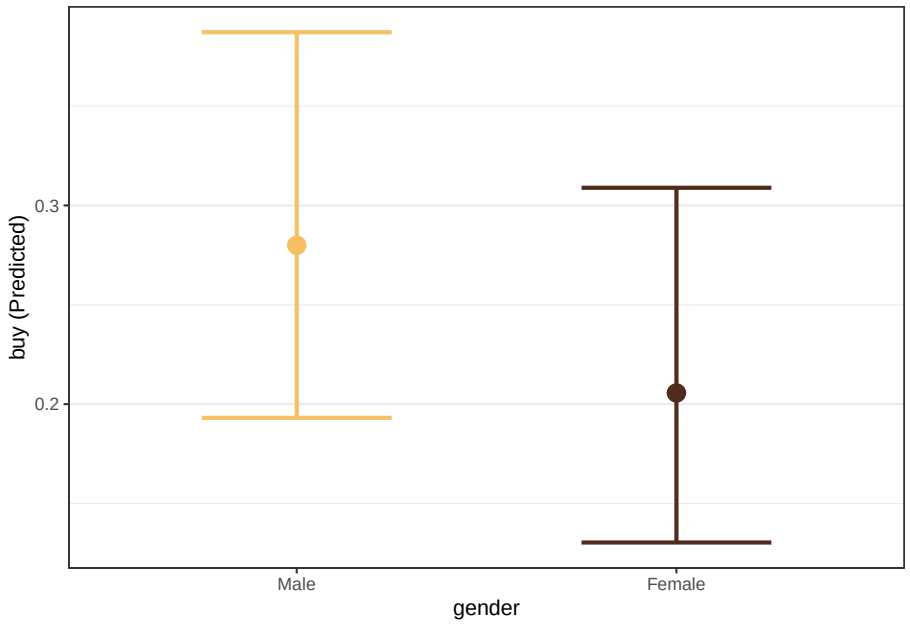
Coefficients are difficult to interpret directly. Marginal effects plots help visualize how predictors influence purchase probability. As with linear regression (see Section 8.6), we use the `easy_mp()` function from the `MKT4320BGSU` package to help create the margin plots.

```
mp_age <- easy_mp(mod, focal = "age")
mp_age$plot
```



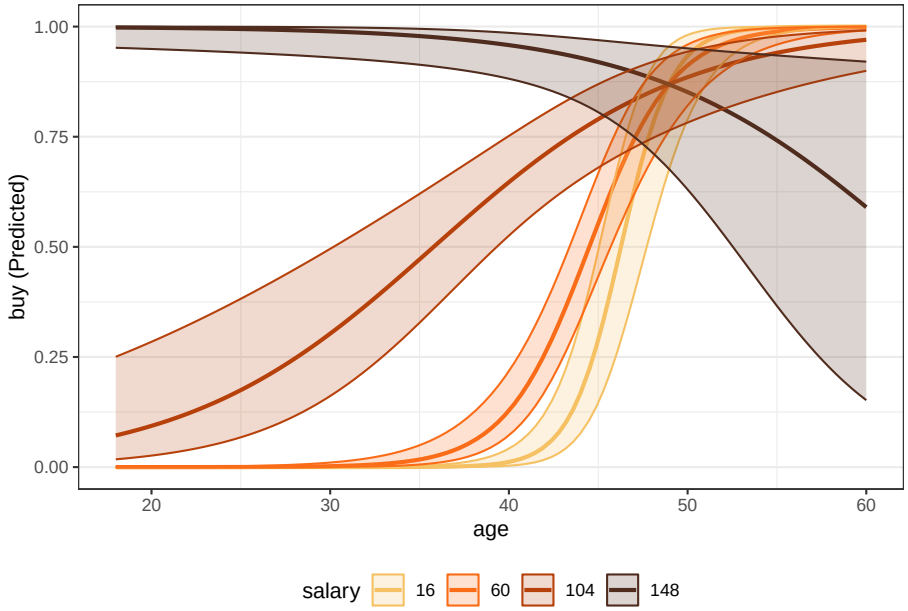
We can also examine categorical predictors.

```
mp_gender <- easy_mp(mod, focal = "gender")
mp_gender$plot
```



We can also examine interactions.

```
mod_int <- glm(buy ~ age * salary + gender, data = train, family = "binomial")
mp_gender <- easy_mp(mod_int, focal = "age", int="salary")
mp_gender$plot
```



9.12 Putting It All Together

Binary logistic regression provides a complete framework for:

- Predicting purchase probabilities
- Classifying prospects
- Evaluating models using multiple metrics
- Supporting targeting decisions

No single metric tells the whole story. Marketing analysts must choose evaluation tools that align with business objectives.

9.13 What's Next

In the next chapter, we shift from prediction to segmentation. We will use cluster analysis to:

- Identify distinct customer segments based on observed attributes
- Understand how customers naturally group together
- Support positioning, targeting, and strategy development
- Translate data-driven segments into actionable marketing insights

Where logistic regression answers questions like "Who is likely to buy?", cluster analysis addresses questions such as:

- What types of customers do we have?
- How are customers meaningfully different from one another?
- How many segments make sense from a managerial perspective?

The next chapter introduces several clustering approaches, discusses how to choose the number of clusters, and emphasizes interpretation and managerial usefulness over purely technical solutions.

Chapter 10

Cluster Analysis

10.1 Why Cluster Analysis Matters in Marketing

Cluster analysis is one of the most widely used tools in marketing analytics for market segmentation. Unlike regression or classification models, cluster analysis is an unsupervised learning technique: there is no dependent variable. Instead, the objective is to identify groups of customers who are similar to one another and meaningfully different from customers in other groups.

In marketing contexts, cluster analysis is commonly used to:

- Identify distinct customer segments
- Support targeting and positioning decisions
- Inform product design and messaging strategy
- Provide structure for downstream analysis and managerial reporting

Because cluster analysis does not optimize a predictive objective, interpretation and managerial judgment play a central role in determining whether a solution is useful.

10.2 The `ffseg` Dataset

This chapter uses the `ffseg` dataset, which contains survey responses related to fast-food consumption behaviors, attitudes, and demographics. Each row represents an individual consumer.

The dataset includes:

- Numeric attitudinal and behavioral variables suitable for segmentation
- Demographic and categorical variables used for describing clusters after they are formed

---

10.3 Hierarchical Agglomerative Clustering

10.3.1 Conceptual Overview

Hierarchical agglomerative clustering builds clusters from the bottom up. Each observation starts as its own cluster, and clusters are successively merged based on similarity until all observations belong to a single cluster.

Key features of hierarchical clustering include:

- No need to specify the number of clusters in advance
- A dendrogram that visualizes the clustering process
- Strong transparency for exploratory segmentation

10.3.2 Initial Hierarchical Clustering Fit

We begin by fitting a hierarchical clustering model using selected segmentation variables from the `ffseg` dataset. While base R can do this for us in a number of steps, we can more easily do the initial fit using the `easy_hc_fit()` function from the `MKT4320BGSU` package.

Usage:

- ```
  easy_hc_fit(data, vars, dist = c("euc", "euc2", "max",
  "abs", "bin"),
  method = c("ward", "single", "complete", "average"), k_range
  = 1:10,
  standardize = TRUE, show_dend = TRUE, dend_max_n = 300)
  ```
- where:
  - `data` is a data frame containing the full dataset.
  - `vars` is a character vector of numeric segmentation variable names.
  - `dist` is a distance measure to use. One of: “euc”, “euc2”, “max”, “abs”, “bin”.
  - `method` is a linkage method to use. One of: “ward”, “single”, “complete”, “average”.
  ```

- `k_range` is an integer vector of cluster solutions to evaluate (default 1:10; allowed 1–20).
- `standardize` is logical; if TRUE (default), standardizes segmentation variables prior to clustering.
- `show_dend` is logical; if TRUE (default), plots a dendrogram (skipped if n > dend_max_n).
- `dend_max_n` is an integer for the maximum sample size for drawing a dendrogram (default 300).

When using the `easy_hc_fit()` function, the results should be saved to an object. The `$stop` diagnostics table saved in the object provides the necessary results to help decide on a (potential) final solution. The diagnostics table summarizes multiple stopping rules and balance checks across different values of k. No single statistic should be used in isolation when choosing the number of clusters. You can also view the `$size_prop` table to see the cluster sizes for all potential solutions.

```
hc_vars <- c("quality", "price", "healthy", "variety", "speed")
hc_fit <- easy_hc_fit(data = ffseg, vars = hc_vars, dist = "euc",
                      method = "ward", k_range = 2:8)
```

```
Skipping dendrogram (n > dend_max_n).

Clustering k = 1,2,..., K.max (= 8): .. done
Bootstrapping, b = 1,2,..., B (= 20)  [one "." per sample]:
.................... 20
```

```
hc_fit$stop
```

Cluster Diagnostics (Euclidean Distance / Ward's D Linkage)

Clusters	Duda.Hart	pseudo.t2	Gap.Stat	Small.Prop	Large.Prop	CV
2	0.837	88.38	0.6212*	0.3949	0.6051 ◊	0.29
3	0.844	54.74	0.6048	0.2247	0.3949	0.28
4	0.849	50.52	0.5861	0.0465 ^	0.3803	0.605
5	0.745	58.58	0.5802	0.0465 ^	0.3484	0.557
6	0.863	41.36	0.5796	0.0465 ^	0.3484	0.640

Cluster Diagnostics (Euclidean Distance / Ward's D Linkage)					
Clusters	**Duda.Hart**	**pseudo.t2**	**Gap.Stat**	**Small.Prop**	**Large.Prop**
7	0.822	36.11	0.5744	0.0465 ^	0.2460
8	0.821	40.00	0.5747	0.0465 ^	0.2460

* 1-SE Gap rule.

^ Smallest cluster < 5% of sample.

◊ Largest cluster > 50% of sample.

• CV < 0.5 well balanced; •• moderately imbalanced.

```
hc_fit$size_prop
```

Cluster Size Proportions by Solution					
Solution	**Cluster 1**	**Cluster 2**	**Cluster 3**	**Cluster 4**	**Cluster 5**
2-cluster solution	0.3949	0.6051			
3-cluster solution	0.3949	0.3803	0.2247		
4-cluster solution	0.3484	0.3803	0.2247	0.0465	
5-cluster solution	0.3484	0.2301	0.2247	0.1503	0.04
6-cluster solution	0.3484	0.1290	0.1011	0.2247	0.15
7-cluster solution	0.2460	0.1290	0.1024	0.1011	0.22
8-cluster solution	0.2460	0.1290	0.1024	0.1011	0.13

10.3.3 Selecting the Number of Clusters

When selecting the number of clusters, consider:

- Statistical indicators such as the Gap statistic
- Whether clusters are reasonably balanced in size
- Whether the resulting segments are interpretable and actionable

Extremely small clusters or one dominant cluster are often warning signs in marketing segmentation.

10.3.4 Final Hierarchical Clustering Solution

Once a reasonable number of clusters has been selected, we finalize the hierarchical clustering solution and attach cluster membership back to the full dataset using the `easy_hc_final()` function from the `MKT4320BGSU` package.

Usage:

- `easy_hc_final(fit, data, k, cluster_col = "cluster", conf_level = 0.95,`
  `auto_print = TRUE)`
- where:
  - `fit` is the object returned by `easy_hc_fit`.
  - `data` is the original full dataset used to create fit.
  - `k` is an integer representing the number of clusters to extract.
  - `cluster_col` is the name of the cluster column to append to `data` (default "cluster"). Must not already exist in data.
  - `conf_level` confidence level for CI error bars when plotting (default 0.95).
  - `auto_print` is logical; if TRUE (default), prints props and profile and displays the plot (if available).

Using the example from above with a 3-cluster solution, the cluster profile table reports mean values of the segmentation variables for each cluster. These means should be interpreted in relative terms across clusters. The cluster plot visualizes those means.

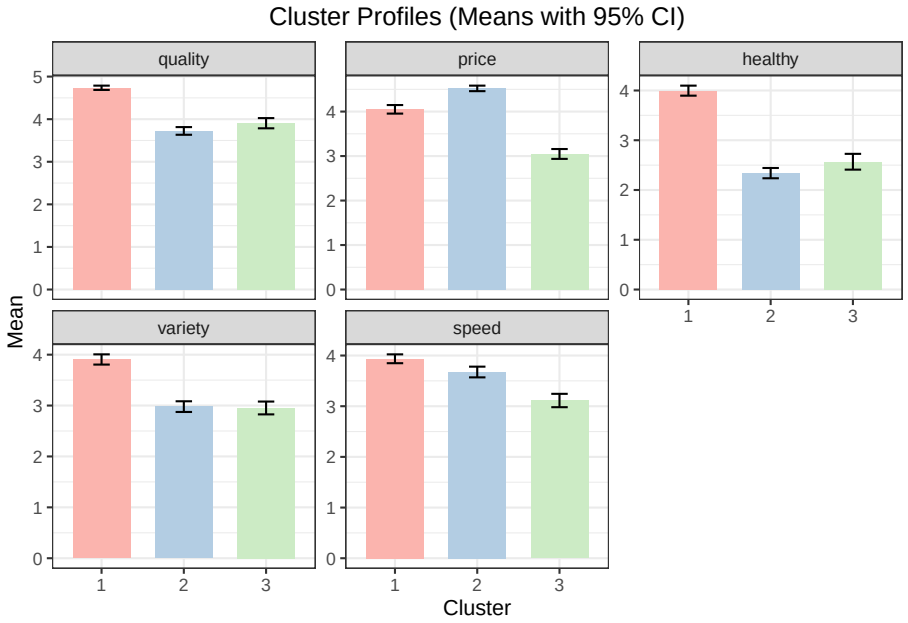
```
hc_final <- easy_hc_final(fit = hc_fit, data = ffseg, k = 3,
                          cluster_col = "hc_cluster", auto_print = FALSE)
hc_final$props
```

```
  Cluster   N      Prop
1       1 297 0.3949468
2       2 286 0.3803191
3       3 169 0.2247340
```

```
hc_final$profile
```

```
  Cluster  quality    price  healthy  variety    speed
1       1 4.737374 4.050505 3.996633 3.905724 3.936027
2       2 3.723776 4.520979 2.339161 2.979021 3.674825
3       3 3.905325 3.047337 2.568047 2.952663 3.112426
```

```
hc_final$plot
```



10.4 Describing and Labeling Hierarchical Clusters

Segmentation variables alone rarely provide enough context to understand who the clusters represent. Additional variables are used to describe and label the clusters. For both hierarchical clustering and k-means clustering (see below), use the `easy_cluster_decribe()` function from the `MKT4320BGSU` package to automate this process.

Usage:

- `easy_cluster_describe(data, cluster_col = "cluster", var, alpha = 0.05, drop_missing = TRUE, auto_print = TRUE), digits = 4`
- where:
  - `data` is the data frame containing the cluster membership column and variables to describe that was saved to the `easy_hc_final` object; should be `objectname$data`.

- `cluster_col` is a character string naming the cluster membership column in data (default = "cluster").
- `var` is a character string naming the single variable to describe.
- `alpha` is the significance level for hypothesis tests (default = 0.05).
- `drop_missing` is logical; if TRUE (default), rows with missing cluster membership are dropped prior to analysis.
- `auto_print` is logical; if TRUE (default), prints summaries to the console; if FALSE returns results list
- `digits` is an integer for rounding/formatting of numeric output (default = 4).

Using the results from above (i.e., the `hc_final$data` object) , the output below highlights which variables significantly differentiate clusters and provides detailed summaries only where differences are statistically meaningful.

```
easy_cluster_describe(data = hc_final$data, cluster_col = "hc_cluster",
                      var="usertype")
```

```
Variable: usertype (categorical)

Overall test
Variable  Test        p_value  Significant
--------  ----------  -------  -----------
usertype  Chi-square  0.05500  FALSE

Not significant at alpha = 0.05; no post-hoc shown.
```

```
easy_cluster_describe(data = hc_final$data, cluster_col = "hc_cluster",
                      var="spend")
```

```
Variable: spend (categorical)

Overall test
Variable  Test        p_value   Significant
--------  ----------  --------  -----------
spend     Chi-square  0.003500  TRUE

Cross-tab (column %; clusters are columns)
Level         1      2      3
------------  -----  -----  -----
$10 or more   25.25  17.48  26.63
$5 to $9      61.95  63.64  65.68
```

```
Less than $5  12.79  18.88  7.692

Post-hoc (Holm-adjusted p-values)
Row                       1:2      1:3     2:3
------------------------  -------  ------  --------
$10 or more:$5 to $9      0.3758   1.000   0.5700
$10 or more:Less than $5  0.06200  0.5700  0.002300
$5 to $9:Less than $5     0.5700   0.5700  0.03350
```

```
easy_cluster_describe(data = hc_final$data, cluster_col = "hc_cluster",
                      var="hours")
```

```
Variable: hours (continuous)

Overall test
Variable  Test   p_value  Significant
--------  -----  -------  -----------
hours     ANOVA  0        TRUE

Cluster means (N, Mean, SD)
Cluster  N      Mean   SD
-------  -----  -----  ------
1        297.0  3.710  0.9428
2        286.0  3.542  1.014
3        169.0  3.065  1.070

Significant differences: 1 > 3, 2 > 3

Post-hoc comparisons (Games-Howell)
.y.    group1  group2  estimate  conf.low  conf.high  p.adj            p.adj.signif
-----  ------  ------  --------  --------  ---------  ---------------  ------------
hours  1       2       -0.1685   -0.3592   0.02224    0.09600          ns
hours  1       3       -0.6453   -0.8781   -0.4126    0.0000000007900  ****
hours  2       3       -0.4769   -0.7166   -0.2372    0.00001220       ****
```

```
easy_cluster_describe(data = hc_final$data, cluster_col = "hc_cluster",
                      var="eatin")
```

```
Variable: eatin (continuous)

Overall test
Variable  Test   p_value  Significant
```

```
--------  -----  -------  -----------
eatin     ANOVA  0.02040  TRUE

Cluster means (N, Mean, SD)
Cluster  N      Mean   SD
-------  -----  -----  -----
1        297.0  3.986  2.000
2        286.0  4.441  1.978
3        169.0  4.160  1.903

Significant differences: 2 > 1

Post-hoc comparisons (Games-Howell)
.y.    group1  group2  estimate  conf.low  conf.high  p.adj    p.adj.signif
-----  ------  ------  --------  --------  ---------  -------  ------------
eatin  1       2       0.4540    0.06692   0.8411     0.01700  *
eatin  1       3       0.1732    -0.2664   0.6129     0.6230   ns
eatin  2       3       -0.2808   -0.7218   0.1602     0.2930   ns
```

```
easy_cluster_describe(data = hc_final$data, cluster_col = "hc_cluster",
                      var="mealplan")
```

```
Variable: mealplan (categorical)

Overall test
Variable  Test        p_value  Significant
--------  ----------  -------  -----------
mealplan  Chi-square  0.8882   FALSE

Not significant at alpha = 0.05; no post-hoc shown.
```

```
easy_cluster_describe(data = hc_final$data, cluster_col = "hc_cluster",
                      var="gender")
```

```
Variable: gender (categorical)

Overall test
Variable  Test        p_value  Significant
--------  ----------  -------  -----------
gender    Chi-square  0        TRUE

Cross-tab (column %; clusters are columns)
```

```
Level   1      2      3
------  -----  -----  -----
Female  81.82  67.83  59.17
Male    18.18  32.17  40.83

Post-hoc (Holm-adjusted p-values)
Row          1:2         1:3  2:3
-----------  ---------  ---  -------
Female:Male  0.0002000  0    0.06810
```

10.5 K-Means Clustering

10.5.1 Conceptual Overview

k-means clustering is a partition-based method that assigns observations to a fixed number of clusters by minimizing within-cluster variation. Unlike hierarchical clustering, k-means requires the analyst to specify the number of clusters in advance.

k-means is often preferred when:

- Working with larger datasets
- Refining a solution suggested by hierarchical clustering
- Stability and computational efficiency are priorities

The process used for k-means clustering follows the process used above in hierarchical agglomerative clustering. That is, we first initially fit a number of potential solutions, and then we anlayze a final solution.

10.5.2 Initial K-Means Clustering Fit

We first fit k-means clustering solutions across a range of cluster counts using the `easy_km_fit()` function from the `MKT4320BGSU` package.

Usage:

- `easy_km_fit(data, vars, k_range = 1:10, standardize = TRUE, nstart = 25, iter.max = 100, B = 20, seed = 4320)`
- where:
  - `data` is a data frame containing the full dataset.

- `vars` is a character vector of numeric variable names used for clustering.
- `k_range` is an integer vector of cluster counts to evaluate (default = 1:10; allowed values 1–20).
- `standardize` is logical; if TRUE (default), clustering variables are standardized before fitting k-means.
- `nstart` is an integer; number of random starts for each k-means solution (default = 25).
- `iter.max` is an integer; maximum number of iterations allowed for each k-means run (default = 100).
- `B` is an integer; number of Monte Carlo bootstrap samples used to compute the Gap statistic (default = 20).
- `seed` is an integer; random seed for reproducible results (default = 4320).

We now fit the same variables we used above in hierarchical clustering. As before, the results should be saved to an object. The `$diag` diagnostics table saved in the object provides the necessary results to help decide on a (potential) final solution. The diagnostics table summarizes multiple stopping rules and balance checks across different values of k. No single statistic should be used in isolation when choosing the number of clusters. You can also view the `$size_prop` table to see the cluster sizes for all potential solutions.

```
km_vars <- c("quality", "price", "healthy", "variety", "speed")
km_fit <- easy_km_fit(data = ffseg, vars = km_vars, k_range = 2:8)
```

```
Clustering k = 1,2,..., K.max (= 8): .. done
Bootstrapping, b = 1,2,..., B (= 20)  [one "." per sample]:
.................... 20
```

```
km_fit$diag
```

K-Means Cluster Diagnostics (Standardized)

Clusters	Silhouette	Gap.Stat	Small.Prop	Large.Prop	CV
2	0.2120	0.6307*	0.4761	0.5239 ◊	0.068 •
3	0.1973	0.6150	0.2739	0.4096	0.208 •
4	0.1610	0.5953	0.2168	0.2806	0.105 •
5	0.1679	0.5855	0.1569	0.2340	0.178 •
6	0.1719	0.5827	0.1303	0.2354	0.224 •
7	0.1803	0.5741	0.1184	0.1835	0.195 •

K-Means Cluster Diagnostics (Standardized)					
Clusters	**Silhouette**	**Gap.Stat**	**Small.Prop**	**Large.Prop**	**CV**
8	0.1804	0.5747	0.0851	0.1582	0.185 •

* 1-SE Gap rule.

^ Smallest cluster < 5% of sample.

◊ Largest cluster > 50% of sample.

• CV < 0.5 well balanced; •• moderately imbalanced.

Silhouette is average silhouette width (higher is better); defined for k >= 2.

```
km_fit$size_prop
```

Cluster Size Proportions by Solution					
Solution	**Cluster 1**	**Cluster 2**	**Cluster 3**	**Cluster 4**	**Cluster 5**
2-cluster solution	0.4761	0.5239			
3-cluster solution	0.3165	0.4096	0.2739		
4-cluster solution	0.2168	0.2487	0.2540	0.2806	
5-cluster solution	0.2128	0.1569	0.2340	0.2287	0.16
6-cluster solution	0.1423	0.1503	0.1303	0.1715	0.17
7-cluster solution	0.1263	0.1184	0.1516	0.1835	0.17
8-cluster solution	0.1330	0.1436	0.1396	0.1582	0.11

10.5.3 Final K-Means Clustering Solution

After selecting a value for k, we finalize the k-means solution using the `easy_km_final()` function from the `MKT4320BGSU` package.

Usage:

- `easy_km_final(fit, data, k, cluster_col = "cluster", conf_level = 0.95, auto_print = TRUE)`
- where:
  - `fit` is an object returned by `easy_km_fit()`.
  - `data` is the original full dataset used in `easy_km_fit()`.
  - `k` is an integer; number of clusters to extract.

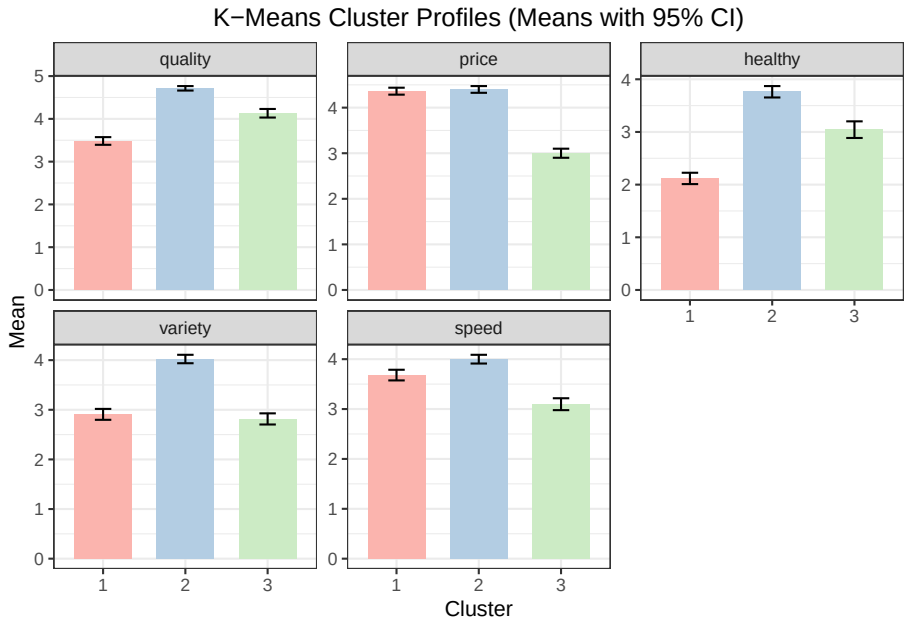
- `cluster_col` is a character string; name of the cluster column to append to data (default = "cluster").
- `conf_level` is numeric; confidence level for CI error bars (default = 0.95).
- `auto_print` is logical; if TRUE (default), prints selected outputs and displays the plot when the function is run.

Using the example from above with a 3-cluster solution, the cluster profile table reports mean values of the segmentation variables for each cluster. These means should be interpreted in relative terms across clusters. The cluster plot visualizes those means.

```
km_final <- easy_km_final(fit = km_fit, data = ffseg, k = 3,
                          cluster_col = "km_cluster", auto_print = FALSE)
km_final$props
```

```
  Cluster   N      Prop
1       1 238 0.3164894
2       2 308 0.4095745
3       3 206 0.2739362
```

```
km_final$plot
```



```
km_final$profile
```

```
  Cluster  quality    price  healthy  variety    speed
1       1 3.483193 4.361345 2.117647 2.907563 3.680672
2       2 4.714286 4.399351 3.762987 4.022727 4.000000
3       3 4.131068 3.000000 3.043689 2.815534 3.097087
```

10.6 Describing and Labeling *k*-Means Clusters

As with hierarchical clustering, segmentation variables alone rarely provide enough context to understand who the clusters represent. Additional variables are used to describe and label the clusters. As with hierarchical clustering, use the `easy_cluster_decribe()` function from the `MKT4320BGSU` package to automate this process. See Section 10.4

10.7 Comparing Clustering Approaches

Hierarchical and *k*-means clustering often produce similar but not identical segmentation results. Differences between solutions can be informative and may reveal alternative ways to think about the market.

In practice, analysts often:

- Use hierarchical clustering to explore the structure of the data
- Use k-means clustering to refine and stabilize the final solution

---

10.8 Chapter Summary

In this chapter, we:

- Introduced cluster analysis as a tool for marketing segmentation
- Applied hierarchical and k-means clustering to the `ffseg` dataset
- Used diagnostics to guide the choice of the number of clusters
- Described and interpreted clusters in a marketing context

10.9 What's Next

In the next chapter, we shift from grouping customers to summarizing and visualizing variables. Specifically, we will introduce Principal Components Analysis (PCA) and then extend it to PCA perceptual maps. PCA is a dimensionality-reduction technique that helps simplify complex datasets by transforming many correlated variables into a smaller set of components that capture the most important patterns in the data.

You will learn how PCA can be used to:

- Reduce a large set of variables into a smaller number of interpretable dimensions
- Identify underlying structures in consumer perceptions and evaluations
- Prepare data for visualization and communication

We will then use these components to create perceptual maps, which are widely used in marketing to:

- Visualize brand or product positions
- Understand competitive structure
- Support positioning and differentiation decisions

Chapter 11

PCA and Perceptual Maps

11.1 Introduction: Why PCA Matters in Marketing Analytics

Marketing datasets often contain many related variables that describe how consumers perceive brands, products, or services. When these attributes are highly correlated, interpretation becomes difficult and redundancy increases. Principal Components Analysis (PCA) is a dimension-reduction technique that helps uncover the underlying structure in such data.

In marketing analytics, PCA is commonly used to:

- Summarize brand image and positioning data
- Reduce large attribute batteries into interpretable dimensions
- Serve as the foundation for perceptual maps

In this chapter, we focus on applying PCA for **interpretation and insight**, not mathematical derivation. We will:

1. Fit a PCA model and evaluate diagnostics
2. Choose an appropriate number of components
3. Interpret component loadings
4. Use PCA results to construct perceptual maps

As a high-level overview, PCA transforms a set of correlated variables into a smaller set of new variables called *principal components*. Each component is a weighted combination of the original variables.

Key ideas:

- Components are ordered by how much variance they explain
- The first component explains the most variance, the second explains the most remaining variance, and so on
- Components are uncorrelated with one another

From a marketing perspective, PCA helps answer: "What are the main dimensions consumers use to differentiate brands?"

11.2 The `greekbrands` Dataset

This chapter uses the `greekbrands` dataset, which contains simulated attribute ratings and brand preference data for ten fictional technology brands. Each observation corresponds to a respondent-brand evaluation.

The dataset includes:

- A brand identifier
- Multiple numeric attribute ratings describing brand perceptions

This type of brand image data is well suited for PCA because many of the attributes tend to be correlated and may reflect a smaller number of latent dimensions.

11.3 Preparing for PCA in a Marketing Context

Before fitting a PCA model, it is important to:

- Use only numeric perceptual attributes
- Exclude identifiers (e.g., brand names, respondent IDs)
- Consider whether PCA should be run at the individual or brand level

For perceptual mapping, brand-level aggregation is typically preferred so that brands (not respondents) appear as points in the map.

11.4 PCA Modeling

11.4.1 Fitting an Initial PCA Model

We begin with an initial PCA fit to evaluate how many components should be retained. This step focuses on diagnostics rather than interpretation. Although PCA is not difficult in base R using the `prcomp()` function, we'll use the `easy_pca_fit()` function from the `MKT4320BGSU` to automate the process for both fitting and a separate function for the final model.

Usage:

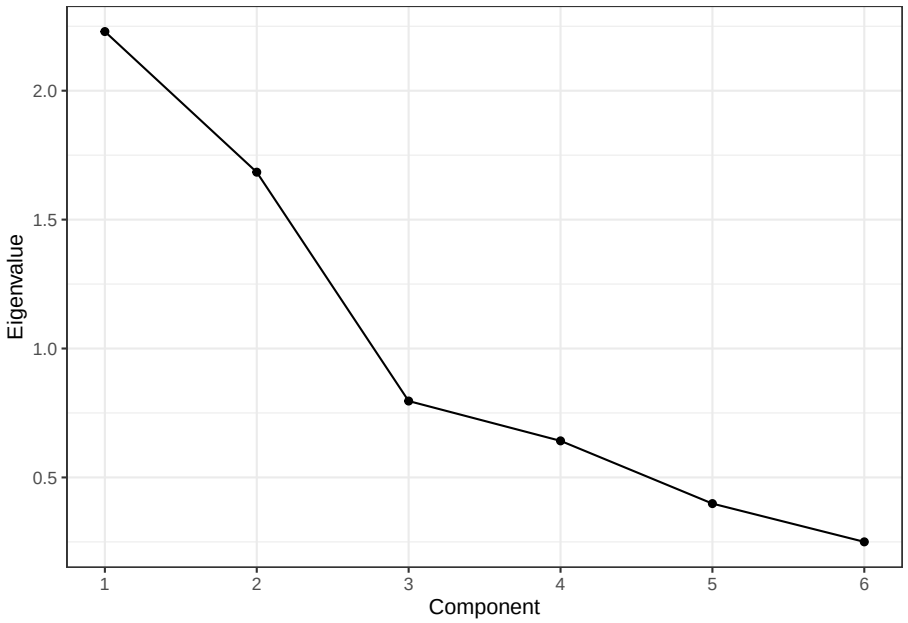
- `easy_pca_fit(data, vars, group = NULL, ft = TRUE)`
- where:
  - `data` is a data frame containing the full dataset.
  - `vars` is a character vector of variable names to use in PCA (required). All variables must be numeric.
  - `group` is an optional character sting of a single variable name to aggregate by before PCA.
  - `ft` is logical; if TRUE, return $table as a flextable (default = TRUE).

In the example below, we do not use the `group` option.

```
attr_vars <- c("perform", "leader", "fun", "serious", "bargain", "value")
pca_fit <- easy_pca_fit(data = greekbrands,  vars = attr_vars, ft=TRUE)
pca_fit$table
```

Component	Eigenvalue	Difference	Proportion	Cumulative
1	2.2293	0.5454	0.3716	0.3716
2	1.6839	0.8876	0.2806	0.6522
3	0.7963	0.1545	0.1327	0.7849
4	0.6418	0.2433	0.1070	0.8919
5	0.3985	0.1484	0.0664	0.9583
6	0.2501		0.0417	1.0000

```
pca_fit$plot
```



The eigenvalue table and scree plot summarize how much variance each component explains. Important columns in the eigenvalue table include:

- **Eigenvalue**: total variance explained by each component
- **Proportion**: share of total variance explained
- **Cumulative**: cumulative proportion of variance explained

Common decision rules:

- Retain components with eigenvalues greater than 1
- Look for an "elbow" where additional components add little explanatory power
- Aim for a solution that balances parsimony and interpretability

There is no single correct answer. Component retention should be guided by marketing judgment as well as statistics.

11.4.2 Final PCA Solution

After deciding how many components to retain, we refit the PCA model and examine the loadings using the `easy_pca_final()` function from the `MKT4320BGSU` package.

Usage:

- `easy_pca_final(data, vars, comp, group = NULL, ft = TRUE)`
- where:
  - `data` is a data frame containing the full dataset.
  - `vars` is a character vector of variable names to use in PCA (required). All variables must be numeric.
  - `comp` is an integer representing the number of components to retain.
  - `group` is an optional character sting of a single variable name to aggregate by before PCA.
  - `ft` is logical; if TRUE, return $table as a flextable (default = TRUE).

In the example below, we again choose not to use the `group` option.

```
pca_final <- easy_pca_final(data = greekbrands, vars = attr_vars,
                            comp = 2, ft=TRUE)
pca_final$rotated
```

Varimax-Rotated PCA Loadings

Variable	Comp_1	Comp_2	Unexplained
perform	0.7243	-0.0750	0.4697
leader	0.8404	-0.0486	0.2914
fun	-0.5475	0.1496	0.6778
serious	0.7849	0.0434	0.3821
bargain	-0.0189	-0.9294	0.1359
value	0.0698	-0.9302	0.1298

We focus on the **varimax-rotated** loadings because they are easier to interpret. A loading represents the relationship between an original attribute and a component:

- Larger absolute values indicate stronger relationships
- Attributes with high loadings on the same component tend to reflect a common underlying dimension

When interpreting loadings:

- Look for patterns across attributes
- Identify which attributes define each component
- Assign descriptive, managerially meaningful names to components

For example:

- A component with high loadings on *perform*, *leader*, and *serious* might be labeled **Performance**
- A component with high loadings on *bargain* and *value* might be labeled **Value Orientation**

11.5 From PCA to Perceptual Maps

PCA components can be used as axes in perceptual maps. Each brand's position on a component reflects how strongly it scores on that underlying dimension.

Perceptual maps translate statistical results into a visual format that is easy to communicate to managers and decision-makers.

11.6 Attribute-Based Perceptual Maps Using PCA

11.6.1 Creating PCA-Based Maps

We now use the retained PCA solution to create perceptual maps. We use the `easy_pca_maps()` function from the `MKT4320BGSU` package to automate the process.

Usage:

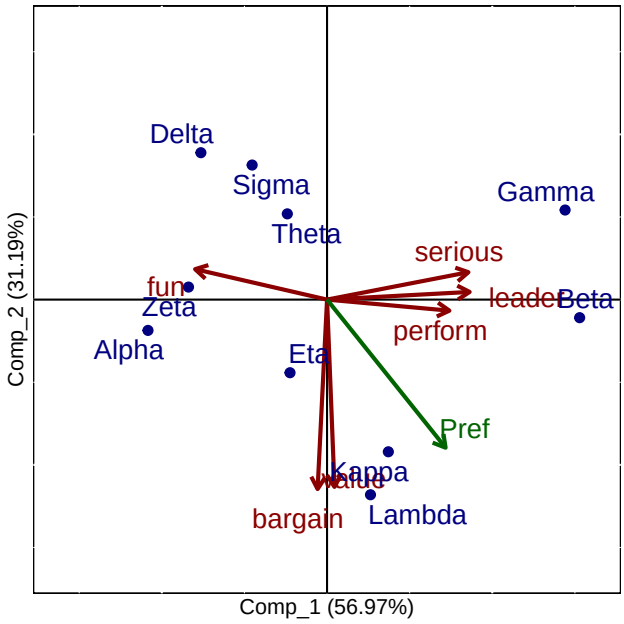
- `easy_pca_maps(data, vars, group, comp, pref = NULL, rotate = TRUE,`
  `arrow_scale = 0.75,label_pad = 0.04)`
- where:
  - `data` is a data frame containing individual-level observations.
  - `vars` is a character vector of numeric attribute variable names used in PCA.
  - `group` is a single string specifying the grouping variable (e.g., brand or product name).
  - `comp` is an integer specifying the number of components to retain (must be ≥ 2).

- `pref` is an optional single string specifying a numeric preference variable name to produce a joint space map with a preference vector (if data is available)
- `rotate` is logical; if TRUE (default), apply varimax rotation to the retained component space before creating perceptual maps.
- `arrow_scale` is numeric in (0, 1]; scales arrow lengths relative to the object range (default = 0.75).
- `label_pad` is numeric; distance (as a proportion of the axis range) used to push attribute arrow labels beyond arrow tips (default = 0.04) for easier viewing of the map.

We'll use the same variables as before, but add in the required `group` and an optional `pref` to create a joint space map.

```
pca_maps <- easy_pca_maps(data = greekbrands, vars = attr_vars,
                          group = "brand", pref = "pref", comp = 2)
pca_maps$plots
```

```
$Comp_1_vs_Comp_2
```



The map displays:

- Brands as points
- Attribute vectors showing how attributes align with the components

Key interpretation guidelines:

- Brands close together are perceived similarly
- Brands far apart are perceived differently
- Attribute vectors indicate the direction of increasing attribute values
- Brands in the direction of an attribute vector score higher on that attribute
- Attribute vectors more parallel with the preference vector (if available) a stronger drivers of preference

Distances are relative and should be interpreted qualitatively rather than precisely.

11.7 Managerial Interpretation and Strategic Insights

PCA-based perceptual maps can help managers:

- Identify direct competitors
- Detect market clusters and white space
- Evaluate whether a brand's positioning matches strategic intent

These insights can inform:

- Positioning statements
- Advertising and messaging strategy
- Product reformulation decisions

Common Pitfalls and Best Practices

- Do not over-interpret small loadings
- Avoid retaining too many components
- Remember that PCA alone reflects perceptions, not preferences
- Always explain components in clear, non-technical language

11.8 Chapter Summary

In this chapter, we:

- Used PCA to reduce and interpret brand perception data
- Applied diagnostics to choose the number of components
- Interpreted rotated component loadings
- Created perceptual maps to visualize brand positioning

PCA is a powerful bridge between data analysis and strategic insight in marketing analytics.

11.9 What's Next

In the next chapter, we shift from describing perceptions to testing causal impact.Specifically, we will study A/B testing and uplift modeling, which are tools used to answer questions such as:

- Does a new message, offer, or design actually change behavior?
- How large is the effect of a treatment compared to a control?
- Are some customers more responsive to an intervention than others?

Where PCA and perceptual maps help us understand how consumers see brands, A/B testing helps us evaluate what actions work, and uplift modeling helps us determine for whom they work best. These methods are central to modern data-driven marketing in areas such as digital advertising, pricing experiments, promotions, and personalization.

Chapter 12

A/B Testing and Uplift Modeling

12.1 Introduction: From Average Effects to Targeted Marketing

A/B testing is one of the most widely used tools in marketing analytics. Whether testing email subject lines, promotional offers, website layouts, or pricing messages, marketers frequently rely on randomized experiments to measure causal effects.

In this chapter, we begin with **average treatment effects (ATEs)**, which is the traditional goal of A/B testing,and then move beyond averages to **uplift modeling**, which focuses on identifying *who* is most likely to be influenced by a marketing intervention.

Throughout the chapter, we use data from an email marketing experiment contained in the `email.camp.w` dataset.

12.2 The Email Campaign Experiment

The dataset `email.camp.w` comes from a randomized email campaign experiment. Customers were randomly assigned to receive either:

- a **promotional email** (treatment group), or
- **no promotional email** (control group).

The primary outcome of interest is whether the customer responded (e.g., clicked or converted). In addition, the dataset contains several customer characteristics (covariates) such as demographics and prior behavior.

Because treatment assignment was randomized, differences in outcomes between the two groups can be interpreted causally.

12.3 Checking the Randomization Assumption

12.3.1 Why Balance Checks Matter

Randomization ensures that treatment and control groups are similar *on average*. However, especially in applied settings, it is good practice to verify that observable covariates are balanced across groups.

Large imbalances may signal problems such as implementation errors or data issues.

12.3.2 Randomization Check Using `rand_check()`

The `rand_check()` function from the `MKT4320BGSU` package compares the distribution of selected covariates across treatment groups and automatically applies appropriate statistical tests.

Usage:

- `rand_check(data, treatment, covariates, ft = TRUE, digits = 3)`
- where:
  - `data` is a data frame containing the treatment indicator and covariates.
  - `treatment` is a character string giving the name of the treatment variable. Must identify two or more groups.
  - `covariates` is a character vector of covariate names to include in the randomization check.
  - `ft` is logical; if TRUE (default), return results as a flextable. If FALSE, return a data frame.
  - `digits` is an integer; number of decimal places to display in the output (default = 3).

An example is provided below. When reviewing the output, focus on:

- **Scaled mean differences**: values close to zero indicate good balance.
- **p-values**: large p-values suggest no systematic differences.

Well-balanced covariates support the validity of the experiment.

```
rand_check(data = email.camp.w, treatment = "promotion",
  covariates = c("recency", "history", "womens", "zip"),
  ft = TRUE)
```

Variable	Mean		SD	Scaled Mean Difference	p-value
	Treatment	Control			
recency	5.810	5.725	3.504	0.024	0.227
history	245.995	242.539	253.384	0.014	0.495
womens	0.545	0.539	0.498	0.011	0.574
zip:Rural	0.143	0.148	0.353	-0.014	0.395
zip:Surburban	0.459	0.445	0.498	0.027	
zip:Urban	0.398	0.406	0.490	-0.017	

12.4 Estimating the Average Treatment Effect (ATE)

12.4.1 What Is the ATE?

The **average treatment effect** measures the average impact of the promotion across all customers. In an email campaign, this answers the question: Did sending the promotion increase response rates overall?

12.4.2 ATE via Regression

For binary outcomes, a linear regression with a treatment indicator is equivalent to a difference-in-means estimator. When the outcome is binary, this is known as a **linear probability model (LPM)**.

12.4.3 Using `easy_ab_ate()`

We estimate the ATE using a regression model that includes the treatment indicator and optionally adjusts for covariates by using the `easy_ab_ate()` function from the `MKT4320BGSU` package.

Usage:

- `easy_ab_ate(model, treatment, ft = TRUE)`
- where:
  - `model` is a fitted linear regression model of class lm. This model should include the treatment variable and (optionally) covariates.
  - `treatment` is a character string with the name of the treatment variable (in quotes).
  - `ft` is logical; if TRUE (default) return a flextable. If FALSE, print full regression results to the console.

Note that to use this function, you must first create a linear regression model using the `lm()` function. The model should have a response variable as the dependent variable, the treatment variable as an independent variable, and any additional covariates as additional independent variables. The results should be saved to an object. For example:

- `object <- lm(response ~ treatment + cov_1 + cov_2 + ... + cov_k, data = data)`

```
m_ab_visit <- lm(visit ~ promotion + recency + history + womens + zip,
                 data = email.camp.w)
easy_ab_ate(model = m_ab_visit, treatment = "promotion", ft = TRUE)
```

	Without Covariates		With Covariates	
Characteristic	**Beta**	**p-value**	**Beta**	**p-value**
(Intercept)	0.106	<0.001	0.151	<0.001
promotion	0.049	<0.001	0.050	<0.001
recency			-0.006	<0.001
history			0.000	<0.001
womens			0.046	<0.001
zip				

	Without Covariates		With Covariates	
Characteristic	**Beta**	**p-value**	**Beta**	**p-value**
Rural			—	
Surburban			-0.053	<0.001
Urban			-0.065	<0.001
p-value	<0.001		<0.001	
R^2	0.005		0.024	

```
m_ab_spend <- lm(spend ~ promotion + recency + history + womens + zip,
                 data = email.camp.w)
easy_ab_ate(model = m_ab_spend, treatment = "promotion", ft = TRUE)
```

	Without Covariates		With Covariates	
Characteristic	**Beta**	**p-value**	**Beta**	**p-value**
(Intercept)	0.651	<0.001	1.265	0.011
promotion	0.436	0.108	0.450	0.097
recency			-0.081	0.042
history			0.000	0.703
womens			0.049	0.858
zip				
Rural			—	
Surburban			-0.596	0.144
Urban			0.098	0.814
p-value	0.11		0.032	
R^2	0.000		0.001	

The table compares two models:

- **Without covariates**: a pure A/B comparison.
- **With covariates**: a regression-adjusted estimate.

The treatment coefficient represents the average change in response probability caused by the promotion in isolation (in the without covariates column) or when controlling for other variables (in the with covariates column).

12.4.4 Why Average Effects Are Not Enough

While the ATE is useful, it hides important heterogeneity:

- Some customers may respond very positively.
- Others may be unaffected or even respond negatively.

From a managerial perspective, sending promotions to everyone may be inefficient or costly. This motivates **uplift modeling**, which focuses on targeting customers who are most likely to be influenced.

12.5 Introduction to Uplift Modeling

12.5.1 What Is Uplift?

Uplift measures the *incremental* effect of treatment on an individual. That is, how much more likely is this customer to respond *because* they received the promotion?

This differs from standard prediction, which focuses on response likelihood regardless of treatment.

12.5.2 Two-Model (Indirect) Approach

The two-model approach estimates:

- one model estimated on treated customers, and
- one model estimated on control customers.

The difference between their predicted outcomes for a given customer is the estimated uplift.

12.6 Estimating Uplift with `easy_uplift()`

12.6.1 Model Specification

The outcome variable can be either continuous, like amount spent (outcome) after a promotion (treatment), or it can be binary, like if they visited or not (outcome) after a promotion (treatment). To perform a uplift modeling using regression, we will use the `easy_uplift()` function from the `MKT4320BGSU` package. This function performs uplift modeling based on either logistic regression (for binary outcomes) or linear regression (for continuous outcomes). The function uses the two-model, indirect modeling approach.

Usage:

- `easy_uplift(model, treatment, newdata = NULL, bins = 10, aspect_ratio = NULL)`
- where:
  - `model` is a fitted regression model of class glm (binary logit) or lm.
  - `treatment` is a character string giving the name of the treatment variable. The variable must have exactly two levels and be coded as (0/1), logical, or ("Yes", "No").
  - `newdata` is an optional data frame on which to compute uplift (e.g., holdout or test data). If NULL, uplift is computed on the model data.
  - `bins` is an integer; number of groups used for the uplift tables and plots. Must be between 5 and 20. Default is 10.
  - `aspect_ratio` is an optional numeric aspect ratio applied to all plots. Default is NULL.

In order to use the function, we must first create our base model:

- The base model is usually a model with no interactions included, along with the treatment variable. But if known interactions are to be used, the base model can include the interactions also.
- The base model must contain the treatment variable.

Base model examples:

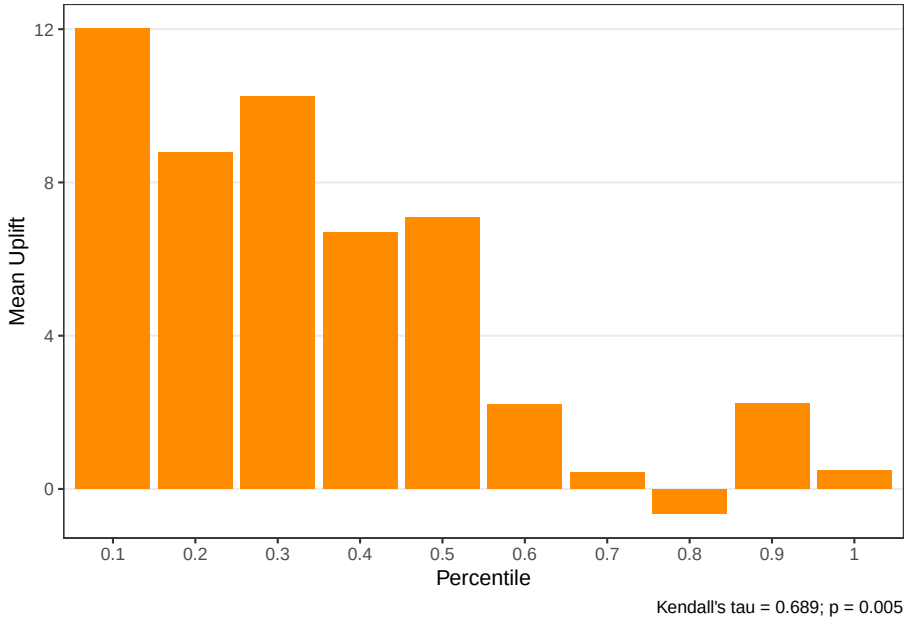
```
email_visit <- glm(visit ~ promotion + recency + history + zip + womens,
                   data=email.camp.w, family="binomial")
email_spend <- lm(spend ~ promotion + recency + history + zip + womens,
                  data=email.camp.w)
```

Once the base model is created, we are aready to use the `easy_uplift()` function:

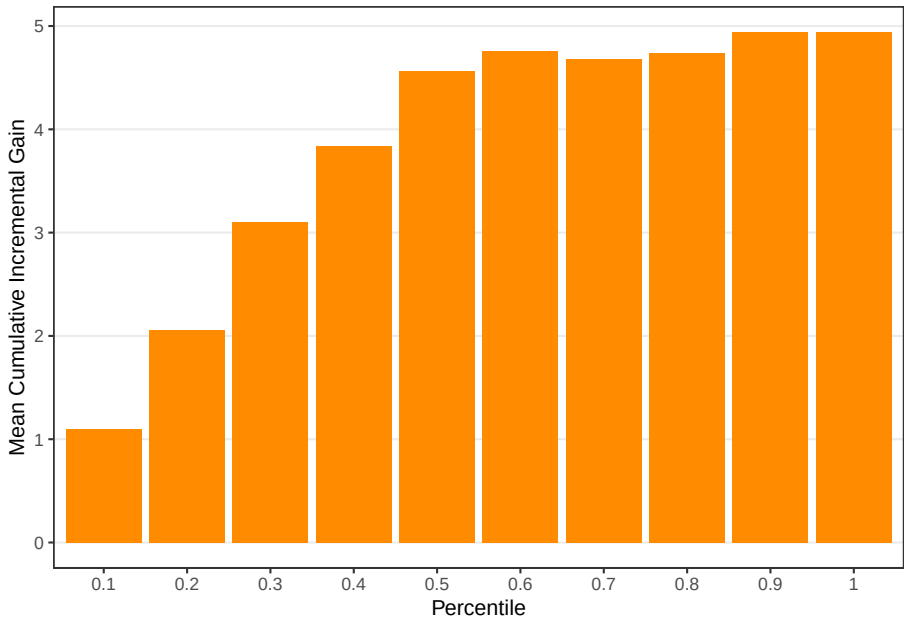
```
visit_uplift <- easy_uplift(model = email_visit, treatment = "promotion")
visit_uplift$plots
```

```
$qini
```


```
$uplift
```

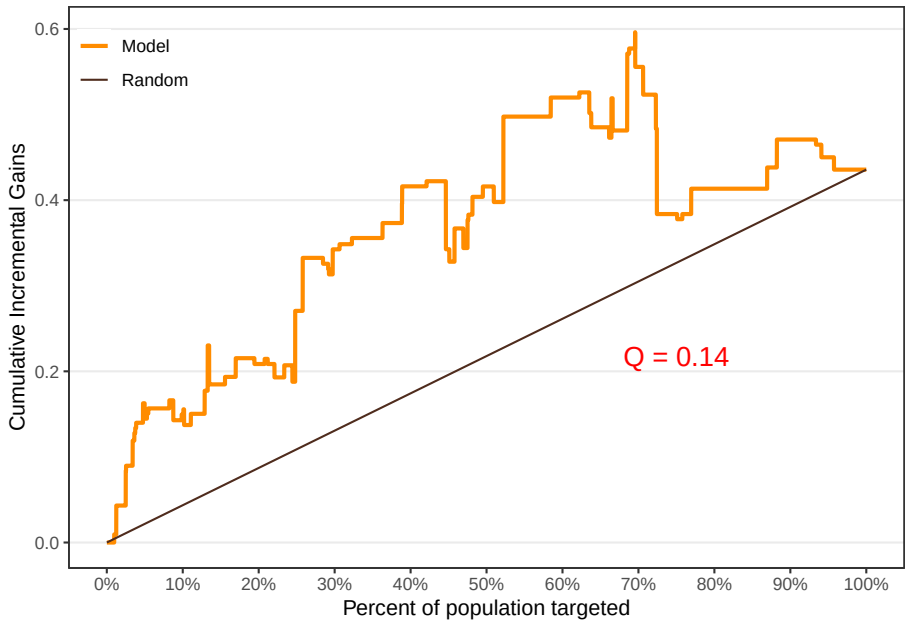

12
8
4
0
Mean Uplift
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Percentile
Kendall's tau = 0.689; p = 0.005

```
$c.gain
```

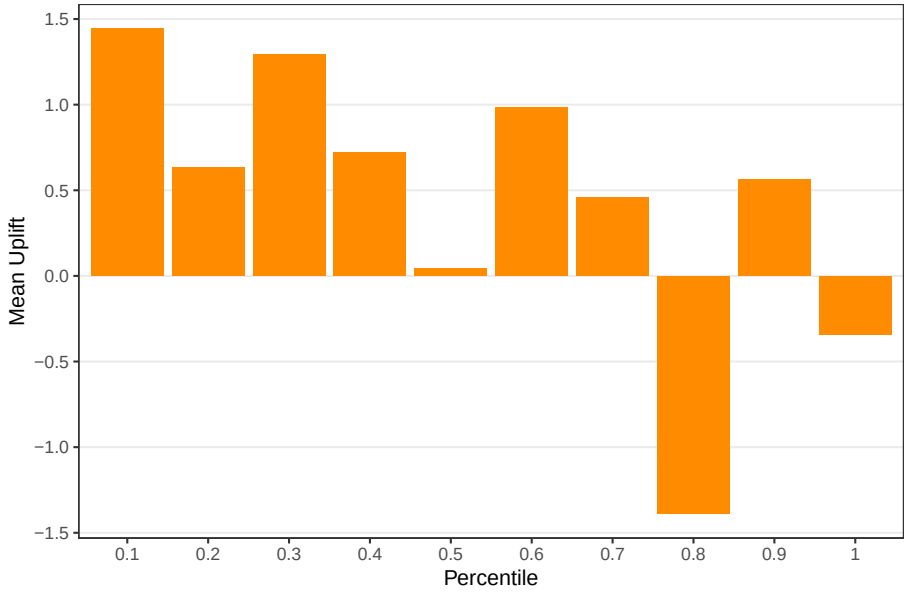

5
4
3
2
1
0
Mean Cumulative Incremental Gain
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Percentile

```
spend_uplift <- easy_uplift(model = email_spend, treatment = "promotion")
spend_uplift$plots
```

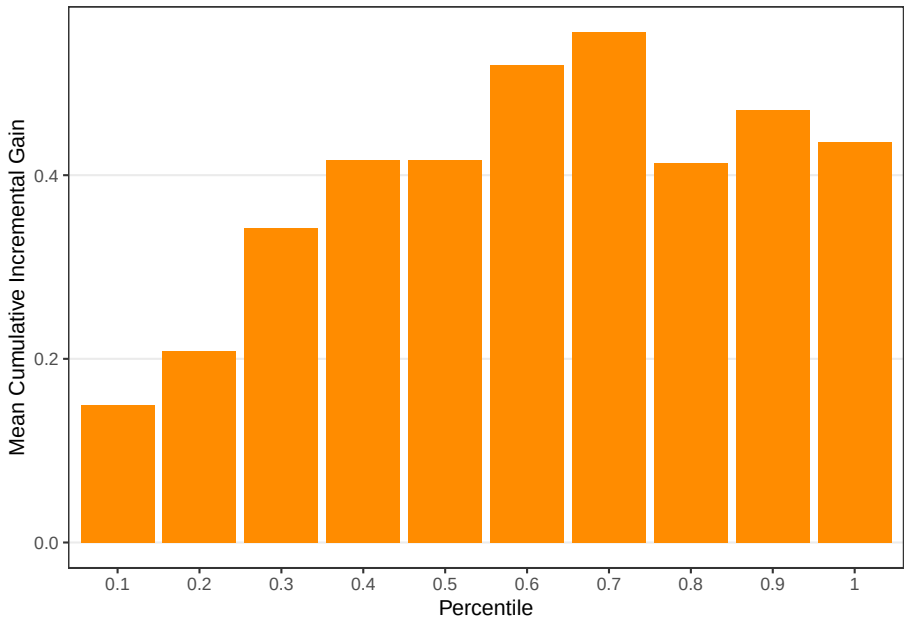
```
$qini
```




```
$uplift
```


1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
Mean Uplift
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Percentile
Kendall's tau = 0.556; p = 0.029

`$c.gain`


0.4
0.2
0.0
Mean Cumulative Incremental Gain
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Percentile

12.6.2 Interpreting Core Outputs

The uplift object includes:

- predicted individual-level lift appended to original data (`$all`),
- uplift table by percentile group (`$group`),
- diagnostic plots, including:
  - uplift by group (`$plots$uplift`),
  - cumulative gain (`$plots$c.gain`),
  - Qini curve (`$plots$qini`).

Customers in the top-ranked groups should exhibit the largest incremental gains from treatment.

12.7 Diagnosing Uplift with Lift Plots

12.7.1 Why Lift Diagnostics Matter

Lift diagnostics help explain *why* uplift varies across customers and which covariates drive heterogeneity.

12.7.2 Using `easy_liftplots()`

We use the `easy_liftplots()` function from the `MKT4320BGSU` package to easily create lift plots.

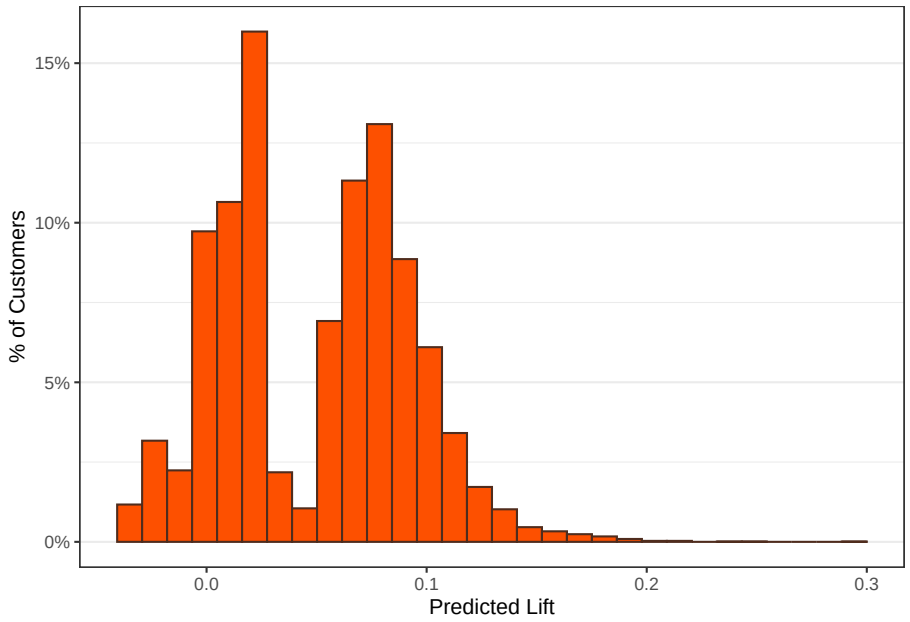
Usage:

- `easy_liftplots(x, vars = "all", pairs = NULL, ar = NULL, ci = 0.95,`
  `bins = 30, numeric_bins = 5, by_numeric_bins = 3, grid = TRUE,`
  `top = NULL, ft = TRUE)`
- where:
  - `x` is an object returned by `easy_uplift()` (must include x*all* and x*cov**ariates* or x*spec* covariates).
  - `vars` is a character vector of covariate names to plot. Default is "all" (uses x*covariates*/*x*spec$covariates).

- `pairs` is an optional list of length-2 character vectors specifying interaction-style plots to create, e.g., list(c("recency","zip"), c("gender","income")).
- `ar` is an optional aspect ratio passed to theme(aspect.ratio = ar). Default is NULL.
- `ci` affects the error-bar style. Use 0 for ±1 SD error bars, or one of c(0.90, 0.95, 0.975, 0.99) for normal-approximation confidence intervals. Default is 0.95.
- `bins` is an integer; number of bins for the histogram. Default is 30.
- `numeric_bins` is an integer; number of quantile bins for numeric covariates. Default is 5.
- `by_numeric_bins` is an integer; number of quantile bins to use for the second variable in a pair when it is numeric. Default is 3.
- `grid` is logical; if TRUE, also return paginated cowplot grids of plots. Default is TRUE.
- `top` is an optional integer. If provided, only the top `top` covariates (by score_wmae) are included in plots_main/pages_main. Rankings are still computed for all covariates.
- `ft` is logical; if TRUE (default), return ranking tables as flextable objects.

A histogram is always produced, which shows the distribution of predicted uplift across customers. It is saved as `$hist`.

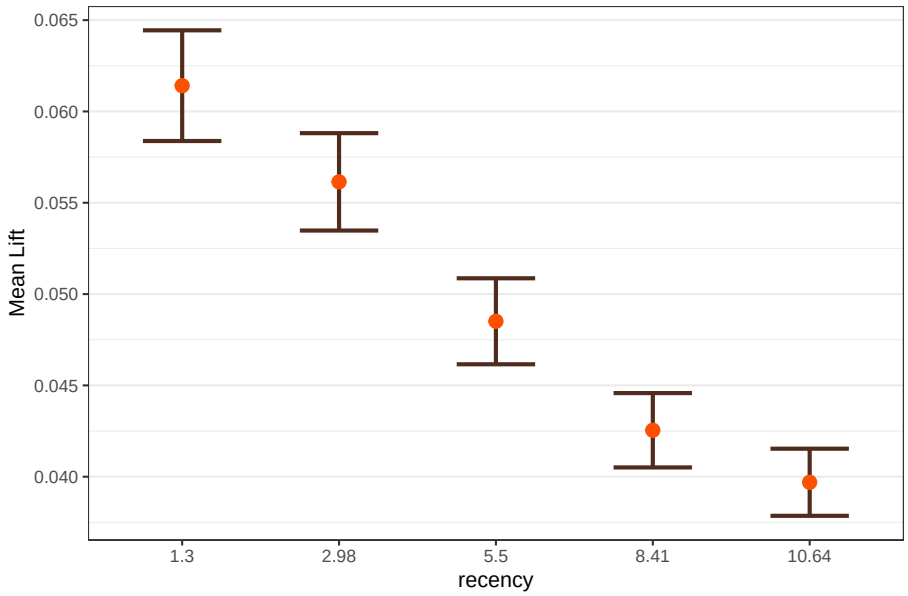
```
lift_out <- easy_liftplots(visit_uplift, vars = "all", ci = 0.99)
lift_out$hist
```



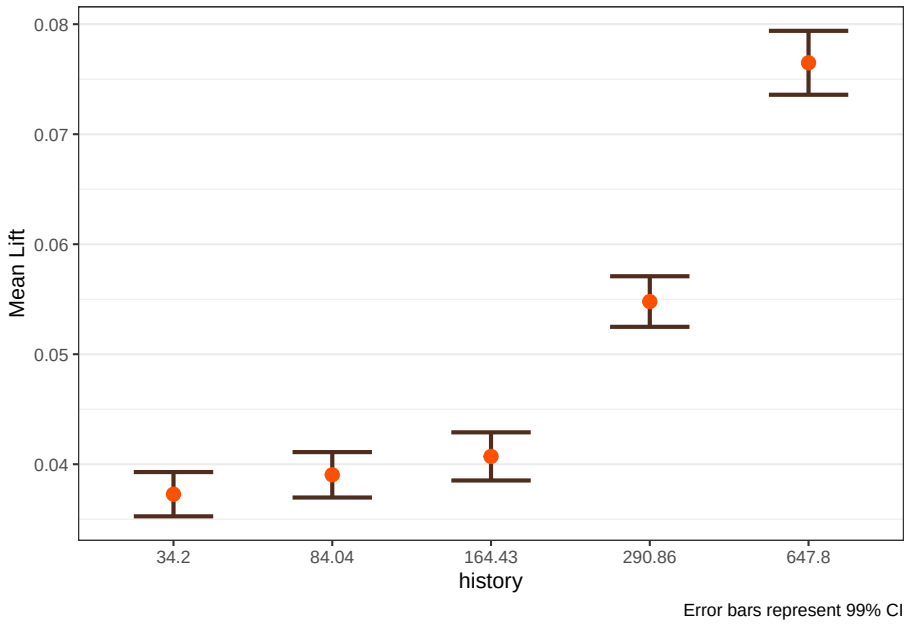
The main lift plots are saved in `$plots_main`. Lift-by-covariate plots display how average uplift varies across customer segments.

```
lift_out$plots_main
```

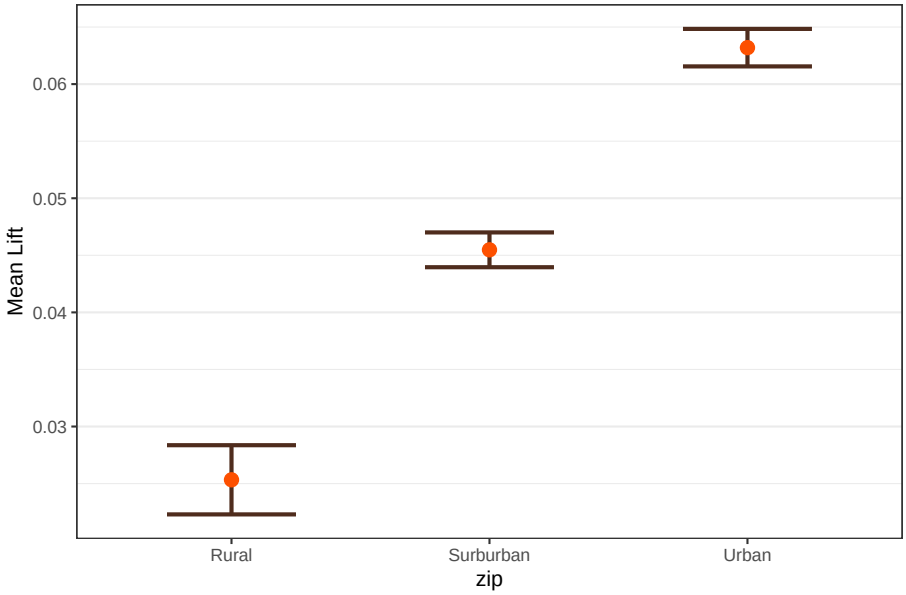
```
$recency
```


0.065
0.060
0.055
0.050
0.045
0.040
Mean Lift
1.3
2.98
5.5
8.41
10.64
recency
Error bars represent 99% CI

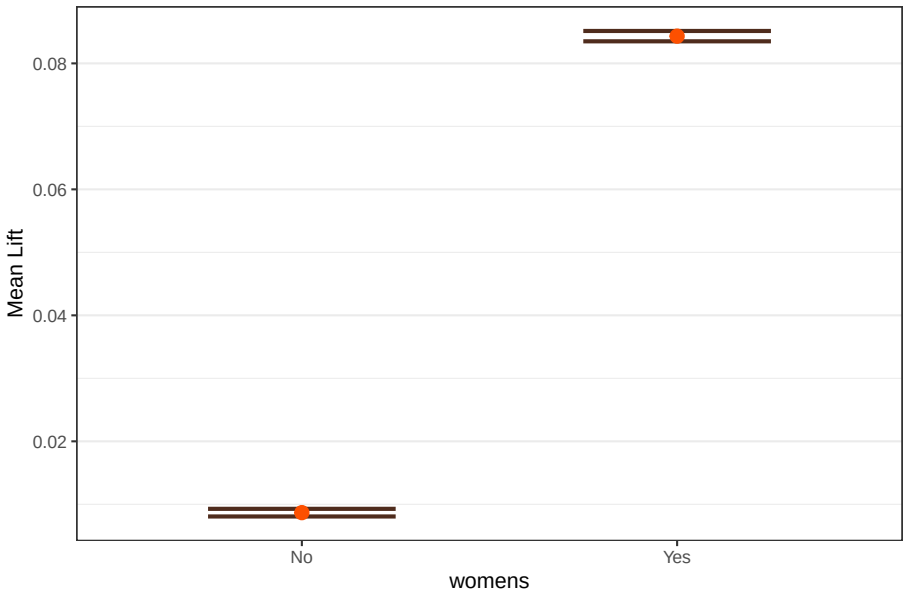
```
$history
```


0.08
0.07
0.06
0.05
0.04
Mean Lift
34.2
84.04
164.43
290.86
647.8
history
Error bars represent 99% CI

```
$zip
```


0.06
0.05
0.04
0.03
Mean Lift
Rural
Surburban
Urban
zip
Error bars represent 99% CI

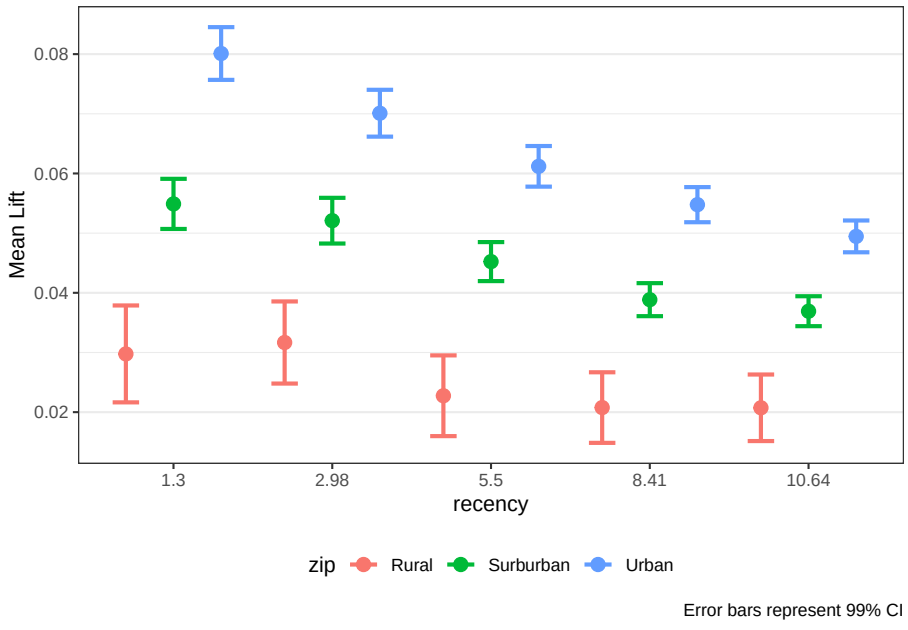
```
$womens
```


0.08
0.06
0.04
0.02
Mean Lift
No
Yes
womens
Error bars represent 99% CI

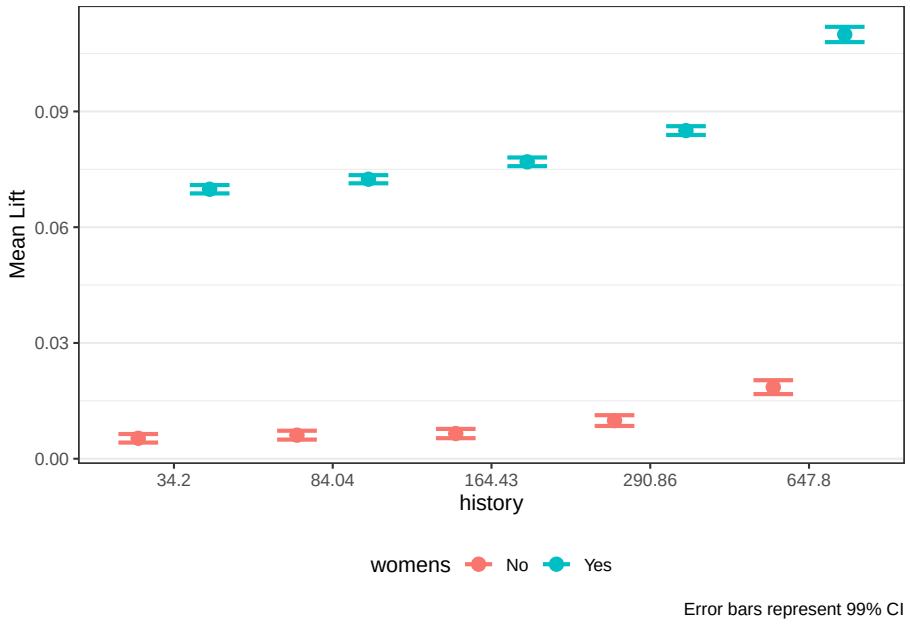
The `pairs` option is exteremely valuable when known interactions are includedin the model. The option can also be useful to help identify if an interaction may be warranted. If `pairs` is provided, the plots are saved in `plots_pairs`.

```
lift_out_pairs <- easy_liftplots(visit_uplift, vars = "all",
                                 pairs = list(c("recency","zip"),
                                              c("history","womens")),
                                 ci = 0.99)
lift_out_pairs$plots_pairs
```

```
$`recency × zip`
```




```
$`history × womens`
```



12.8 From Analysis to Action

Uplift modeling enables smarter targeting strategies:

- Send promotions only to customers with positive uplift.
- Prioritize customers in the highest uplift deciles.
- Avoid over-targeting customers unlikely to respond.

These strategies can improve campaign profitability and customer experience.

12.9 Summary

In this chapter, you learned how to:

- validate randomization in A/B tests,
- estimate average treatment effects,

- move beyond averages using uplift modeling,
- interpret uplift diagnostics for targeting decisions.

A/B testing answers *whether* a campaign works. Uplift modeling answers *for whom* it works.

12.10 What's Next

In many real-world marketing problems, managers face a different challenge: customers are not choosing between respond and not respond, but among multiple competing alternatives. Examples include:

- Which brand a customer purchases
- Which product variant is selected
- Which service tier is chosen

In the next chapter, we introduce standard multinomial logistic regression, a workhorse model for analyzing and predicting choice among more than two options. You will learn how to:

- Model customer choice across multiple alternatives
- Interpret coefficients and predicted choice probabilities
- Evaluate model fit and classification performance
- Use multinomial logit models for applied marketing decisions

This next step shifts our focus from experimental treatment effects to choice modeling, setting the foundation for more advanced models of consumer decision-making used throughout marketing analytics and research.

Chapter 13

Standard Multinomial Logit Models

13.1 Introduction to Multinomial Choice in Marketing

Many marketing decisions involve choices among more than two discrete alternatives. Consumers may choose among competing brands, subscription plans, service providers, or product variants. When the outcome variable has more than two unordered categories, linear regression and binary logistic regression are no longer appropriate.

The **standard multinomial logit (MNL)** model is the most common baseline model for analyzing and predicting such outcomes. In marketing analytics, it is widely used for brand choice, product selection, and competitive response analysis. The focus of this chapter is on **applied interpretation** rather than mathematical derivation.

13.2 The `bfast` Dataset

In this chapter, we use the `bfast` dataset, which contains data on breakfast food preferences. Each observation represents a consumer choice occasion in which one type of food was selected from a competitive set.

The outcome variable records the chosen type, while predictor variables capture marketing mix and consumer characteristics that may influence choice. Our

core marketing question is: Which factors increase or decrease the probability that a consumer chooses a particular fast-food brand?

13.3 Training and Test Samples

To evaluate predictive performance, we split the data into training and test samples. As with binary logistic regression, we use the `splitsample()` function from the `MKT4320BGSU` package. This function creates reproducible partitions and supports stratification on the outcome variable.

Usage:

- `splitsample(data, outcome = NULL, group = NULL, choice = NULL, alt = NULL, p = 0.75, seed = 4320)`
- where:
  - `data` is the data frame to split.
  - `outcome` is the outcome variable in quotes used for stratification. Required when group is `NULL`. Optional when group is provided. For standard MNL, it is required.
  - `group` is NOT USED FOR STANDARD MNL
  - `choice` is NOT USED FOR STANDARD MNL
  - `alt` is NOT USED FOR STANDARD MNL
  - `p` is the proportion of observations to place in the training set. Must be strictly between 0 and 1. Default is 0.75.
  - `seed` is the random seed for reproducibility. Default is 4320.

Below, we create are training and test samples. We also check the outcome variable in the two samples to ensure they are similar proportions in each.

```
sp <- splitsample(data = bfast, outcome = "bfast")
train <- sp$train
test  <- sp$test

proportions(table(train$bfast))
```

```
   Cereal       Bar   Oatmeal
0.3851964 0.2628399 0.3519637
```

```
proportions(table(test$bfast))
```

```
   Cereal       Bar   Oatmeal
0.3853211 0.2614679 0.3532110
```

13.4 Estimating a Standard Multinomial Logit Model

We estimate the standard multinomial logit model using `nnet::multinom()`. It is important to include `model = TRUE` so that model diagnostics and classification results can be computed later.

Using the `summary()` function in base R will provide the raw coefficients from the model. The estimated coefficients describe how each predictor affects the relative log-odds of choosing one product versus the reference product.

```
library(nnet)
mnl_fit <- multinom(bfast ~ gender + marital + lifestyle + age,
                    model = TRUE, data=train)
```

```
# weights:  18 (10 variable)
initial  value 727.281335
iter  10 value 579.014122
final  value 574.997631
converged
```

```
summary(mnl_fit)
```

```
Call:
multinom(formula = bfast ~ gender + marital + lifestyle + age,
    data = train, model = TRUE)

Coefficients:
        (Intercept)  genderMale maritalUnmarried lifestyleInactive         age
Bar       0.8832457 -0.21298963        0.6126977        -0.7865772 -0.02532866
Oatmeal  -4.4920408 -0.02262325       -0.3897362         0.3187473  0.07996475

Std. Errors:
```

```
        (Intercept) genderMale maritalUnmarried lifestyleInactive         age
Bar       0.3256994  0.2064320        0.2123832         0.2090460 0.006655803
Oatmeal   0.4596750  0.2094666        0.2366511         0.2156992 0.007755708

Residual Deviance: 1149.995
AIC: 1169.995
```

13.5 Evaluating Model Fit

Raw coefficients alone do not indicate whether a model performs well. We use `eval_std_mnl()` from the `MKT4320BGSU` package to compute model-fit statistics and diagnostics.

Usage:

- `eval_std_mnl(OBJ, exp = FALSE, digits = 4, ft = FALSE, newdata = NULL, label_model = "Model data", label_newdata = "New data", class_digits = 3)`
- where:
  - `model` is a fitted `multinom` model.
  - `exp` is logical; if TRUE, return relative risk ratios (exp(beta)). If FALSE, return log-odds coefficients (default = FALSE).
  - `digits` is an integer; number of decimals used to round coefficient and model-fit results (default = 4).
  - `ft` is logical; if TRUE, return coefficient and classification tables as flextable objects (default = FALSE).
  - `newdata` is an optional data frame for an additional classification matrix (e.g., a holdout or test set). If NULL, only the model-data classification is produced.
  - `label_model` is a character string; label for the model-data classification output (default = “Model data”).
  - `label_newdata` is a character string; label for the newdata classification output (default = “New data”).
  - `class_digits` is an integer; number of decimals used to round classification statistics (default = 3).

Key outputs include:

- A likelihood-ratio test comparing the fitted model to an intercept-only model

- McFadden's pseudo R-squared
- Classification accuracy and diagnostics

In applied marketing contexts, even modest pseudo R-squared values can indicate meaningful improvements over random choice.

```
mnl_eval <- eval_std_mnl(model = mnl_fit, newdata = test, ft = TRUE)
mnl_eval
```

LR chi2 (8) = 288.1568; p < 0.0001					
McFadden's Pseudo R-square = 0.2004					
y.level	term	logodds	std.error	statistic	p.value
Bar	(Intercept)	0.8832	0.3257	2.7118	0.0067
Bar	genderMale	-0.2130	0.2064	-1.0318	0.3022
Bar	maritalUnmarried	0.6127	0.2124	2.8849	0.0039
Bar	lifestyleInactive	-0.7866	0.2090	-3.7627	0.0002
Bar	age	-0.0253	0.0067	-3.8055	0.0001
Oatmeal	(Intercept)	-4.4920	0.4597	-9.7722	0.0000
Oatmeal	genderMale	-0.0226	0.2095	-0.1080	0.9140
Oatmeal	maritalUnmarried	-0.3897	0.2367	-1.6469	0.0996
Oatmeal	lifestyleInactive	0.3187	0.2157	1.4777	0.1395
Oatmeal	age	0.0800	0.0078	10.3104	0.0000

Classification Matrix - Model data				
Accuracy = 0.562				
PCC = 0.341				
	Reference			
Predicted	Cereal	Bar	Oatmeal	Total
Cereal	124	85	46	255
Bar	52	68	7	127
Oatmeal	79	21	180	280
Total	255	174	233	662

Classification Matrix - Model data				
Accuracy = 0.562				
PCC = 0.341				
	Reference			
Predicted	Cereal	Bar	Oatmeal	Total
Statistics by Class:				
Sensitivity	0.486	0.391	0.773	
Specificity	0.678	0.879	0.767	
Precision	0.486	0.535	0.643	

Classification Matrix - New data				
Accuracy = 0.583				
PCC = 0.342				
	Reference			
Predicted	Cereal	Bar	Oatmeal	Total
Cereal	45	24	20	89
Bar	18	25	0	43
Oatmeal	21	8	57	86
Total	84	57	77	218
Statistics by Class:				
Sensitivity	0.536	0.439	0.740	
Specificity	0.672	0.888	0.794	
Precision	0.506	0.581	0.663	

13.5.1 Interpreting Coefficients

Coefficient estimates in a standard MNL model are interpreted **relative to the reference brand**. A positive coefficient means that higher values of the predictor increase the likelihood of choosing that brand relative to the baseline.

To aid interpretation, coefficients can also be expressed as **relative risk ratios (RRRs)**. RRRs greater than 1 indicate increased relative likelihood, while values below 1 indicate decreased likelihood. These interpretations are often more intuitive for managerial audiences.

```
mnl_eval_rrr <- eval_std_mnl(model = mnl_fit, exp = TRUE,
                             newdata = test, ft=TRUE)
mnl_eval_rrr$coef_table
```

LR chi2 (8) = 288.1568; p < 0.0001					
McFadden's Pseudo R-square = 0.2004					
y.level	term	RRR	std.error	statistic	p.value
Bar	(Intercept)	2.4187	0.3257	2.7118	0.0067
Bar	genderMale	0.8082	0.2064	-1.0318	0.3022
Bar	maritalUnmarried	1.8454	0.2124	2.8849	0.0039
Bar	lifestyleInactive	0.4554	0.2090	-3.7627	0.0002
Bar	age	0.9750	0.0067	-3.8055	0.0001
Oatmeal	(Intercept)	0.0112	0.4597	-9.7722	0.0000
Oatmeal	genderMale	0.9776	0.2095	-0.1080	0.9140
Oatmeal	maritalUnmarried	0.6772	0.2367	-1.6469	0.0996
Oatmeal	lifestyleInactive	1.3754	0.2157	1.4777	0.1395
Oatmeal	age	1.0832	0.0078	10.3104	0.0000

13.5.2 Classification Performance

Beyond fit statistics, classification results help assess how well the model predicts observed choices.

The output includes:

- Overall accuracy
- Proportional Chance Criterion (PCC)
- Product-specific sensitivity, specificity, and precision

These metrics help identify which brands are easier or harder to predict based on observed covariates.

13.5.3 Holdout Sample Evaluation

Evaluating the model on a test sample provides insight into how well it generalizes to new data. Large discrepancies between training and test performance may indicate overfitting.

In practice, marketing data often contain substantial noise, so perfect prediction is neither expected nor required for managerial usefulness.

13.6 Predicted Probabilities

Coefficients and classification tables are not always the most intuitive outputs for decision-makers. Predicted probabilities translate model results into directly interpretable quantities.

We use the `pp_std_mnl()` function from the `MKT4320BGSU` package to compute and visualize average predicted probabilities for a focal predictor.

Usage:

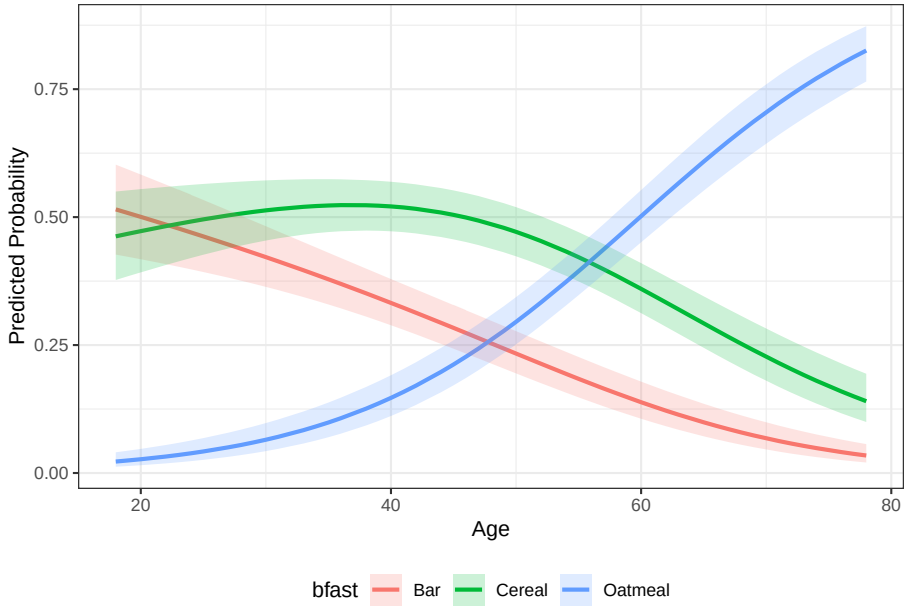
- `pp_std_mnl(model, focal, xlab = NULL, ft_table = FALSE)`
- where:
  - `model` is a fitted `multinom` model.
  - `focal` is a character string; name of the focal predictor variable.
  - `xlab` is an optional character string; label for the x-axis in the plot.
  - `ft_table` is logical; if TRUE, return the probability table as a flextable (default = FALSE).

Below, examples are provided for `age` (a continuous variable) and `lifestyle` (a categorical variable).

```
pp_age <- pp_std_mnl(model = mnl_fit, focal = "age", xlab  = "Age")
pp_age$table
```

```
# A tibble: 9 x 5
    age bfast   p.prob lower.CI upper.CI
  <dbl> <chr>    <dbl>    <dbl>    <dbl>
1    31 Cereal  0.516    0.459     0.573
2    31 Bar     0.413    0.357     0.472
3    31 Oatmeal 0.0708   0.0473    0.105
4    49 Cereal  0.479    0.431     0.527
5    49 Bar     0.243    0.204     0.287
6    49 Oatmeal 0.277    0.234     0.326
7    67 Cereal  0.266    0.218     0.320
8    67 Bar     0.0856   0.0604    0.120
9    67 Oatmeal 0.649    0.590     0.704
```

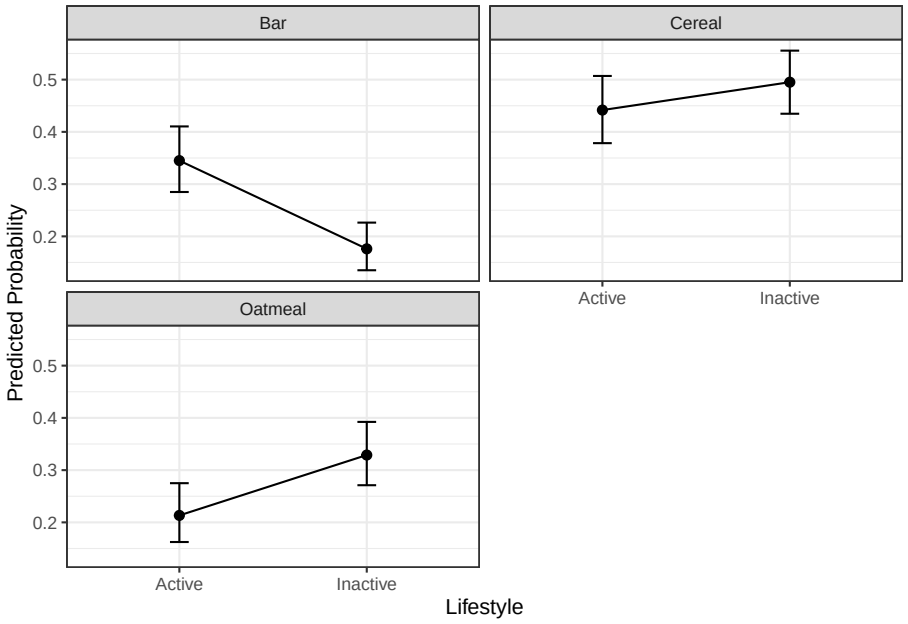
```
pp_age$plot
```




```
pp_lifestyle <- pp_std_mnl(model = mnl_fit, focal= "lifestyle",
                           xlab = "Lifestyle")
pp_lifestyle$table
```

```
# A tibble: 6 x 5
  lifestyle bfast   p.prob lower.CI upper.CI
  <fct>     <chr>    <dbl>    <dbl>    <dbl>
1 Active    Cereal   0.442    0.378    0.507
2 Active    Bar      0.345    0.285    0.410
3 Active    Oatmeal  0.213    0.162    0.275
4 Inactive  Cereal   0.495    0.435    0.556
5 Inactive  Bar      0.176    0.135    0.226
6 Inactive  Oatmeal  0.329    0.271    0.392
```

```
pp_lifestyle$plot
```



Predicted probabilities help answer questions such as:

- How does increasing price shift brand choice probabilities?
- Which brands are most sensitive to changes in a given variable?

13.7 Marketing Interpretation

The standard multinomial logit model provides a powerful yet accessible framework for understanding brand choice. It allows marketers to:

- Compare competitive positioning across brands
- Assess price and promotion sensitivity
- Translate statistical estimates into actionable probabilities

However, it also has limitations, including restrictive substitution patterns across alternatives.

13.8 Summary

In this chapter, you learned how to:

- Estimate a standard multinomial logit model
- Evaluate model fit and predictive performance
- Interpret coefficients and relative risk ratios
- Use predicted probabilities for marketing insight

The standard MNL model serves as a foundational tool in marketing analytics and provides a benchmark against which more advanced choice models can be compared.

13.9 What's Next

In this chapter, we treated brand choice as a function of consumer-level characteristics and marketing variables that affect all alternatives in the same way. This approach works well as a baseline, but it imposes an important limitation: it assumes that predictors influence every brand symmetrically.

In the next chapter, we relax this restriction by introducing the alternative-specific multinomial logit (AS-MNL) model. This framework allows predictors—such as price, promotions, or product attributes—to vary by brand, more closely reflecting how consumers actually evaluate competing options.

You will learn how to: - Specify predictors that differ across alternatives - Interpret coefficients that are specific to each brand - Compare alternative-specific results to the standard MNL model - Gain deeper insight into competitive positioning and substitution patterns

Alternative-specific MNL models provide a major step forward in realism and interpretability, especially in marketing settings where attributes like price, availability, or features differ meaningfully across brands.

Chapter 14

Alternative-Specific Multinomial Logit Models

14.1 Introduction: Why Alternative-Specific MNL?

In the last chapter, we modeled brand choice using standard multinomial logit (MNL) models, where all predictors were **case-specific**. That is, they described the consumer or choice situation and took the same value for all brands in a given choice set.

In many real marketing applications, however, the most important predictors vary **by brand**. Examples include:

- Price of each brand
- Package size
- Sugar content or nutritional attributes
- Promotional indicators
- Brand-specific features

Alternative-specific multinomial logit (AS-MNL) models allow us to include these variables directly, providing richer managerial insight into how brand attributes drive choice.

In this chapter, you will learn how to:

- Work with long-format choice data
- Split alternative-specific data correctly into training and test samples
- Estimate an alternative-specific MNL model

- Evaluate model fit and classification performance
- Interpret predicted probabilities and marginal effects in a marketing context

Throughout the chapter, we will use the **yogurt** dataset.

14.2 The Yogurt Choice Data

The yogurt dataset records consumer brand choices in repeated choice situations. Each row represents **one alternative within one choice situation**, not a single consumer.

Key implications:

- Each choice situation appears multiple times (once per brand)
- Exactly one alternative is chosen per choice set
- Many predictors vary across brands within the same choice set

This "long" structure is required for alternative-specific MNL models and differs from the wide-format data used earlier in the course.

14.3 Preparing the Data for Modeling

14.3.1 Why Splitting Is Different for Choice Data

With alternative-specific data, we **cannot** randomly split rows into training and test sets. Doing so would break apart choice sets and contaminate model evaluation.

Instead, we must split at the **choice-set level**, ensuring that all rows belonging to the same choice situation stay together.

14.3.2 Creating Training and Test Samples

We still use the `splitsample()` function from the `MKT4320BGSU` package, which supports group-level splitting. Whereas before we didn't use several parameters, will will use them for alternative specific MNL.

Usage:

- `splitsample(data, outcome = NULL, group = NULL, choice = NULL, alt = NULL, p = 0.75, seed = 4320)`
- where:
  - `data` is the data frame to split, in long-format.
  - `outcome` is NOT (USUALLY) USED FOR ALTERNATIVE SPECIFIC MNL
  - `group` is the grouping variable (e.g., choice situation id or respondent id). If provided, splitting is done at the group level. Required for alternative specific MNL.
  - `choice` is the 0/1 (or TRUE/FALSE) indicator for the chosen alternative. Used only when `group` is provided. Required for alternative specific MNL.
  - `alt` is the optional alternative label/ID. Used with `choice` to stratify at the group level. Required for alternative specific MNL.
  - `p` is the proportion of observations to place in the training set. Must be strictly between 0 and 1. Default is 0.75.
  - `seed` is the random seed for reproducibility. Default is 4320.

Before, we were interested in the `$train` and `$test` data frames. Now, we are interested in the `train.mdata` and `test.mdata` objects that are saved. They are in the format needed for the using `mlogit` (see below). However, to avoid a console error, you'll access the a slightly different way.

```
sp <- splitsample(data = yogurt, group = "id", choice = "choice", alt = "brand")

train <- sp[["train.mdata"]]
test  <- sp[["test.mdata"]]
```

At this point:

- `train` contains complete choice sets for model estimation
- `test` contains unseen choice sets for out-of-sample evaluation

---

14.4 Specifying an Alternative-Specific MNL Model

In an alternative-specific MNL model:

- Case-specific variables enter once

- Alternative-specific variables enter as brand-varying predictors

We use the `mlogit` function from the `mlogit` package to estimate the model. We separate the alternative specific from the case specific variables with a `|`. Alternative specific come first, then the case specific. We can use the base R `summary()` function to get the raw log-odds estimates.

```
library(mlogit)
as_mnl_fit <- mlogit(choice ~ price + feat | income, data = train)
summary(as_mnl_fit)
```

```
Call:
mlogit(formula = choice ~ price + feat | income, data = train,
    method = "nr")

Frequencies of alternatives:choice
  Dannon   Hiland   Weight  Yoplait
0.401988 0.029818 0.229155 0.339039

nr method
8 iterations, 0h:0m:0s
g'(-H)^-1g = 0.000171
successive function values within tolerance limits

Coefficients :
                      Estimate Std. Error  z-value  Pr(>|z|)
(Intercept):Hiland   0.7587200  0.5677111   1.3365  0.181401
(Intercept):Weight  -0.0263906  0.2078931  -0.1269  0.898986
(Intercept):Yoplait -3.9886941  0.2679762 -14.8845 < 2.2e-16 ***
price               -0.4424450  0.0295572 -14.9691 < 2.2e-16 ***
feat                 0.4230830  0.1491240   2.8371  0.004552 **
income:Hiland       -0.1081164  0.0149201  -7.2464 4.281e-13 ***
income:Weight       -0.0114764  0.0037707  -3.0436  0.002338 **
income:Yoplait       0.0729207  0.0040281  18.1030 < 2.2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Log-Likelihood: -1618.4
McFadden R^2:  0.23972
Likelihood ratio test : chisq = 1020.6 (p.value = < 2.22e-16)
```

Interpretation notes:

- Coefficients reflect changes in **relative utility**

- Signs and magnitudes should be interpreted in marketing terms
- Alternative-specific variables capture within-choice substitution effects

---

14.5 Evaluating Model Performance

14.5.1 Model Fit and Coefficients

We use the `eval_as_mnl()` function from the `MKT4320BGSU` package to obtain fit statistics, coefficients (both log-odds and odds ratio), and classification diagnostics.

Usage:

- `eval_as_mnl(model, digits = 4, ft = FALSE, newdata = NULL, label_model = "Model data", label_newdata = "New data", class_digits = 3)`
- where:
  - `model` is a fitted mlogit model.
  - `digits` is an integer; decimals to round coefficient and fit results (default 4).
  - `ft` is logical; if TRUE, return coefficient and classification tables as flextable objects (default FALSE).
  - `newdata` is an optional `dfidx` object (e.g., `test.mdata`) for an additional classification matrix. If NULL, only the training-data matrix is produced.
  - `label_model` is a character string label for the training-data classification matrix (default "Model data").
  - `label_newdata` is a character string label for the newdata classification matrix (default "New data").
  - `class_digits` is an integer; decimals to round classification results (default 3).

Key outputs include:

- Log-likelihood χ^2 test
- McFadden's pseudo R^2
- Odds ratios for interpretation
- Classification accuracy and diagnostics

```
as_eval <- eval_as_mnl(as_mnl_fit, ft = TRUE, newdata = test)
as_eval$coef_table
```

LR chi2 (5) = 1020.5649; p < 0.0001					
McFadden's Pseudo R-square = 0.2397					
term	logodds	OR	std.error	statistic	p.value
(Intercept):Hiland	0.7587	2.1355	0.5677	1.3365	0.1814
(Intercept):Weight	-0.0264	0.9740	0.2079	-0.1269	0.8990
(Intercept):Yoplait	-3.9887	0.0185	0.2680	-14.8845	0.0000
price	-0.4424	0.6425	0.0296	-14.9691	0.0000
feat	0.4231	1.5267	0.1491	2.8371	0.0046
income:Hiland	-0.1081	0.8975	0.0149	-7.2464	0.0000
income:Weight	-0.0115	0.9886	0.0038	-3.0436	0.0023
income:Yoplait	0.0729	1.0756	0.0040	18.1030	0.0000

```
as_eval$classify_model
```

Classification Matrix - Model data					
Accuracy = 0.621					
PCC = 0.330					
	Reference				
Predicted	Dannon	Hiland	Weight	Yoplait	Total
Dannon	577	39	324	97	1037
Hiland	1	12	0	2	15
Weight	18	2	38	18	76
Yoplait	132	1	53	497	683
Total	728	54	415	614	1811
Statistics by Class:					
Sensitivity	0.793	0.222	0.092	0.809	
Specificity	0.575	0.998	0.973	0.845	
Precision	0.556	0.800	0.500	0.728	

Classification Matrix - Model data					
Accuracy = 0.621					
PCC = 0.330					
	Reference				
Predicted	Dannon	Hiland	Weight	Yoplait	Total

```
as_eval$classify_newdata
```

Classification Matrix - New data					
Accuracy = 0.607					
PCC = 0.331					
	Reference				
Predicted	Dannon	Hiland	Weight	Yoplait	Total
Dannon	199	14	104	38	355
Hiland	2	2	1	1	6
Weight	8	1	12	13	34
Yoplait	33	0	21	152	206
Total	242	17	138	204	601
Statistics by Class:					
Sensitivity	0.822	0.118	0.087	0.745	
Specificity	0.565	0.993	0.952	0.864	
Precision	0.561	0.333	0.353	0.738	

14.5.2 Classification Performance

Classification is evaluated at the **choice-set level**:

- The predicted brand is the one with the highest predicted probability
- Accuracy reflects correct brand predictions
- PCC provides a baseline comparison

This approach mirrors how managers think about predicting actual consumer choices.

14.6 Predicted Probabilities and Marginal Effects

14.6.1 Why Predicted Probabilities Matter

Coefficients are not always intuitive. Predicted probabilities translate the model into outcomes managers care about:

- Market shares
- Brand switching
- Competitive responses

14.6.2 Why Marginal Effects Are Useful

Marginal effects quantify how much choice probabilities change in response to a small change in an attribute, holding everything else constant. Marginal effects can be computed in two common ways:

- **At observed values (Average Marginal Effects, AME)**
  Marginal effects are calculated for each observation using its actual attribute values and then averaged.
- **At means (Marginal Effects at the Mean, MEM)**
  Marginal effects are calculated at a single "average" profile, where each attribute is set to its sample mean.

Both approaches summarize how sensitive choice probabilities are to changes in attributes, but they differ in interpretation.

Marginal effects **at observed values**:

- Reflect the full distribution of the data
- Avoid relying on a potentially unrealistic "average consumer"
- Are often preferred for descriptive and policy interpretation

Marginal effects **at means**:

- Are easier to reproduce by hand or with software defaults
- Provide a clear, single reference point
- Can be useful for illustrating model mechanics and comparing effects across variables

The marginal effects tables can therefore answer questions such as:

- "On average, how does a $1 increase in price affect brand choice?"
- "How would choice probabilities change for a typical consumer if an attribute increased slightly?"
- "Which brands are most sensitive to changes in a specific attribute?"

In practice, the choice between observed values and means depends on the goal of the analysis. For interpretation and real-world impact, average marginal effects at observed values are often preferred. For teaching, demonstration, or simplified comparisons, marginal effects at means can be equally informative.

14.6.3 The `pp_as_mnl()` Function

For both case-specific and alternative-specific predictors, we use the `pp_as_mnl()` function from the `MKT4320BGSU` package to get both predicted probabilities and marginal effects.

Usage:

- `pp_as_mnl(model,focal_var, focal_type = c("auto", "alt", "case"), grid_n = 25, digits = 4, ft = FALSE, marginal = TRUE, me_method = c("observed", "means"), me_step = 1)`
- where:
  - `model` is a fitted mlogit model.
  - `focal_var` is a character string name of the focal variable.
  - `focal_type` is a character string; one of "case", "alt", or "auto" (default = "auto").
  - `grid_n` is an integer; number of points used to construct the grid of focal values for predicted probability plots when the focal variable is continuous (default = 25).
  - `digits` is an integer; rounding for numeric output (default = 4).
  - `ft` is logical; if TRUE, return tables as flextable objects (default = FALSE).
  - `marginal` is logical; if TRUE, compute marginal effects (default = TRUE).
  - `me_method` is a character string; one of "observed" AME or "means" (default = "observed").
  - `me_step` is numeric; finite-difference step size for AME (default = 1).

14.6.4 Case-Specific Predictors

We first examine how a consumer-level variable affects brand choice probabilities.

```
pp_income <- pp_as_mnl(as_mnl_fit, focal_var = "income", ft = TRUE, me_method="means")
pp_income$me_table
```

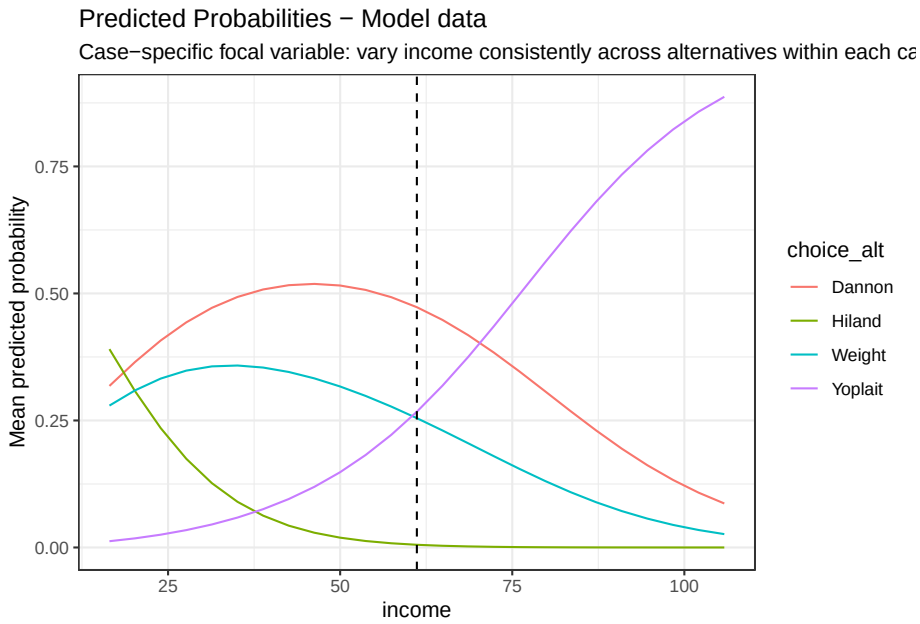
Marginal effects for ***income*** (at means)			
Dannon	Hiland	Weight	Yoplait
-0.0073	-0.0006	-0.0068	0.0147

```
pp_income$pp_table
```

Predicted Probability Table (income) - Model data				
focal_value	Dannon	Hiland	Weight	Yoplait
60.1438	0.4788	0.0060	0.2608	0.2544
61.1438	0.4729	0.0053	0.2545	0.2672
62.1438	0.4667	0.0047	0.2482	0.2804

Because ***income*** is continuous, the values shown include the mean and +/- 1 unit.

```
pp_income$pp_plot
```



14.6.5 Alternative-Specific Predictors

Now we examine a brand-specific variable such as price (a continuous variable) and feature (a categorical variable).

```
pp_price <- pp_as_mnl(as_mnl_fit, focal_var = "price", ft=TRUE, me_method="means")
pp_price$me_table
```

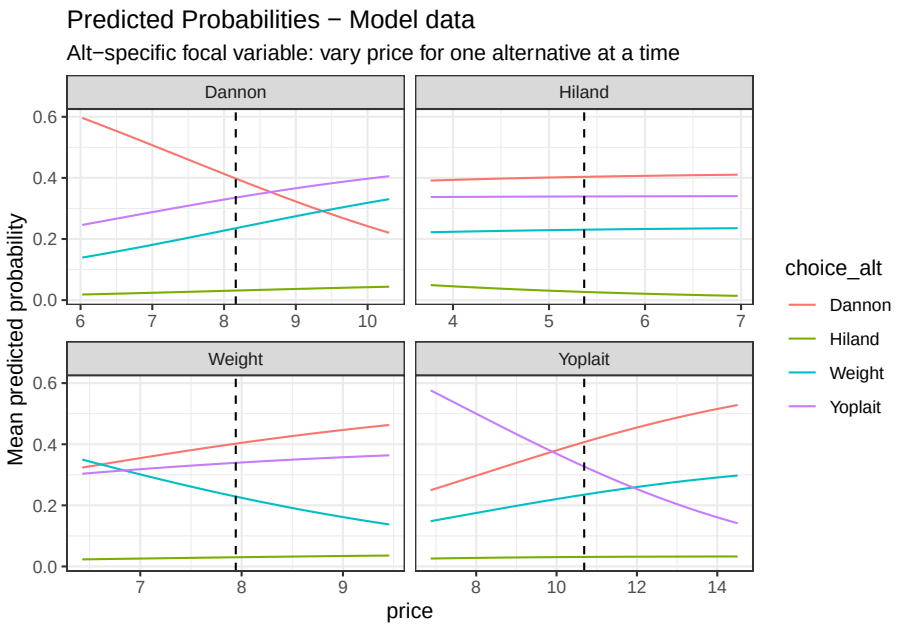
Marginal effects for ***price*** (at means)				
Alternative	Dannon	Hiland	Weight	Yoplait
Dannon	-0.1105	0.0010	0.0550	0.0545
Hiland	0.0010	-0.0021	0.0005	0.0005
Weight	0.0550	0.0005	-0.0845	0.0290
Yoplait	0.0545	0.0005	0.0290	-0.0840

```
pp_price$pp_table
```

Predicted Probability Table (price) - Model data					
varied_alt	focal_value	Dannon	Hiland	Weight	Yoplait
Dannon	7.1628	0.4918	0.0250	0.1883	0.2949
Dannon	8.1628	0.3980	0.0313	0.2353	0.3355
Dannon	9.1628	0.3088	0.0375	0.2821	0.3716
Hiland	4.3663	0.3966	0.0394	0.2258	0.3382
Hiland	5.3663	0.4035	0.0268	0.2305	0.3392
Hiland	6.3663	0.4083	0.0179	0.2338	0.3399
Weight	6.9421	0.3516	0.0256	0.3060	0.3169
Weight	7.9421	0.4019	0.0301	0.2287	0.3392
Weight	8.9421	0.4446	0.0340	0.1648	0.3566
Yoplait	9.6874	0.3673	0.0302	0.2138	0.3886
Yoplait	10.6874	0.4069	0.0311	0.2350	0.3269
Yoplait	11.6874	0.4438	0.0318	0.2545	0.2699

Because ***price*** is continuous, the values shown include the mean and +/- 1 unit.

```
pp_price$pp_plot
```




```
pp_feat <- pp_as_mnl(as_mnl_fit, focal_var = "feat", ft=TRUE, me_method="means")
pp_feat$me_table
```

Marginal effects for ***feat*** (at means)				
Alternative	Dannon	Hiland	Weight	Yoplait
Dannon	0.1057	-0.0010	-0.0526	-0.0521
Hiland	-0.0010	0.0020	-0.0005	-0.0005
Weight	-0.0526	-0.0005	0.0808	-0.0277
Yoplait	-0.0521	-0.0005	-0.0277	0.0804

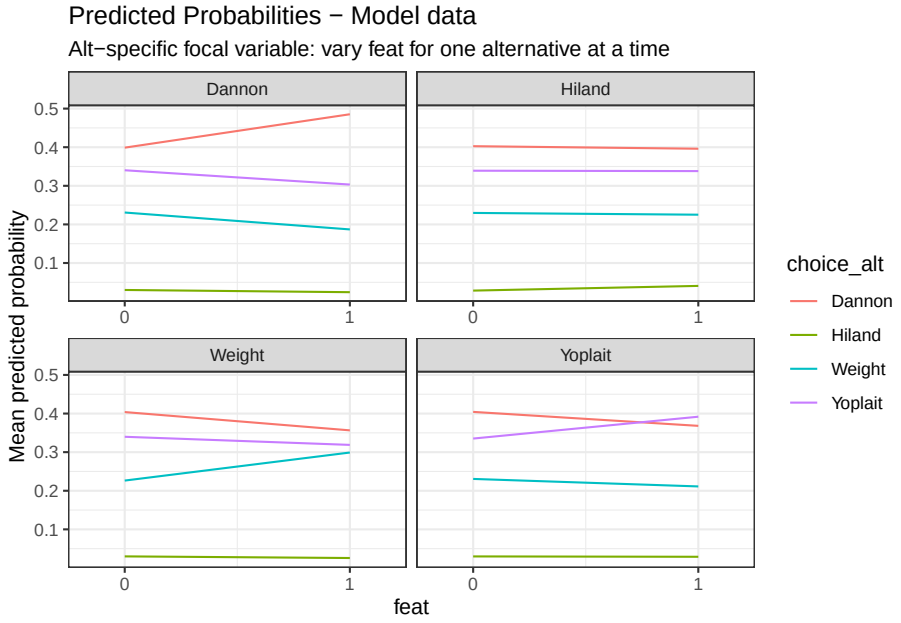
```
pp_feat$pp_table
```

Predicted Probability Table (feat) - Model data					
varied_alt	focal_value	Dannon	Hiland	Weight	Yoplait
Dannon	0	0.3988	0.0301	0.2308	0.3403

Predicted Probability Table (feat) - Model data					
varied_alt	focal_value	Dannon	Hiland	Weight	Yoplait
Dannon	1	0.4853	0.0244	0.1870	0.3033
Hiland	0	0.4027	0.0285	0.2297	0.3391
Hiland	1	0.3960	0.0407	0.2251	0.3381
Weight	0	0.4038	0.0300	0.2264	0.3398
Weight	1	0.3565	0.0257	0.2990	0.3188
Yoplait	0	0.4043	0.0299	0.2306	0.3352
Yoplait	1	0.3681	0.0289	0.2112	0.3918

Because ***feat*** is binary, only the two observed values are shown.

```
pp_feat$pp_plot
```



14.7 Managerial Insights

Alternative-specific MNL models allow managers to:

- Evaluate pricing and promotion strategies
- Understand competitive substitution patterns
- Predict market share changes under different scenarios

Compared to standard MNL models, AS-MNL models provide more realistic insights when brand attributes vary within choice sets.